PARALLELIZING UNSTRUCTURED SPARSE MATRIX COMPUTATIONS ON LARGE-SCALE MULTIPROCESSORS

BY SESHADRI VENUGOPAL

A dissertation submitted to the
Graduate School—New Brunswick
Rutgers, The State University of New Jersey
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
Graduate Program in Computer Science

Written under the direction of
Professor Gerard Richter
and approved by

__________________________________

__________________________________

__________________________________

__________________________________

__________________________________

New Brunswick, New Jersey
October, 1993
ABSTRACT OF THE DISSERTATION

Parallelizing Unstructured Sparse Matrix Computations on Large-scale Multiprocessors

by Seshadri Venugopal, Ph.D.

Dissertation Director: Professor Gerard Richter

Problems in the class of unstructured sparse matrix computations are characterized by highly irregular dependencies and communication patterns that are not known at compile-time, but can be completely determined at run-time before the computations are actually performed. For this class of problems, current parallelizing compilers are unable to produce efficient code on large-scale distributed memory MIMD multiprocessors, and manual techniques are inflexible and too ad hoc to be generally effective. In this thesis, we propose a run-time automatic partitioning and scheduling methodology for unstructured sparse matrix computations on large-scale multiprocessors. Our methodology is based on extracting information from the problem instance by pre-processing its symbolic structure, and using this information to achieve high performance in repeated iterations of the computations during which the symbolic structure is unchanged. We present efficient software tools to help users build their parallelization system by following this methodology.

We demonstrate the efficacy of our methodology on sparse Cholesky factorization, which has historically proven to be hard to parallelize. The highlight of our approach is a new two-dimensional block partitioning scheme. We build a run-time parallel system for block sparse Cholesky factorization called Sparse Hybrid Automatic Parallelization
Environment (SHAPE), consisting of a parallel partitioner, a parallel scheduler and a parallel communication optimization algorithm. These are modular tools tied together by an explicit representation for block-based unstructured computations. We employ SHAPE to carry out an extensive experimental study of sparse Cholesky factorization on the iPSC/860. The experimental results show that with a judicious choice of partitioning parameters, our block-based partitioning and scheduling method outperforms a well-known column-based method in delivering high performance on a variety of structured and unstructured matrices. The preprocessing itself is shown to be very efficient, its cost being recovered in a small number of iterations of the factorization. Our methodology and tools may be used to parallelize other unstructured sparse matrix computations for which the same symbolic structure is used in several iterations of the computations. Such computations include sparse triangular solution and sparse matrix-vector multiplication.
Acknowledgements

During the seven-year journey to my Ph.D., several faculty members, graduate colleagues, friends and family have influenced my academic and psychological growth.

I would like to thank my committee members for bringing it all together. Vijay Naik was foremost in guiding me towards a thesis topic and thereafter collaborating closely with me in research. I thank him for sharing his ideas in problem-solving and offering me advice whenever I needed it, Computer Science or otherwise. He led by example, teaching me perseverance and efficiency, and the magic of achieving objectives by courage of conviction. Gerard Richter was extremely helpful in advising me at Rutgers, and insistently made me see all sides of any issue under debate. He inculcated in me the habit of objective thought. Apostolos Gerasoulis was one of my earliest teachers in Rutgers, to whom I am grateful for encouraging me to do research in parallel computing. I would also like to thank him for supporting me in my nascent research endeavors. Barbara Ryder provided the opportunity to broaden my interests by always making me feel welcome in her programming languages reading groups. I am indebted to her for helping me survive times of significant setbacks in my research. Joel Saltz took me under his wing for a stint at NASA Langley, and has since been a happy model of continual striving for bigger and better things.

Several other Computer Science faculty members have contributed towards my growth at Rutgers. Ken Kaplan and Ann Yasuhara have played special roles, lending support and advice at all times, especially during the more difficult periods. Diane Souvaine was most helpful in making me see the finer points of technical writing, and in guiding me towards my research topic. I would like to thank Chinh Hoang for bringing alive the wonders of algorithmic complexity, and Anthony Terrano for advising me in my first efforts at implementing non-trivial parallel algorithms on distributed memory
machines.

The Computer Science staff has been most wonderful in creating a familial atmosphere. I thank Priscilla Rasmussen for going out of her way to help me with support for conference travel and other administrative matters. Diana Creighton, Dotty Hinz, Elsie Jackson, Janet Hillis, Ida Vincler, Maureen Meyer, Mary Hoffman, and Carole Schickner made sure that, apart from other things, I got my paychecks in time! Lastly, I owe special thanks to Valentine Rolfe for her unconditional support during all these years, for joining me in suffering every disappointment and enjoying every bright period in my thesis journey, and ultimately, for playing a parental role at my wedding.

My graduate colleagues and friends have helped in various ways. Among them, Rakesh Sinha, Jaikumar Radhakrishnan, Yong-Fong Lee, Emmi Schatz, Jim Llewellyn, Izzy Nelken, Bhaskar Ghosh, Tao Yang, Steve Masticola, Souri Das, Arup Acharya, Ed Segall, Arun Balakrishnan have discussed technical matters with me on numerous occasions. Dan Arena has provided moral support over the years. Srimat Chakradhar has been a great friend and research companion, and has provided constant encouragement. I thank him for sharing ideas so very often, and for being there all the time. Kumar Vadaparty has been an easy-going room-mate, a great colleague, and a most understanding friend, whom I have always counted on to clear the cobwebs from my mind. Dawn Cohen has brightened the duller day with her cheerful demeanor, and I thank her for many a simulating conversation. Vinayak Karnataki has been a super friend, whose company I have grown to enjoy enormously.

I thank my non-Computer Science friends for sharing their thoughts and feelings with me, and for taking me out of Computer Science for forays into strange and exciting worlds. They have had a big part to play in trying to make me a balanced individual. I thank my family for encouraging me to go my academic way, especially my father from whom I have developed a passion for learning.

Finally, I would like to thank my wife, Zehra, for her complete understanding and unconditional support at all times.
Dedication

To my parents
## Table of Contents

Abstract ................................................................. ii
Acknowledgements ..................................................... iv
Dedication ................................................................. vi

List of Tables .......................................................... viii
List of Figures .......................................................... ix

1. Introduction ......................................................... 1
   1.1. Unstructured sparse matrix computations ..................... 2
   1.2. Thesis contributions ............................................. 3
   1.3. Dissertation organization ....................................... 9

2. Motivation .......................................................... 10
   2.1. Sequential dense Cholesky factorization ...................... 10
   2.2. Sequential sparse Cholesky factorization ...................... 11
   2.3. Manual parallelization ......................................... 15
   2.4. Compiler techniques for parallelization ...................... 16
   2.5. Methodology and tools for run-time parallelization .......... 18

3. Representation for block-based unstructured computations ....... 19
   3.1. Partitioning of unstructured computations .................. 20
   3.2. The Block Computation Representation (BCR) for algorithms .. 21
       3.2.1. The Attribute-Value Representation (AVR) for data blocks . 24
       3.2.2. Representation for computational tasks .................. 25
3.2.3. The Block Dependency directed Acyclic Graph (BDAG) representation for precedence relationships  

3.3. Conclusions .................................................................................................................. 30

4. Analysis of column and block partitioning for sparse Cholesky factorization  

4.1. Models for parallel sparse Cholesky factorization ................................................. 33

4.2. Two dimensional regular grids .............................................................. 34

4.2.1. Column assignment schemes ............................................................ 36

4.3. Results for column-based partitioning ................................................. 36

4.4. Results for block-based partitioning ................................................. 39

4.4.1. Sparse BLOCC scheme ......................................................... 40

4.4.2. Sparse BLOCC scheme for message-passing multiprocessors ... 41

Performance model and time analysis .................................................. 41

Number of messages for distributed sparse BLOCC .......................... 55

4.5. Conclusions ............................................................................................................. 58

5. Block partitioning for sparse Cholesky factorization  

5.1. Introduction ............................................................................................................. 61

5.2. Distributed column sparse Cholesky factorization ........................................... 63

5.3. Pre-partitioner ............................................................................................... 68

5.3.1. Cluster identification .............................................................................. 71

5.3.2. Rectangle identification ........................................................................... 75

5.3.3. Parameters .................................................................................................. 79

5.4. Block sparse Cholesky .................................................................................... 80

5.4.1. Block partitions ........................................................................................ 80

5.4.2. Task types ............................................................................................... 82

5.4.3. Block sparse Cholesky factorization .................................................... 83

5.4.4. Block dependencies ............................................................................ 85

Projection ........................................................................................................... 87
Characterization of dependencies ........................................... 88

5.4.5. Block structure .......................................................... 92
5.4.6. Code for task computations ........................................... 96
5.4.7. BCR for block sparse Cholesky factorization ................. 97

5.5. Partitioner ................................................................. 100
5.5.1. Triangle and rectangle partitioning ............................. 101
5.5.2. Computation of dependencies .................................... 105
5.5.3. Parallelizing the partitioner ...................................... 117

5.6. Conclusions ............................................................... 120

6. Scheduling and communication optimization .......................... 121

6.1. Allocation for block sparse Cholesky factorization ............ 123
6.1.1. Allocation of cluster ................................................ 123
6.1.2. Selection of cluster and allocation of column ............... 124

6.2. Scheduling for unstructured computations ....................... 126
6.2.1. Scheduling pre-pass ................................................ 126
6.2.2. Parallel static scheduling using primitives .................. 133
6.2.3. Primitives for scheduling ........................................ 134
   Primitive SET_UP ....................................................... 135
   Primitive READY_BLOCK ............................................. 135
   Primitive GET_BLOCK ................................................ 136

6.3. Scheduling for block sparse Cholesky factorization .......... 137
6.3.1. Scheduling pre-pass for block sparse Cholesky factorization 137
6.3.2. Parallel static scheduling of block sparse Cholesky factorization
   using primitives ..................................................... 139

6.4. Communication optimization ........................................... 139
6.4.1. Communication model using a static schedule ............... 141
6.4.2. A scalable deadlock-free strategy ............................... 143
6.4.3. Extension to windows of receives ................................ 148
6.4.4. An approximation to the deadlock-free strategy
6.4.5. Optimization on the iPSC/860
6.5. Conclusions

7. Experimental evaluation of distributed block sparse Cholesky factorization using SHAPE

7.1. Overview of SHAPE
7.2. Performance measures
7.3. Testing environment
7.3.1. SHAPE: implementation
7.3.2. Distributed block-based factorization: implementation
7.3.3. Column-wrap: implementation
7.3.4. Test matrices
7.4. Numerical factorization timings
7.4.1. Column-wrap: numerical factorization
7.5. Single-processor performance
7.5.1. Task computational speed
7.5.2. Overheads
7.5.3. Improvement of performance
7.5.4. Column-wrap: single processor performance
7.6. Communication performance
7.6.1. Messages and volume
7.6.2. Communication optimization
7.7. Idle time
7.8. Comparison with a distributed multifrontal code
7.9. Selection of partitioning parameters
7.9.1. Cluster width \( W \)
7.9.2. Grain size \( G \)
7.10. SHAPE: Preprocessing cost
7.11. Conclusions ...................................................... 202

8. Sparse triangular systems and sparse matrix-vector multiplication . 213
  8.1. Block sparse triangular systems .................................. 214
    8.1.1. Dependencies ............................................... 214
            Lower triangular system .................................. 215
            Upper triangular system .................................. 216
    8.1.2. A hybrid fan-out and fan-in algorithm ...................... 218
    8.1.3. BCR .......................................................... 221
            AVR for blocks ............................................ 221
            Computational tasks ...................................... 223
            BDAG .......................................................... 224
  8.2. Sparse matrix-vector multiplication .............................. 224
  8.3. Conclusions ..................................................... 227

9. Conclusions ......................................................... 228

References .............................................................. 231

Vita ................................................................. 237
List of Tables

5.1. The twelve generic update and scale tasks. ............................. 82
5.2. Examples of update tasks. ............................................. 83
7.1. Test matrices. ......................................................... 167
7.2. Block factorization timings, in milliseconds, for unstructured matrices. 169
7.3. Block factorization timings, in milliseconds, for structured matrices. 171
7.4. Column-wrap factorization timings, in milliseconds, for unstructured matrices. 173
7.5. Column-wrap factorization timings, in milliseconds, for structured matrices. 174
7.6. Single-processor performance. ....................................... 175
7.7. Computational rates for tasks. ....................................... 176
7.9. Structured grid matrices for communication performance evaluation. 183
7.10. Communication for structured grid matrices. ......................... 184
7.11. Communication for unstructured matrices. ........................... 187
7.12. Factorization timings in milliseconds for B15, large grain size. .... 191
7.13. Partitioner and scheduler timings. .................................. 200
7.14. Size of partitioning. .................................................. 201
8.1. Communication-space tradeoffs for block triangular solves. ........... 219
List of Figures

2.1. A column-oriented Cholesky factorization algorithm .......................... 11
2.2. Column-oriented sparse Cholesky factorization ................................. 12
2.3. Inter-element dependencies in Cholesky factorization .......................... 13
2.4. Lower triangular factor and dependencies ........................................ 13
2.5. CSC representation .......................................................................... 14
3.1. AVR for CSC storage ......................................................................... 25
3.2. Illustration of algorithm Construct BDAG .......................................... 28
3.3. Construction of the BDAG ................................................................. 31
4.1. Nested dissection ordering on a $7 \times 7$ grid ..................................... 35
4.2. $+\$-separator for grid of size $i \times i$ ................................................. 38
4.3. Clique corresponding to separator of size $m$ ..................................... 44
4.4. External work, triangular block .......................................................... 46
4.5. External work, rectangular block ....................................................... 48
4.6. Separator S' updates separator S .......................................................... 56
5.1. An example of zero-nonzero structure in a $49 \times 49$ matrix ................ 64
5.2. Sparse $L$ and elimination tree .......................................................... 65
5.3. Supernodes in the lower triangular factor ......................................... 66
5.4. Clusters and rectangles ................................................................. 69
5.5. Dense triangles and rectangles ......................................................... 70
5.6. Procedure for identification of clusters ............................................. 73
5.7. Contour traced by algorithm Id_Clusters .......................................... 75
5.8. Procedure for identification of rectangles ........................................ 77
5.9. Unit blocks ....................................................................................... 81
5.10. Block update tasks ......................................................................... 84
5.11. Block sparse Cholesky factorization. .......................... 85
5.12. Inter-block dependencies in sparse Cholesky factorization. .......................... 86
5.13. Block computation, column-major and row-major. .......................... 94
5.14. Column-major and row-major major access. .......................... 95
5.15. Code for RRR, RRT, TRR, RRC and RC. .......................... 97
5.16. Code for RT. .......................... 97
5.17. Code for TR. .......................... 97
5.18. Partitioner schematic. .......................... 100
5.19. Partitioning of a dense triangle and rectangle into unit blocks. .......................... 101
5.20. Procedure for partitioning of a triangle. .......................... 102
5.21. Procedure for partitioning of a rectangle. .......................... 104
5.22. An interval tree. .......................... 106
5.23. Procedure for building an interval tree. .......................... 109
5.24. Query with a vertical strip. .......................... 111
5.25. Procedure for querying an interval tree. .......................... 111
5.26. Procedure for computing base dependencies. .......................... 112
5.27. Procedure for computing base-multiplier dependencies. .......................... 114
5.28. Procedure for computing intra-cluster triangle dependencies. .......................... 115
5.29. Procedure for computing intra-cluster rectangle dependencies. .......................... 116
5.30. Procedure for processing a column. .......................... 117
5.31. Procedure for processing a cluster. .......................... 118
5.32. Parallel partitioner. .......................... 119
6.1. Allocation of blocks in cluster. .......................... 124
6.2. Selection of cluster for allocation. .......................... 125
6.3. Procedure to release an edge in a BDAG. .......................... 129
6.4. Main procedure to schedule a BDAG. .......................... 131
6.5. Distributed algorithm with primitives. .......................... 133
6.6. AVR structures. .......................... 135
6.7. Scheduling pre-pass for block sparse Cholesky factorization. .......................... 138
6.8. Parallel block sparse Cholesky factorization .................................. 139
6.9. Deadlock situation ............................................................................ 145
6.10. Synchronous message passing calls .................................................. 153
6.11. Asynchronous receive ....................................................................... 153
7.1. Solution of a sparse linear system using SHAPE .................................. 158
7.2. Functional diagram of SHAPE ............................................................ 159
7.3. Column-wrap numerical factorization code ........................................ 166
7.4. Variation of block factorization time with processors: unstructured ma-
    trices ................................................................................................. 170
7.5. Variation of block factorization time with processors: structured matrices. 172
7.6. Grain sizes of natural rectangles ........................................................ 178
7.7. Vector lengths and locality in computation .......................................... 179
7.8. Overheads in execution ...................................................................... 180
7.9. Coalescing rectangles ....................................................................... 181
7.10. Communication messages: block vs. column-wrap ............................. 185
7.11. Communication volume: block vs. column-wrap ................................. 186
7.12. Communication growth rate: unstructured matrices ............................ 188
7.13. Unoptimized communication relative to computation ......................... 189
7.14. Optimized communication cost ......................................................... 205
7.15. Optimized communication relative to computation ............................ 206
7.16. Factorization time before and after communication optimization, 32 pro-
    cessor ............................................................................................... 206
    and L127 .......................................................................................... 207
7.17. Factorization time before and after communication optimization, B15
    and L127 .......................................................................................... 207
7.18. Idle time before and after communication optimization ........................ 208
7.19. Comparison of block with multifrontal .............................................. 209
7.20. Effect of \( W \) on total factorization time .......................................... 209
7.21. Effect of \( W \) on computation time .................................................... 209
7.22. Effect of \( W \) on communication time ................................................ 210
7.23. Effect of $W$ on idle time. ........................................... 210
7.24. Number of CC tasks. ..................................................... 210
7.25. Number of RRR tasks. .................................................... 210
7.26. Cost per floating point operation. ..................................... 211
7.27. Effect of $G$ on uniprocessor performance. ......................... 211
7.28. Effect of $G$ on factorization time. .................................... 211
7.29. Effect of $G$ on communication time. .................................. 212
7.30. Effect of $G$ on idle time. ............................................. 212
7.31. Preprocessing and factorization times. ............................... 212
8.1. Dependencies in block forward solve. ................................. 215
8.2. Dependencies in block back solve. .................................... 216
8.3. Rectangle for triangular systems. ...................................... 222
Chapter 1

Introduction

In recent years, the commercial success of large-scale general-purpose parallel architectures has made them the machines of choice for high performance computing. These machines are built by employing up to hundreds of low-cost processors with distributed memory, with the goal of delivering high performance at a lower price-to-performance ratio than traditional supercomputers. They support the Multiple Instruction Multiple Data (MIMD) program model, and the programs on different processors communicate with each other via message-passing. These machines are also called massively parallel processors (MPPs) or scalable architectures. For the sake of uniformity, we will refer to them as large-scale multiprocessors. Current large-scale multiprocessors include the nCUBE/2, iPSC/860, Paragon XP/S and CM-5. While the hardware has steadily progressed toward delivering teraflop performance, the development of efficient compilers and programming tools to help users parallelize their applications on large-scale multiprocessors has lagged far behind.

In any application, the parallelism can be exploited at varying levels of granularity. At the finest level of granularity, the parallelism is exploitable by distributing single instructions or operations among the processors. In the worst case, this could result in inter-processor communication at the end of every instruction. On current large-scale multiprocessors, the inter-processor communication cost is much higher than the computation cost on a processor, making such instruction-level or operation-level parallelism infeasible. For these machines, multiple operations need to be grouped together into atomic tasks to be performed locally on a processor so that a higher computation to communication ratio is achieved, and the cost of communication is amortized over computational work. We will refer to the resulting parallelism as coarse-grained in
nature. This is a relative term. The goal is to balance the computation against the 
communication so as to achieve high performance, and the absolute coarseness of grain 
required to attain this balance is determined by the communication to computation 
ratio of the architecture. Continuing the grouping of operations beyond this balance 
point would lead to loss in parallelism, and in the extreme case the entire computation 
would end up as one single task. Partitioning or dividing data and computations into 
atomic units, and distributing them among processors in order to achieve high per-
formance is a difficult task in general. Unstructured sparse matrix computations are 
among the problems that are hardest to parallelize.

1.1 Unstructured sparse matrix computations

By unstructured computations, we mean computations that are characterized by highly 
irregular data dependencies and communication patterns that are not known at compile 
time. In data-driven computations, the total work is fixed and known before execution, 
but the order of execution of the constituent tasks is determined only at execution-
time by the availability of data on which these tasks operate. Problems in the area of 
unstructured, sparse matrix computations are typically data-driven in nature. These 
problems can grow to extremely large sizes, making their parallelization on large-scale 
multiprocessors an attractive proposition. Sparse Cholesky factorization is one such 
problem, and is an important kernel in a variety of engineering and optimization appli-
cations. Distributed sparse Cholesky factorization has received close attention in recent 
years, and is known to be hard to parallelize on large-scale multiprocessors.

Current parallelizing compilers for distributed memory multiprocessors are unable 
to produce efficient code for unstructured sparse matrix computations. Research in 
manual parallelization techniques, on the other hand, has been very active, spurred on 
by the observation that there is a large amount of parallelism available at the level of 
arithmetic operations, sufficient to keep hundreds of processors busy at all times during 
the entire computation. This has led to several interesting parallel algorithms. However, 
the experience in the last several years indicates that the performance of these algo-
rithms on the existing large-scale multiprocessors is below par, and decreases rapidly
as the number of processors increases. There are several reasons for the poor performance. Although there is an abundance of parallelism, it is most easily extractable at the smallest level of arithmetic operations, such as multiply or add. Identifying and scheduling these operations appropriately at run-time is prohibitively expensive, given the extremely large problem sizes. Even if one were to ignore the scheduling cost, the amount of communication during execution would easily offset any gain in parallelism, even on a relatively small number of processors. The goal, then, is to increase the granularity of distributable computations so as to reduce the parallelization cost to manageable proportions and achieve high performance. However, extracting the necessary parallelism at larger levels of granularity becomes increasingly difficult due to the unstructured nature of dependencies, given that manual techniques have limited flexibility and are too ad hoc to be effective for the large variety of problems in this class. Lastly, modern RISC-based architectures deliver high performance only when cache utilization is high and the vector units are used efficiently. Both requirements demand data structures that use memory in a regular and structured fashion. This is not possible with the conventional representations for sparse matrix computations.

To achieve high performance for unstructured sparse matrix computations on general purpose large-scale multiprocessors, we need to build a partitioning and scheduling system that would provide an efficient and effective interface between the problem and the coarse-grained architecture. The system should, at run-time, take the sequential, instruction-level algorithm and generate efficient coarse-grain distributed code with message-passing. Currently we are working on developing such a system. This thesis proposes a framework on which such a system may be built. In the rest of the thesis, we will use the term unstructured computations to mean unstructured sparse matrix computations.

1.2 Thesis contributions

We have developed a run-time methodology to parallelize unstructured computations on large-scale multiprocessors. Within this framework, the main thesis contributions are:
1. A new representation for block-based unstructured computations, called Block Computation Representation, or BCR, for short.

2. A parallel partitioner for block-based sparse Cholesky factorization.

3. A parallel scheduler for block-based unstructured computations, including a deadlock-free parallel communication optimization algorithm. The scheduler is designed in the style of an SPMD (Single-Program-Multiple-Data) program. A set of primitives serves as an interface between the application program and the scheduler.

4. A parallel partitioning and scheduling system for block-based sparse Cholesky factorization. This system is called Sparse Hybrid Automatic Parallelization Environment, or SHAPE, for short.

Of these contributions, the partitioner is specific to sparse Cholesky factorization. The BCR and the scheduler, including the communication optimization algorithm, are applicable to the general class of unstructured computations in which data is partitioned into blocks and the owner computes rule is used to perform the computations. These computations include, apart from sparse Cholesky factorization, sparse triangular system solution and sparse matrix-vector multiplication. The general BCR and the scheduler are presented in the dissertation. Apart from this, their specific instances for sparse Cholesky factorization are covered as a part of SHAPE, along with the partitioner and an allocator which is specific to sparse Cholesky factorization. In the following, we elaborate on these contributions and discuss their role as inter-related tools in the methodology.

The symbolic structure of the computations in a problem instance is available at run-time, before the actual numerical computations are performed. Our methodology is based on extracting information from the problem instance by preprocessing its symbolic structure at run-time, and using this information to achieve high performance in repeated iterations of the computations during which the symbolic structure is unchanged. There are several applications in which the same symbolic structure is repeatedly used for computations with varying data values. These include circuit
simulation, numerical solution of partial differential equations, linear programming, control systems, Newton methods for nonlinear optimization problems. Some of these are described in [39, 65, 66]

The methodology follows a two-phase process of partitioning and scheduling. The partitioning phase and a part of the scheduling phase constitute the preprocessing portion of the methodology. The rest of the scheduling phase, including the optimized communication support for the computations, is combined with the numerical computations.

In the partitioning phase, the data domain is divided into blocks, and the overall computation is divided into tasks. Blocks are the atomic units of distribution of data to processors, and tasks are the atomic computational units which operate on these blocks. The output of the partitioning phase is the Block Computation Representation (BCR). The BCR is a graph representation that we have designed for expressing the computations and precedence relationships in unstructured problems in which data is partitioned into blocks.

In the scheduling phase, the computations are executed in parallel, given the BCR. The scheduling phase is further divided into allocation and scheduling of computations and inter-processor communication. In the rest of the thesis, we use the term scheduler to mean the algorithm or software that schedules the computations and communication, after the allocation has been done. In the allocation step, the data blocks are distributed among the processors. A processor is said to “own” the data blocks which have been allocated to it. The BCR, and the methodology overall, assumes the “owner-computes” rule, i.e. all computations that modify a block of data are performed in the processor that owns that block. Thus, the allocation of tasks to processors is completely determined by the allocation of data blocks to processors. The scheduling of computations and communication is broken up into two stages: a pre-pass, which results in a static schedule on every processor, and the subsequent passes in which the static schedule is followed in order to execute the computations. The static schedule is a complete ordering of the tasks and communication. Apart from this, in order to optimize the
communication, the scheduler sets up the appropriate infrastructure during the pre-pass stage. This is then used by a communication optimization algorithm to provide support during the computations in the second stage. We refer to this as execution-time support. The allocation and scheduling pre-pass are the additional preprocessing steps after partitioning.

**Representation for block-based unstructured computations:** We have introduced a novel explicit representation called the Block Computation Representation, or BCR for short. We examined two other representations, the Directed Acyclic Graph (DAG) [80] and the Minimally Constrained Task Graph (MCTG) for fine-grain Gaussian elimination [62], before devising the BCR and tailoring it for the class of block-based unstructured computations using the owner computes rule. The BCR has three components: (a) a representation for data blocks called Attribute-Value Representation, or AVR for short; (b) a representation for tasks in the form of updates to blocks; and (c) a representation for the precedence relationships among tasks called Block Dependency directed Acyclic Graph, or BDAG for short. The BCR serves as a connecting thread among the modules of the methodology. It is described in Chapter 3, and its construction for block-based sparse Cholesky factorization is described in Chapter 5.

**Parallel partitioner for block-based sparse Cholesky factorization:** We have developed a parallel partitioner for the sparse Cholesky factorization of any general sparse matrix [73, 74, 75], which partitions the symbolic factor into a hybrid mixture of sparse columns, dense triangles and dense rectangles. Tuning knobs in the form of parameters are provided in order that the partitioner may be ported across problems with widely ranging sparsity structures and across architectures with different granularities. Thus, the partitioner is highly flexible, and, as far as we are aware, is the first general block-based parallel partitioner for sparse Cholesky factorization. The partitioner is discussed in Chapter 5. Two-dimensional block-based partitioning for dense matrices is known to have superior asymptotic communication behavior compared to column-based partitioning, and possesses better cache utilization properties [68, 52, 14]. However, exploiting these features in the Cholesky factorization of general sparse matrices is a complex task and requires sophisticated automatic tools. Naik and Patrick [52]
have proposed a two-dimensional block partitioning scheme for the Cholesky factorization of sparse matrices arising from a class of structured grid problems, and has shown that the data traffic complexity for this scheme is optimal. In Chapter 4 we prove that this block partitioning scheme is more scalable than column partitioning for structured grid problems. Our parallel partitioner is based on this blocking scheme, but adapts itself closely to the matrix structure and the architecture granularity.

Parallel scheduler for block-based unstructured computations: The scheduler assumes that the blocks, and therefore the tasks, have been allocated to processors. It is divided into two stages. In the first, or pre-pass stage, it makes one pass over the algorithm, as represented by the BCR, and "executes" the algorithm in parallel, in SPMD style. It uses a dynamic, data-driven strategy to locally schedule, on every processor, the execution of tasks and the messages sent from and received on that processor. In the pre-pass, all precedence constraints among the tasks as represented by the BDAG are obeyed. However, the actual numerical computations associated with the tasks are performed only if required. If the computations are performed, the numerical answers to the problem are available at the end of the pre-pass itself. During the pre-pass, the scheduler maintains a record of the order in which the tasks were executed and the order in which messages were sent and received on every processor. Apart from this, it records, for every message, the size of the message, and its destination if the message is sent out from the processor. The output of the pre-pass is, therefore, a static task and communication schedule for each processor.

In the subsequent passes, the numerical computations are executed in parallel using the static schedule generated by the pre-pass. The overheads of manipulating data structures in order to track the arrival of data on a processor, and the resultant activation of tasks is avoided by following the pre-computed schedule. Apart from this, the complete a priori knowledge of inter-processor communication allows for aggressive communication optimization in which the latency of message passing may be reduced significantly. We have designed a deadlock-free, parallel communication optimization algorithm for unstructured computations, as a part of this second phase of the scheduler. We have designed a set of efficient primitives that provide an
interface to the scheduler during the pre-pass and to the message-passing with communication optimization during the subsequent numerical computations, making the distribution of data and the underlying architectural details transparent to the user. These primitives also alleviate the programming effort on the part of the user, enabling the user to write sequential-looking SPMD code.

The methodology is structured in the spirit of the run-time compilation approach pioneered by Saltz [78]. In [76], we showed the effectiveness of this two-phase scheduling approach of pre-pass with execution-time support, including the use of primitives, on column-based sparse Cholesky factorization. The general scheduler and its specific application to block-based sparse Cholesky factorization is discussed in Chapter 6.

**Parallel partitioning and scheduling system for block-based sparse Cholesky factorization:** Following our methodology, we have built a parallel software system for partitioning and scheduling block-based sparse Cholesky factorization on the iPSC/860. The system is called SHAPE, for Sparse Hybrid Automatic Parallelization Environment. An overview of SHAPE is presented in Chapter 7. SHAPE is built by integrating the partitioning and scheduling tools listed above, appropriately tailored for block-based sparse Cholesky factorization. It functions both as a preprocessor prior to Cholesky factorization, and as an execution-time support tool for the numerical factorization. As far as we are aware, SHAPE is the first distributed, general block-based partitioning and scheduling system for the Cholesky factorization of structured and unstructured sparse matrices. It provides the required infrastructure to experiment with a variety of structured and unstructured matrices, with very little effort on the part of the user. This infrastructure provides parametric control over the partitioning and scheduling steps, and an exhaustive set of performance figures. We have employed SHAPE to carry out a thorough experimental study of block sparse Cholesky factorization on the iPSC/860. The results are compiled in Chapter 7. The results verify that the block-based partitioning and scheduling method is more scalable than column-based methods in communication. The results also show that high performance on a variety of structured and unstructured matrices can be obtained with an appropriate choice of partitioning parameters. Moreover, the results of our experimental study provide
an insight into block partitioning and scheduling, so that general principles may be developed.

1.3 Dissertation organization

In Chapter 2, we briefly discuss current parallelizing compilers and manual techniques for parallelizing unstructured problems, and provide the motivation for our research. Chapter 3 discusses coarse-grain data and task partitioning and introduces the BCR. In Chapter 4, we first summarize the known theoretical results for distributed sparse Cholesky factorization and then present our analysis of block partitioning for the sparse Cholesky factorization of structured grid problems. Chapter 5 describes the design of a parallel partitioner for block-based sparse Cholesky factorization.

In Chapter 6 we first describe the allocator for block-based sparse Cholesky factorization. We then give the design of a general parallel scheduler for unstructured problems, and describe its application to block-based sparse Cholesky factorization. A parallel, deadlock-free algorithm for communication optimization is presented, along with the design of the primitives for scheduling. Chapter 7 describes SHAPE and presents an extensive experimental evaluation of block-based distributed sparse Cholesky factorization on the iPSC/860. Chapter 8 sketches the extensions of the methodology to the solution of sparse triangular systems and sparse matrix-vector multiplication. Chapter 9 presents concluding remarks and proposes some topics for future research.

Every algorithm in this thesis is specified as either sequential or parallel. All the parallel algorithms proposed are data parallel, in the sense that data is partitioned and distributed among processors, and the computations on the processors operate on their local data in SPMD mode. Thus, each parallel algorithm is presented as a single-processor code, with the understanding that the same code executes on all processors in parallel.
Chapter 2
Motivation

The goal of our work is to build a run-time system for the parallelization of unstructured sparse matrix computations on large-scale multiprocessors. For this class of problems, manual parallelization techniques are inflexible due to their ad hoc nature, while current compilers need adequate run-time support to generate efficient code. In this chapter, we first introduce sparse Cholesky factorization, a problem which is representative of the class of unstructured sparse matrix computations. We preface the discussion of sparse Cholesky factorization with a description of dense Cholesky factorization. We then discuss the limitations of manual techniques, briefly describe current compiler-based parallelization techniques and provide a justification for the parallelization approach proposed in this thesis.

2.1 Sequential dense Cholesky factorization

The problem of solving a linear system of equations, $Ax = b$, arises in a large number of scientific applications, and may be solved using direct or iterative methods. If the coefficient matrix $A$ is symmetric, positive definite, the system can be solved by directly computing the Cholesky factor of $A$. The Cholesky factorization of $A$ is given by $A = LL^T$, where the Cholesky factor $L$ is a lower triangular matrix with positive diagonal elements. Then, the solution to the system is obtained by solving the following the two triangular systems $Ly = b$ and $L^T x = y$.

The left part of Figure 2.1 shows a sequential, column-oriented Cholesky factorization algorithm for an $N \times N$ matrix $A$ [24]. In this and other codes which follow, it is assumed that the entries of $L$ are initialized to the corresponding entries of $A$. The outermost for loop can be treated as steps or stages of factorization. At step $k$ of
\[
\begin{align*}
\text{for } k = 1 \text{ to } n \text{ do} \\
L_{kk} &= \sqrt{L_{kk}} \\
\text{for } i = k + 1 \text{ to } n \text{ do} \\
L_{ik} &= L_{ik} / L_{kk} \\
\text{end for} \\
\text{for } j = k + 1 \text{ to } n \text{ do} \\
\text{for } i = j \text{ to } n \text{ do} \\
L_{ij} &\leftarrow L_{ij} - L_{ik} * L_{jk} \\
\text{end for} \\
\text{end for} \\
\text{end for}
\end{align*}
\]

Figure 2.1: A column-oriented Cholesky factorization algorithm.

the computation, column \( k \) of \( L \) is formed by scaling the elements of column \( k \) by the reciprocal of the square root of the diagonal element. The columns of \( L \) that are to the right of column \( k \) are then updated by column \( k \) in the doubly nested \( j, i \) loop. In the literature, the scaling operations on column \( k \) are grouped into a single \texttt{cdiv}(k) task and the update of a column \( j \) using column \( k \) is defined to be a \texttt{cmod}(j, k) task [27]. The code on the right hand part of Figure 2.1 uses these tasks. This version in which a column is factored and immediately used to update all columns to its right is called the sub-matrix form of Cholesky factorization [28].

2.2 Sequential sparse Cholesky factorization

When the system of equations is sparse, several other considerations come into play in designing the factorization algorithm. As the factorization proceeds from left to right, entries which were originally zeros may be replaced by non-zero values - this is called \textit{fill-in}. The lower triangular factor must have enough space to store the fills generated during factorization. For an efficient implementation of factorization, it is required to know the zero-nonzero structure of \( L \) before factorization so that the appropriate amount of space can be allocated a priori. The step of \textit{symbolic factorization} efficiently computes the structure of \( L \). The sequential column-oriented sparse Cholesky factorization code may be written as shown in Figure 2.2.
for \( k = 1 \) to \( n \) do \\
\[ L_{k,k} = \sqrt{L_{k,k}} \] \\
foreach \( i \in [k + 1, n] \) s.t. \( L_{i,k} \neq 0 \) do \\
\[ L_{i,k} = L_{i,k} / L_{k,k} \] \\
end foreach \\
foreach \( j \in [k + 1, n] \) s.t. \( L_{j,k} \neq 0 \) do \\
foreach \( i \in [j, n] \) s.t. \( L_{i,k} \neq 0 \) do \\
\[ L_{i,j} \leftarrow L_{i,j} - L_{i,k} * L_{j,k} \] \\
end foreach \\
end foreach

end for

Figure 2.2: Column-oriented sparse Cholesky factorization.

As in sequential dense sparse Cholesky factorization, the outermost for loop can be treated as steps or stages of the factorization. At step \( k \) of the computation, column \( k \) of \( L \) is formed by scaling its non-zero elements by the reciprocal of the square root of the diagonal element. Next, only those columns \( j \) to the right of \( k \) for which \( L_{j,k} \) is non-zero are updated. Also, within a column \( j \) which is updated, only those elements \( L_{i,j} \) that have a corresponding non-zero in element \( L_{i,k} \) need to be updated. The computational definitions of cdiv and cmod are the same as in the dense case, except that in the sparse factorization, only the non-zero elements are involved in all operations. Again, the factorization scheme is the submatrix form in which each column, as soon as it is factored, updates all the columns to its right that depend on it.

To understand the details of the column-oriented sparse Cholesky factorization algorithm, we show the generic element-level data dependencies in Figure 2.3. In that figure, the direction of the arrows indicates the direction of data flow. Thus, elements \( L_{j,k} \) and \( L_{i,k} \) from column \( k \) of the factor \( L \) are required in computing element \( L_{i,j} \). 
\[ L_{i,j} = L_{i,j} - L_{i,k} * L_{j,k} \] is the corresponding operation in the Cholesky factorization algorithm. Note that in computing the final value of \( L_{i,j} \), it must be updated by all pairs of non-zero elements \( L_{j,k} \) and \( L_{i,\bar{k}} \), \( 1 \leq \bar{k} < j \). Since there is no particular structure to a general sparse matrix, the element-level dependencies are also unstructured. If we consider the column-based cmod and cdiv tasks as indivisible units of computation, we
get column-level dependencies. Column $j$ depends on column $k$ if at least one element of column $j$ depends on at least one element of column $k$, for its update. Again, the column-level dependencies are unstructured.

Figure 2.4 shows the sparsity structure of a $5 \times 5$ matrix, $L$. The x’s represent the positions of the non-zeros in $A$ and the circles show the positions of the non-zeros which are filled-in during numerical factorization. The dependencies among the columns are shown by the graph with the directed edges representing precedence relationships as well as flow of data. Note that the directed edges also represent the `cmax` tasks. Consider the column 5, for instance, which is represented by the node labeled 5 in the graph.
The directed edges into 5, viz. (1, 5), (3, 5) and (4, 5) represent the tasks \( \text{cmod}(5, 1) \), \( \text{cmod}(5, 3) \) and \( \text{cmod}(5, 4) \) respectively. These tasks can take place in any relative order with respect to each other. Only, no two of them take place simultaneously, since all of them update the same column. The execution of tasks represented by the graph are completely data-driven, and the partial ordering among them is dictated solely by the precedence constraints, with no implicit or explicit synchronization.

Special data structures are needed for the efficient storage and manipulation of sparse matrices. The CSC (compressed storage of columns) format is one such commonly used data structure. In the CSC format, the \( N \times N \) sparse matrix \( L \) is represented using two “parallel” arrays: \( \text{Lval} \) and \( \text{Lstruct} \), and a third, auxiliary array, called \( \text{Lindex} \). Figure 2.5 shows the CSC format for the matrix of Figure 2.4.

\[
\begin{array}{c}
\text{Lval} \\
1 \times \times \times \times \times \times \times \times \times \\
\text{Lstruct} \\
1 \ 3 \ 5 \ 2 \ 3 \ 3 \ 5 \ 6 \ 4 \ 5 \ 5 \ 6 \ 6 \\
\text{Lindex} \\
1 \ 4 \ 6 \ 9 \ 11 \ 13 \ 14
\end{array}
\]

Figure 2.5: CSC representation.

Array \( \text{Lval} \) stores the non-zero values in column major order. For each non-zero value, the corresponding row number is stored in the parallel \( \text{Lstruct} \) array. \( \text{Lindex} \) gives pointers to the beginning of the list of non-zero elements for each column. Thus, the non-zero elements in column \( j \) are stored in the \( \text{Lval} \) array from location \( \text{Lindex}[j] \) through \( \text{Lindex}[j + 1] - 1 \). For instance, \( \text{Lindex}[3] = 6 \) is the index in \( \text{Lval} \) and \( \text{Lstruct} \) at which the first non-zero element for column 3 is stored and \( \text{Lindex}[4] = 9 \) is the index in \( \text{Lval} \) and \( \text{Lstruct} \) at which the first non-zero for column 4 is stored. Thus, \( \text{Lval}[6] \) through \( \text{Lval}[8] \) store the non-zero values in column 3 and \( \text{Lstruct}[6] \) through \( \text{Lstruct}[8] \) hold the corresponding row numbers. Here, 3, 5, and 6 are the respective row numbers of the non-zero elements in column 3. Row numbers are sometimes also referred to as row indices.
2.3 Manual parallelization

The advantage of manual parallelization techniques lies in their ability to make use of domain information and problem structure to exploit parallelism. But for large problems, manual techniques have to be confined to simple partitioning and scheduling schemes. For Cholesky factorization, much work has been done in parallelization based on simple column partitioning [5, 11, 23, 24, 43, 81].

In [55], a study was conducted to explain the poor performance of the then existing column-based sparse Cholesky factorization algorithms on message-passing systems. The conclusion was that this poor performance is mainly due to the poor communication performance of the current distributed memory message-passing architectures relative to their floating point speed. In other words, column-wise partitioning of data is too fine-grain for the current generation of large-scale multiprocessors. Over the past few years, while large-scale multiprocessors have evolved into more and more powerful compute engines, the communication cost has remained very high. This has to increasing communication to computation ratios. For instance, the iPSC/860 has a higher communication to communication ratio than the iPSC/2, an earlier generation machine [35]. It is evident that if we want to see high performance for any algorithm on current large-scale multiprocessors, we need to exploit parallelism at a coarse grain level. Apart from this, the partitioning or decomposition of tasks and data has to be sensitive to the machine granularity, so that the same partitioner may be ported to different architectures with different granularities. A study of various algorithmic components and performance aspects of PDE sparse solvers is presented in [51]. An outcome of the study was the observation that sparse matrix techniques should be used in different ways at different stages of the factorization because the problem's nature varies from very sparse to dense during the solution process. One way to deal with this is to use the structure of the matrix to partition the matrix differently for different stages of the factorization process.

It would be impossible to manually partition and schedule large problems efficiently
by taking into account the structure of the problem and the architectural costs. Furthermore, experimentation with different partitionings or scheduling strategies would be too time-consuming to be practical. Thirdly, for a given task and data partitioning and distribution, manual scheduling of computations and message-passing would require significant effort and a knowledge of the architecture. These drawbacks make effective manual parallelization impractical for most realistic problems.

2.4 Compiler techniques for parallelization

Fortran D, a version of Fortran enhanced with data decomposition specifications [36], provides constructs for specifying regular and irregular distribution of arrays to processors. An irregular distribution is described at run-time by a mapping function. The compiler decomposes a program automatically into functional parts based on the data decomposition and the owner computes rule, where every processor only performs computation on data it owns. The Fortran D program is converted into single-program multiple data (SPMD) programs with explicit message-passing. The compiler handles both regular and irregular computations. Computations which can be accurately characterized at compile-time are called regular computations; those which cannot are described as irregular computations.

The communication messages required for irregular computations cannot be decided at compile-time. A technique for run-time preprocessing of loops is proposed in [63, 40], given that the dependencies are known before entering the loop at run time, and that the dependencies do not change in the course of executing the loop iterations. The preprocessing step or inspector constructs communication schedules which are then realized as communication messages by the executor which uses these schedules during loop execution. Both the inspector and executor are supported by a library of run-time message-passing procedures called Parti [13]. The language ARF (ARguably Fortran) [78] has been designed by taking Fortran 77 and adding extensions to serve as an interface between application programs and Parti primitives.
Unstructured sparse matrix computations are characterized by indirect array referencing. Lu and Chen [47] propose a hybrid compiler and run-time approach for parallelizing loops with indirections and pointers. In this approach, the scheduler itself is generated by the compiler with run-time support, while Saltz’s approach used in Parti is purely run-time, with the scheduling code being composed of hand-written run-time library routines.

Jade [58] is a high-level, implicitly parallel language designed for coarse-grain, task-level concurrency. To parallelize an application, the programmer has to specify a decomposition of data into atomic units, a decomposition of the sequential program into tasks and a description of how each task will access data. The implementation then automatically extracts and exploits the task-level concurrency. The authors demonstrate how Jade exploits concurrency in sparse Cholesky factorization.

Fortran M [21], a language derived from Fortran 77 by means of message-passing extensions, facilitates a modular or object-oriented approach to parallel program design. The extensions include constructs for defining program modules called processes, for specifying processes that are to execute concurrently, for establishing typed, one-to-one communication channels between processes, and for sending and receiving messages on channels. Fortran M is built on the paradigm of task parallelism, and can be used to coordinate multiple data-parallel computations.

In recent work [12], the authors Choudhury et. al. assert that parallel computing demands support from software that precisely and effectively captures the structure of the application for better performance. They define about ten broad classes of computations, each of which is large enough to warrant individually tailored category-specific software support. The class of problems we address in our work falls in a category termed *implicit multiphase loosely synchronous computations* by the authors. These problems exhibit irregular inter-iteration dependencies that are not known at compile-time but can be fully predicted at run-time before the program enters the irregular loop which carries out the computations. Examples of problems that fall in this category include sparse triangular solves and sparse direct factorization, such as Cholesky factorization.
2.5 Methodology and tools for run-time parallelization

The trend is toward developing automatic run-time support software and linking it to compilers, in order to parallelize large real-world unstructured sparse matrix problems. Increasing emphasis is being placed on having the compiler itself generate parts of the run-time software.

In [12], the authors conclude that for the implicit multiphase loosely synchronous problems, there is still a clear need for development of appropriate run-time support targeted toward SIMD and MIMD distributed memory architectures. Our goal is to start building such a system for large-scale multiprocessors that support the MIMD programming model. In this thesis, we propose a parallelization methodology and a set of tools that may be used to build a run-time software system for parallelizing unstructured computations. This system may either be used as a stand-alone package, or may be linked to compilers in the form of a library. The following set of tools are needed in order to build such a system.

- A partitioner to decompose the data and computations by appropriately "sensing" the structure of the data domain, and by taking into account the granularity of the architecture. Our experience in parallelizing sparse Cholesky factorization leads us to believe that a good partitioning that can lead to high performance requires significant knowledge of the problem structure.

- A tool to allocate or distribute tasks and data among the processors, with the aim of reducing communication cost without compromising on the load balance. This tool must be able to quantify the goodness of any given data and task distribution.

- A scheduler to generate a schedule of the computations and communication for all the processors, with adequate interface support to hide the architectural details from the application programmer.

- A representation for unstructured computations which would tie all of these together into an integrated run-time system.
Chapter 3

Representation for block-based unstructured computations

The partitioning of an algorithm for a distributed memory large-scale multiprocessor can be defined as the division of computations and data into tasks and data blocks respectively. A task is a group of computational operations. Tasks are atomic units of computation i.e. when a task begins execution, it must finish execution without interruption. A block is any group of data elements. Data blocks are the atomic units of the data domain i.e. data is distributed among the processors in units of data blocks. An algorithm may be partitioned in either of two ways: computations may be partitioned first, which then determines the data partitioning, or data may be partitioned first, followed by the partitioning of computations.

We choose to partition the data first, which then determines the partitioning of computations into tasks. We assume the owner computes rule, i.e. all computations that update a block are performed in the processor to which the block is mapped. We study the issue of representing an unstructured computation in which data has been partitioned into blocks, and the owner computes rule is assumed. In this study, we examine two other representations, the Directed Acyclic Graph (DAG) [80] and the Minimally Constrained Task Graph (MCTG) for fine-grain Gaussian elimination [62], neither of which assume the owner computes rule. We propose a new representation, called the Block Computation Representation (BCR), for block-based, unstructured computations, that is different from both the DAG and the MCTG.
3.1 Partitioning of unstructured computations

Sarkar [64] has dealt with task partitioning at the computation graph level. One can start with the entire algorithm as one task and recursively split the computations, thereby unrolling the graph. Or, starting with the statement level dependency graph, one could coalesce nodes together into bigger tasks. Sarkar defines a cost function for a partitioned graph, minimizing which leads to a good partition.

A coarse-grain partition has a longer critical path whereas a fine-grain partition has a higher overhead component. The greater the communication overhead in a multiprocessor, the coarser should be the partition. A partition with minimum cost function does not guarantee minimum parallel execution time, but any optimal partition is guaranteed to have a parallel execution time which is within a factor of two of the optimal. The problem of partitioning to minimize the cost function is shown to be NP-complete. We do not know of any task partitioning heuristics with guaranteed performance bounds. Yang and Gerasoulis [30] have performed a formal analysis of granularity for task graphs and propose a new definition of the granularity for a DAG which captures the trade-off point between parallelization and sequentialization. It is not known as to how this granularity analysis may be used to partition unstructured computations.

In parallelizing numerical scientific algorithms, good data partitioning is crucial for high performance. In recent years, the evolution of RISC architectures for large-scale multiprocessors has made this even more important. To gain the most out of the cache and vector units in these RISC processors, data has to be accessed in a structured manner. Algorithms for scientific computations typically work on matrices and data partitioning techniques are based on how data is accessed by the algorithm over the period of its execution. Computational tasks are then associated with these data partitions. The column-wise partitioning of a matrix for factorization algorithms has been successfully used in dense matrix factorization. In order to obtain the same partitioning of tasks and data by using a general task partitioning scheme such as Sarkar's, the scheme would need to group together computations into a task in such a
manner that no two computations in a task access data which lie in different columns of the matrix.

Apart from the data access pattern, the size and shape of data partitions also depends on the architecture. Dongarra et al. [16] cite several methods of row-oriented and column-oriented partitionings for matrix multiplication and LU decomposition, choosing the partitioning depending on the vector machine used. The different sets of BLAS (Basic Linear Algebra Subprograms) routines also give an indication as to how partitions change depending on hardware. Out of the original BLAS set grew the Level 2 BLAS for vector-processing machines. These aimed at matrix-vector operations. Finally, for computers with a hierarchy of memory and true parallel processing capabilities, it is desirable to partition matrices into blocks and to perform the computations by using matrix-matrix operations on the blocks [15]. This approach avoids excessive movement of data to and from memory and gives a surface-to-volume effect for the ratio of operations to data movement.

From the preceding discussion it is clear that doing task partitioning on the computation graph first, and then fitting the data partitioning to the task partitioning is not advisable for scientific computations. The geometry of the data partitioning is crucial, and is not captured by computation graphs. For sparse matrix computations, it is even more important to achieve a good data partitioning. We will therefore take the approach of first carrying out the data partitioning, and then dividing the computation into tasks. We refer to a data partition as a data block. The computational tasks are atomic units of computation which update the data blocks, and are completely determined by the data partitioning and the element-level dependencies in the algorithm.

3.2 The Block Computation Representation (BCR) for algorithms

We consider unstructured computations in which the data space is partitioned into blocks, possibly in a non-uniform manner. We say that a partitioning is uniform if the size and shape of any partition can be formulated as a general function of some size
and shape quantification parameters; otherwise the partitioning is non-uniform. For instance, a partitioning of a dense lower triangular matrix into columns is uniform since the size and shape of any column is a function of the column number and the order of the matrix. An example of a non-uniform partitioning is a row-wise partitioning of a
general sparse matrix in which only the non-zeros of every row are stored. In this case,
the sizes of the rows have to be explicitly enumerated.

When the data partitioning is non-uniform, the precedence relationships among
tasks become unstructured. The entire computation in the execution of an algorithm is
divided into updates to blocks. Each update to a block is a computational task. A task
may use one or more blocks to update exactly one block. We use the macro-dataflow
model of task execution. In this model, a task starts executing only when all its input
data are available, and executes to completion without interruption. This model has
previously been used by Sarkar [64], Wu and Gajski [79], Lewis [19], and Yang and
Gerasoulis [30]. The order of execution of tasks is determined only by the precedence
relationships among them and the availability of data on which they operate, without
any kind of synchronization constraints. During the parallelization process, the data
blocks are distributed among or mapped to the processors. A data block is "owned" by
the processor to which it is mapped. The owner computes rule states that all changes
to the data values in a block may only be computed in the processor which owns that
block. We will use the owner computes rule in our parallelization methodology.

The directed acyclic dependency graph, DAG for short, has been traditionally used
to model computations in algorithms, and the dependencies among the computations.
Each node of a DAG represents a computational task. Each edge of a DAG, from a
source node to a destination node, enforces the precedence constraint that that the des-
tination task may not begin execution until the source task has finished execution. An
edge may, in addition, represent the flow of data from the source task to the destination
task. Nodes and edges are assigned weights. The weight of a node is a function of the
number of computational operations required to execute the corresponding task. The
weight of an edge is a function of the amount of data flowing from the output of the
source task to the input of the destination task. For a formal definition of a DAG, see
Yang [80].

There is one primary limitation in the DAG representation for algorithms in which each task is an update to some data block, and where the updates to a data block may take place in any relative order. Typical examples of such algorithms include LU factorization, Cholesky factorization, triangular solves in the solution of a linear system of equations, and matrix-vector multiplication. The DAG overconstrains such update tasks by forcibly sequentializing them in an arbitrary a priori order, because they are output dependent on each other. This sequentialization may not have any significant negative impact on the scheduling of computations for structured problems such as dense Cholesky factorization. However, for sparse matrix computations, we would prefer to use a representation which does not impose such apriori sequentialization. The appropriate, weaker constraint that is needed to be represented is that no two updates to the same data block may take place simultaneously. The actual order in which the non-simultaneous tasks may be executed is determined at execution time, depending on the order of arrival of the data blocks on which these tasks operate.

Sadayappan and Visvanathan [62] call this the “non-simultaneity” constraint. They incorporate this in a new representation to model parallel sparse Gaussian Elimination, called the Minimally Constrained Task Graph (MCTG). Each node of an MCTG represent a computation which updates one element of the matrix. Hence, it is a fine grain graph. An MCTG may contain both directed and undirected edges. The directed edges represent temporal dependency constraints, as in a DAG, and the undirected edges represent non-simultaneity constraints.

We introduce a new representation which captures the notion of non-simultaneity, but does not explicitly maintain the non-simultaneity edges. This is so because the owner computes rule is assumed, which automatically ensures that the non-simultaneity constraint among output dependent tasks is obeyed, since they must all take place on the same processor. Apart from this, we also represent data blocks in an explicit manner. This is done by storing, for each block, both the geometry of its shape, and its data values. Finally, although the representation is in the form of a precedence graph, it uses nodes to represent data blocks, and edges to represent tasks and precedence
relationships, unlike the DAG or the MCTG. We call this representation the Block Computation Representation, or BCR for short.

The BCR consists of the following components.

- A representation for data blocks called the Attribute-Value Representation, or AVR for short.
- A representation for computational tasks.
- A minimally constrained representation for precedence relationships and data flow among the computational tasks, called the Block Dependency directed Acyclic Graph, or BDAG for short.

These components are covered in detail in the following.

3.2.1 The Attribute-Value Representation (AVR) for data blocks

A data block is characterized by specifying two quantities: its static attributes and its dynamic values. The attributes of a block specify its shape and size and are fixed during the computation while the values of the elements in the data block may change. The attributes and values of a block could be split into several groups, each group stored separately. The attributes and values within each group are stored in a contiguous manner in memory. Each group is described by a structure called ATTR_DATA. A structure BLOCK_CHAR combines these into a single entity, by listing all the attribute groups and all the data groups. Most often, though, the attributes would all be in a single contiguous address space and the values would all also be in a single contiguous address space.

As an example, consider the column-wise partitioning of the sparse matrix of Figure 2.4. Each column of the matrix is a block. Assume that columns are wrap-mapped to three processors i.e. columns 1, 4 are mapped to processor 0, columns 2, 5 are mapped to processor 1 and columns 3, 6 are mapped to processor 2. Figure 3.1 shows the AVR for column 4.

Columns 1 and 4 are both mapped to processor 0 and the non-zeros for these two columns are stored in CSC format in two parallel arrays as shown in the leftmost
part of Figure 3.1. The lower array gives the row indices of the non-zeros in column 1 followed by the row indices of the non-zeros in column 4. The upper array gives the corresponding non-zero values, which in the figure are simply shown by x’s. The BLOCK_CHAR structure for column 4 is shown rightmost in that figure. There is one group of attributes and one group of values. The attributes group is of size 2 and consists of the row indices 4 and 5; the matching values group is also of size 2 and consists of the non-zeros in rows 4 and 5 of column 4.

Or, if we need to represent a rectangular data block, the attributes could be the co-ordinates of the top-left and bottom-right corner and the number of values would be equal to the size of the block, which in turn is equal to the area of the rectangle.

3.2.2 Representation for computational tasks

We build a representation for computational tasks. In the process, we define terminology and introduce notations for the BCR as a whole.

- The data space is a collection of disjoint data blocks. We use

\[ B = \{ B_i \}, \quad i = 1, 2, \ldots, |B|, \] to represent the set of blocks. Each block is represented using the AVR.
• A portion of a block is a simple sub-section of the block, represented using the AVR. We use the subset notation to denote the fact that a portion of a block is a sub-section of the block.

• The overall computation in the algorithm is divided into updates to blocks. Each update is a computational task, in which one or more updating blocks are used to update exactly one updated block. We use $\mathcal{U}$ to denote the set of all updates.

• Let $B_u \in \mathcal{B}$. We denote the set of updates to $B_u$ by $C_u = \{C_u^j\}, \ j = 1, 2, \ldots, |C_u|$. Note that $\mathcal{U} = \bigcup_u C_u$ where $u = 1, 2, \ldots, |\mathcal{B}|$.

• The number of updating blocks in the task $C_u^i$ is denoted by $\eta_u^i$. The task is represented by a tuple

$< R_u^i, R_u^i, 1, R_u^i, 2, \ldots, R_u^i, r >$, where $r = \eta_u^i$.

Here, $R_u^i$ is the portion of the block being updated, i.e. $R_u^i \subseteq B_u$, and each $R_u^{i,k}, \ k = 1, 2, \ldots, \eta_u^i$ is the portion of an updating block used in the update. No two of the updating portions can belong to the same updating block.

• Let the total number of tasks be $NT$. Each task $C_u^i$ is assigned a unique task number $t_u^i$, $1 \leq t_u^i \leq NT$.

• A task $C_u^i$ depends on another task $C_q^j$, denoted by $C_u^i \prec C_q^j$ if and only if $\exists r, \ 1 \leq r \leq \eta_u^i$, such that $R_u^i r \subseteq B_q^j$. Note that this allows a task to be dependent on itself, which would happen if a portion of the updated block itself is used to update it.

• Let $\mathcal{P}$ be the set of processors and $P = |\mathcal{P}|$ be the number of processors. We define the function $\Pi : \mathcal{B} \to \mathcal{P}$ to be a mapping of blocks to processors. The processor $\Pi(B_i)$ is the owner of block $B_i$.

• Let $T$ be the set of program modules which implement all the different types of update tasks in the computation. Each task is an instance of one of these program modules. Define a function $\Gamma : \mathcal{U} \to T$ which maps each task into a program module.
3.2.3 The Block Dependency directed Acyclic Graph (BDAG) representation for precedence relationships

A BDAG $G$ consists of a set of nodes, $\mathcal{N}$, and a set of directed edges, $\mathcal{E}$. Each node $n_i \in \mathcal{N}$ in a BDAG denotes a block, and each edge denotes a part of an update task. The mapping of nodes to blocks is given by the function $\Psi : \mathcal{N} \rightarrow \mathcal{B}$. There are two types of edges in the graph, Type I and Type II. An edge of Type I from $n_i$ to $n_j$ stands for an update of $\Psi(n_j)$ using $\Psi(n_i)$. Let $\Psi(n_j) = B_j$ and $\Psi(n_i) = B_i$. Let the update be performed in task $U_j^p \in \mathcal{C}_u$, $1 \leq p \leq |\mathcal{C}_u|$. Let this task be given the task number $t_j^p$, and let the portion of $\Psi(n_i)$ used in the update be $\mathcal{R}_{j,r}^p$, where $1 \leq r \leq t_j^p$. Then, the edge $(n_i, n_j)$ is annotated with the task number $t_j^p$. Also, the data flow along this Type I edge will result in inter-processor communication equivalent to $\mathcal{R}_{j,r}^p \subseteq B_i$, only if $\Pi(B_j) \neq \Pi(B_i)$.

Recall that according to the data-flow model of computation, a task cannot be executed until all the portions of the updating blocks are available. The block $\Psi(n_i)$ is available for the edge $(n_i, n_j)$, which in turn implies that the task $U_j^p$ cannot begin execution, until all the tasks corresponding to the edges coming into $n_i$ have finished execution. Note that there could be multiple edges of Type I between any pair of nodes, with each edge being annotated with a different task number. An edge of Type II from $n_i$ to $n_j$ is used only when $\Psi(n_i) = \Psi(n_j)$. This type of edge by itself does not stand for a task, hence it is not annotated with a task number. It is used to impose the precedence constraint that all the tasks corresponding to the edges coming into $n_i$ must be done before any of the tasks corresponding to the edges coming into $n_j$ are started. There is no communication along an edge of Type II since $\Psi(n_i) = \Psi(n_j)$ and the owner computes rule would ensure that all the computations that update $\Psi(n_i)$ are done on the same processor.

Each node is also assigned a weight. The weight on node $n_i$ is the total number of computational operations performed in updating the elements of $\Psi(n_i)$ in the execution of the tasks corresponding to the edges coming into $n_i$. 
The algorithm $\text{Construct.BDAG}$ shown in Figure 3.3 describes how a BDAG is constructed. The input to the algorithm is the set of blocks, $\mathcal{B}$ and the set of updates, $\mathcal{U}$. The output is the BDAG, $G$, represented by the set of nodes $\mathcal{N}$ and the set of edges $\mathcal{E}$, and the mapping of nodes to blocks, $\Psi$.

We illustrate the working of this algorithm by means of an example. The example is defined as follows.

- The set of blocks, $\mathcal{B} = \{B_1, B_2, B_3, B_4\}$.

- Consider only the updates to block $B_3$. Given that $C_3 = \{C^i_3\}$, $i = 1, 2, \ldots, 6$.

- The following dependencies are given for the tasks in $C_3$. (1) $C^1_3 \prec C^i_1$ for some $i$, (2) $C^1_3 \prec C^j_2$ for some $j$, (3) $C^2_3 \prec C^k_3$, (4) $C^2_3 \prec C^l_4$ for some $k$, (5) $C^3_3 \prec C^m_4$, (6) $C^3_3 \prec C^o_5$, (7) $C^4_3 \prec C^p_4$, (8) $C^5_3 \prec C^q_4$ for some $p$, (9) $C^6_3 \prec C^r_5$ for some $q$. (10) $C^6_3 \prec C^s_5$ for some $r$.

Figure 3.2 shows the part of the BDAG for block $B_3$. The construction of this is explained as we trace through the algorithm $\text{Construct.BDAG}$.

![Diagram](image)

Figure 3.2: Illustration of algorithm $\text{Construct.BDAG}$.

Each block, $B_u \in \mathcal{B}$ is assigned a “designated” node $d_u$. This node functions as the source node for all update tasks in which a part of $B_u$ is used to update other blocks.
The addition of these designated nodes to $N$ is shown in step 1 of the algorithm in Figure 3.3. In Figure 3.2, the designated nodes are shown in double circles.

The BDAG is constructed one block at a time. The main foreach loop of step 2 goes over each block, and adds to the graph all the nodes and edges associated with that block. Step 2.1 adds all the new nodes needed for a block, by taking up each task $C_u^i$ that updates block $B_u$. The first if statement in the loop of step 2.1 sets up a new node for each task that updates $B_u$ and depends on another task that also updates $B_u$. The second if statement sets up a new node for each task that updates $B_u$ and on which another task that also updates $B_u$ is dependent.

There are two additional checks. The first check avoids duplicate nodes. If a task both depends on another and has another task dependent on it, only one node is assigned for it. This check is made in the elseif part of the third if statement in step 2.1. The second check is for self-updates. This happens in a task when a portion of $B_u$ is used to update another portion of $B_u$, and the task depends on itself, shown in the check ($i = j$) in the third if statement in step 2.1. Note that if this task is $C_u^i$, then both the first and the second if conditions of step 2.1 are satisfied for this task, and two nodes are created, $n_u^i$ and $n_{u,1}^i$.

In the illustrative example, we consider the updates to block $B_3$. The tasks $C_3^1$ and $C_3^2$ contribute one node each, viz. $n_3^1$ and $n_3^2$ respectively. The task $C_3^3$ contributes two nodes, $n_{3,1}^3$ and $n_3^3$ since $C_3^3$ depends on itself.

Next, consider the set of tasks that update $B_u$ and possess the property that none of these tasks depends on any other task that also updates $B_u$. For this set of tasks, a single node $n_u$ is created. For each of these tasks, neither does the task depend on another task that updates $B_u$, nor does any other task that updates $B_u$ depend on it. This is shown in the condition check of the fourth if statement of step 2.1.

In the example, the tasks $C_3^4, C_3^5$, and $C_3^6$ are all assigned a single node, $n_3$.

Step 2.2 sets up the intra-block edges i.e. the edges of type II which do not represent tasks but simply enforce precedence constraints, and those edges of type I which indicate self-update tasks. Each edge of type I is appropriately labeled with the task number corresponding to the self-update task that is represented by the edge.
For the example, the type II edges \((n^4_3, n^3_3)\) and \((n^3_3, n^2_3)\), and the type I edge \((n^3_3,1, n^3_3)\) are created in step 2.2.

Step 2.3 sets up edges of type II from each of the 0-outdegree nodes (except the designated node) created for block \(B_u\) to the designated node \(d_u\). This ensures that the block \(B_u\) is used to update other blocks only when all the updates to \(B_u\) are done.

In the example, the type II edges \((n^2_3, d_3)\), \((n^3_3, d_3)\) and \((n_3, d_3)\) are created in step 2.3.

Step 2.4 sets up all the inter-block edges representing those tasks in which a portion of some block different from \(B_u\) is used to update a portion of \(B_u\). Each such edge originates from the designated node of the updating block.

In the example, the type I edges \((d_1, n^3_3)\), \((d_2, n^3_3)\), \((d_2, n^2_3)\), \((d_1, n_3)\), and the double-edge \((d_4, n_3)\) are created in step 2.4. Note, for instance, that the tasks \(C^4_3\), \(C^5_3\) and \(C^6_3\) can execute in any relative order, and the non-simultaneity of these tasks is guaranteed by the owner-computes rule applied to block \(B^3\). Also, for instance, each of these tasks can execute either before or after the task \(C^3_3\) with guarantee of non-simultaneity. Thus explicit edges to represent non-simultaneity are not required.

### 3.3 Conclusions

We have proposed a new representation for block-based unstructured computations, called Block Computation Representation, or BCR, for short. We view the computations as a collection of updates to data blocks, using the owner computes rule in parallelizing them. This rule is used in most parallelizing compilers and algorithms for scientific computations. It makes programming and maintaining data coherence easier, but it is not known whether this assumption unduly affects the performance of the parallelized code. Two other representations for computations and parallelism in the literature, the DAG and the MCTG, do not assume the owner computes rule. As a result, scheduling computations using a DAG or MCTG is less sensitive to the mapping of data to processors compared to scheduling using the BCR.
algorithm Construct BDAG
inputs: \( \mathcal{U} \) /* set of all updates */
\( \mathcal{B} \) /* set of blocks */
return: BDAG \( G = (\mathcal{N}, \mathcal{E}) \), mapping \( \Psi \)

1. foreach \( B_u \in \mathcal{B} \) do
   \( \mathcal{N} \leftarrow \mathcal{N} \cup \{d_u\} ; \ \Psi(d_u) \leftarrow B_u \)
end foreach

2. foreach \( B_u \in \mathcal{B} \) do
   
   2.1 foreach \( C_u^i \in \mathcal{C}_u \) do
      if (\( \exists j \ s.t. \ C_u^i \prec C_u^j \)) then \( \mathcal{N} \leftarrow \mathcal{N} \cup \{n_u^i\} ; \ \Psi(n_u^i) \leftarrow B_u \) endif
      if (\( \exists j \ s.t. \ C_u^i \prec C_u^j \)) then
         if (\( i = j \)) then \( \mathcal{N} \leftarrow \mathcal{N} \cup \{n_u^i\} ; \ \Psi(n_u^i) \leftarrow B_u \)
         elseif (\( n_u^i \notin \mathcal{N} \)) then \( \mathcal{N} \leftarrow \mathcal{N} \cup \{n_u^i\} ; \ \Psi(n_u^i) \leftarrow B_u \)
      endif
      endif
      if (\( \exists j \ s.t. \ C_u^i \prec C_u^j \) or \( C_u^j \prec C_u^i \)) then
         if (\( n_u \notin \mathcal{N} \)) then \( \mathcal{N} \leftarrow \mathcal{N} \cup \{n_u\} ; \ \Psi(n_u) \leftarrow B_u \) endif
      endif
   end foreach

   2.2 foreach \( C_u^i \in \mathcal{C}_u \)
      foreach \( j \ s.t. \ C_u^i \prec C_u^j \) do
         if (\( i \neq j \)) then \( \mathcal{E} \leftarrow \mathcal{E} \cup \{(n_u^i, n_u^j)\} \) of type II
      end foreach
   end foreach

   2.3 foreach \( n_u^i \ s.t. \ \text{outdegree of} \ n_u^i = 0 \) do
      \( \mathcal{E} \leftarrow \mathcal{E} \cup \{(n_u^i, d_u)\} \) of type II
   end foreach
   if (\( n_u \in \mathcal{N} \)) then \( \mathcal{E} \leftarrow \mathcal{E} \cup \{(n_u, d_u)\} \) of type II endif

   2.4 foreach \( C_u^i \in \mathcal{C}_u \)
      foreach \( j \ s.t. \ C_u^i \prec C_u^j \), \( u \neq q \) do
         if (\( n_u^i \in \mathcal{N} \)) then \( \mathcal{E} \leftarrow \mathcal{E} \cup \{(d_q, n_u^i)\} \) of type I
      else \( \mathcal{E} \leftarrow \mathcal{E} \cup \{(d_q, n_u)\} \) of type I
      endif
      label new edge with \( t_u^i \)
   end foreach
end foreach

end foreach

Figure 3.3: Construction of the BDAG.
Chapter 4

Analysis of column and block partitioning for sparse
Cholesky factorization

The parallel execution time of a numerical scientific algorithm on a message-passing multiprocessor is a measure of its goodness. Each step of the parallelization process, starting with the partitioning of the computation and data space of the algorithm, affects the performance of the final parallel code. These steps are very closely coupled, and it is very difficult to separately quantify the effect of each on the performance. Tied to these steps are several architectural and implementation issues that are hard to model mathematically, and whose effect on performance can only be gauged by rigorous experimentation. We discuss these issues in more detail in Chapter 7 where we present experimental results and evaluate performance.

It is difficult to quantify the amount of parallelism available in a problem instance for sparse Cholesky factorization, given the unstructured nature of the dependencies involved. In fact, even analyzing the computation and communication times is a difficult task since the distribution of computations and communication varies widely with the structure of the problem, and cannot be predicted a priori for general sparse matrices. For these reasons, theoretical results for sparse Cholesky factorization have been confined to a special class of structured problems.

In this chapter, we first give a qualitative description of the parallelism in sparse Cholesky factorization for column and 2-D block partitioning. We then describe the class of structured problems used in the analysis of column and block partitioning, and summarize the results in the literature for column-based partitioning methods. We then present our analysis for a block-based partitioning and allocation scheme and show that the communication in this block-based partitioning scheme is more scalable than the
column-based partitioning methods.

4.1 Models for parallel sparse Cholesky factorization

In sparse Cholesky factorization, parallelism is available at various levels. Depending on the granularity at which parallelism is extracted, parallel factorization methods can be grouped into four categories, each with its own model of parallel computation. In [43], the author has described three computational models based on task parallelism. We prefer to classify the models based on data parallelism. This classification is better suited for the methods proposed in this paper.

The finest grain parallelism in Cholesky factorization is extractable at the level of individual add-multiply and division operations associated with the update and scale computations. This is the operation-level model of parallelism. Under this model, a very high degree of parallelism is achievable, but because of its fine-grain nature, methods based on this model do not perform well on the existing and proposed MIMD architectures where memory latency costs are relatively high. Also, the associated bookkeeping costs tend to be high on general purpose systems. The parallel factorization method presented in [77] is based on this model.

The granularity of parallel operations is increased if parallelism is extracted at the level of a non-zero element in \( L \). Refer to this as the element-level model of parallelism. In this model, all the computations corresponding to any element are grouped into an atomic unit, and the parallelism is viewed at this level of atomicity. The parallelism in this model is still too fine-grained to be efficiently exploited on the current MIMD architectures.

The level of granularity can be further increased by grouping together elements of the matrix into larger units. A single column in a matrix is the most common example of such grouping of elements. In the update and scale operations, a column is considered as an atomic entity. Under such a model, the data structures and the operations are all column or row oriented and parallelism is extracted at the level of column-update-column or column-scale operations. Hence this is the column-based model of parallelism.
In this model, the data distribution schemes are also column based; i.e., processors are assigned one or more columns of $L$. We shall refer to such data distribution schemes as \textit{column-based partitioning} methods. Most commonly used and studied parallel sparse factorization methods are based on this model \cite{11, 23, 43, 82, 24, 4}. This is so because of its relative simplicity in analysis and the close resemblance of the data structures with the algebraic representation.

Instead of grouping the elements of $L$ only along its columns (or rows), one can group these into two dimensional blocks; i.e., along both rows and columns and not necessarily grouping together all the elements of a row or a column. The update and scale operations in Cholesky factorization can then be performed using these blocks as single entities. We refer to this as the \textit{2-D block model} of parallelism. Under this model, the matrix data is partitioned in the form of 2-D blocks. The blocks may contain only the non-zeros or may contain both the zeros and non-zeros. Once the matrix is partitioned, the resulting 2-D blocks are then distributed among the processors. This type of partitioning is the \textit{2-D block partitioning} method. Because of the greater flexibility offered in choosing the extents of each block, methods based on the 2-D block model have greater potentials for exploiting parallelism and locality than it is possible with the column-based methods. The block-based partitioning we propose is based on this 2-D block model.

\section{Two dimensional regular grids}

Consider the sparse matrix representing a two dimensional regular grid of size $n \times n$, whose grid connectivity is defined by a five point stencil. If this matrix is reordered using the nested dissection reordering method \cite{29}, the ordering is optimal with respect to both number of arithmetic operations and fill-in for sequential sparse Cholesky factorization of the corresponding sparse matrix. The number of multiplicative operations is $9.87n^3$ and the number of non-zeros in the Cholesky factor is $7.75n^2 \log n$ \cite{29}. In the rest of this chapter we refer to this as the regular grid problem, and the structure of the corresponding lower triangular Cholesky factor as the grid matrix.
Figure 4.1(a) shows a $7 \times 7$ grid reordered using nested dissection. The ordering

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{nested_dissection.png}
\caption{Nested dissection ordering on a $7 \times 7$ grid.}
\end{figure}

algorithm proceeds by recursively subdividing the grid, and in the process, numbering the grid points from 1 through $n^2$. At any given level of the recursive process, let the largest number available for numbering be $k$ and let the size of the grid under consideration be $m \times m$. The algorithm identifies three separators of this grid - one vertical separator and two horizontal separators. The grid points in the vertical separator are first numbered from $k - m + 1$ to $k$. Then the grid points in the left horizontal separator are numbered from $k - 3m/2 + 1$ to $k - m$, followed by the grid points in the right horizontal separator, which are numbered from $k - 2m + 1$ to $k - 3m/2$. The vertical separator and horizontal separators are collectively referred to as the '+'-separator. The '+'-separators divide the grid into four sub-grids, which are then recursively numbered. The recursion terminates when a sub-grid has only one grid point in it.

The structure of the sparse Cholesky factor for this reordering for the first seven separators is shown in Figure 4.1(b). The size of this matrix is $n^2 \times n^2$. Notice that the rightmost dense triangle corresponds to the first vertical separator, and is of size $n \times n$. In general, corresponding to each separator in the grid, there is a clique of columns in the grid matrix, and corresponding to each grid point in this separator,
there is a column in this clique. Details of the data dependencies among the separators are described in [52].

4.2.1 Column assignment schemes

The subtree-to-subcube scheme of assigning columns to processors was proposed by George, Liu and Ng [26] for hypercube network based distributed memory machines. The scheme is based on recursively mapping a subtree of the elimination tree to a subcube of the hypercube. Given \( P \) processors and the \( n \times n \) grid problem, the grid points in the first level '+'-separator are assigned to the \( P \) processors in a wrap-around manner. This '+'-separator creates four subgrids, each of which is assigned to a disjoint set of \( \frac{1}{4}P \) processors. Specifically, if the hypercube is of dimension \( d \), then the four disjoint sets of processors are the four \( d - 2 \) dimension subcubes of the hypercube. This process is recursively repeated until the number of subgrids at some level is equal to or more than \( P \).

In the bottom-up scheme described in [24], columns are assigned to processors in wrap-around fashion beginning at the leaf level of the elimination tree and going towards the root.

4.3 Results for column-based partitioning

We first list the known theoretical results for communication during the Cholesky factorization of the sparse matrix for the regular grid problem.

1. Assuming load balancing, and a distributed fan-out algorithm, George, Liu and Ng [26] showed that the communication volume for the bottom-up assignment is \( O(Pn^2 \log n) \).

2. Assuming load balancing, and a distributed fan-out algorithm, the communication volume for the subtree-to-subcube assignment is \( O(Pn^2) \) in [26]. This result was improved by Gao and Parlett [22], who showed that the communication volume is \( O(n^2) \) per processor. The traffic is thus balanced among the processors.
3. George et al. [26] also showed that assuming load balance, and a distributed fan-out algorithm, the communication volume is at least $O(Pn^2)$, irrespective of the ordering scheme and the scheme for assigning columns to processors. In other words, this is asymptotically the minimum volume of communication for the fan-out algorithm, given the assumption of load balance.

4. Hulbert and Zmijewski [37] showed that for their column-based algorithm with the subtree-to-subcube assignment, the communication volume is $O(Pn^2)$, and the total number of messages is $O(Pn \log P)$. They also show that the fan-out algorithm in [26] communicates $O(Pn \log n)$ messages.

5. Pothen and Sun [57] showed that their distributed multifrontal algorithm using clique trees, and a subtree-to-subcube mapping of the clique tree to processors, requires a communication volume of $O(Pn^2)$.

We have independently derived the total number of messages for the column-oriented fan-out algorithm for the grid matrix, using the subtree-to-subcube assignment scheme. This result follows.

**Theorem 4.3.1:** The total number of messages sent during the factorization is $O(Pn \log n)$.

**Proof:**

We denote the ' + ' -separator at the topmost level as the level-1 separator, the four ' + ' -separators at the next level as the level-2 separators, and so on. The outline of the proof is as follows. We count individually the number of messages coming into the $p$ processors assigned to the level-1 ' + ' separator, the number of messages coming into each of the disjoint groups of $p/4$ processors assigned to the four level-2 separators, and so on. The sum of these counts gives us the result.

Consider a generic subgrid of size $i \times i$, and let the ' + ' -separator of this grid be at level $l$. Assume that $p$ processors are assigned to work on this separator. Figure 4.2 shows such a grid and its separator. Let $m(i,p)$ denote the number of messages coming
into these $p$ processors. We can write

$$m(i, p) = s(i, p) + 4r(i/2, p),$$

(4.1)

where $s(i, p)$ denotes the number of messages exchanged among the processors assigned to the horizontal and vertical sub-separators of the $'+'$-separator, and $r(i/2, p)$ is the number of messages that come into the processors assigned to the $'+'$-separator from each of the four $i/2 \times i/2$ sub-grids.

Consider any one of these $i/2 \times i/2$ sub-grids. It can be broken up to four sub-sub-grids again, as shown in Figure 4.2. Of these sub-sub-grids, the lower left does not communicate with the level-$l$ separator. The other three can be classified into three types depending on how their boundaries relate to the level-$l$ separator. The $h$ sub-sub-grid has part of a level-$l$ horizontal sub-separator in its boundary, the $hv$ sub-sub-grid has part of a level-$l$ horizontal sub-separator and part of the level-$l$ vertical sub-separator in its boundary, and the $v$ sub-sub-grid has part of the level-$l$ vertical sub-separator in its boundary.

Let $h(j, p)$ denote the number of messages that a $h$-type $j \times j$ sub-grid sends to the $p$ processors that are assigned to the level-$l$ horizontal sub-separator on its boundary. Similarly, let $v(j, p)$ and $hv(j, p)$ denote the number of messages sent to the level-$l$ horizontal and vertical sub-separators on their boundaries.
We have the following recurrence relations:

\[ h(j, p) \leq 2jp + 2h(j/2, p), \]
\[ v(j, p) \leq (3j/2)p + 2v(j/2, p), \]
\[ hv(j, p) \leq 2jp + h(j/2, p) + v(j/2, p) + hv(j/2, p), \]

with \( h(1, p) = v(1, p) = hv(1, p) = 1. \) Solving these recurrence relations we get:

\[ h(j, p) \leq 2 pj \log j, \]
\[ v(j, p) \leq (3p/2) j \log j, \]
\[ hv(j, p) \leq 4pj \log j. \]

Note that \( r(i/2, p) = hv(i/2, p) \leq 2pi \log i. \) Also, \( s(i, p) \leq p(i/2) + p(i/2) + ip = 2pi. \)

Hence, from equation 4.1, we obtain

\[ m(i, p) = s(i, p) + 4r(i/2, p) \leq 2pi + 8pi \log i \leq 10pi \log i. \]

Let \( M(n, P) \) denote the total number of messages into all the processors which work on all the separators at all the levels. We have another recurrence relation

\[ M(n, P) = m(n, P) + 4M(n/2, P/4), \]

solving for which gives the us required result,

\[ M(n, P) \leq 20Pn \log n = O(Pn \log n) \]

\[ \blacksquare \]

4.4 Results for block-based partitioning

Naik [54, 53] proposed a block partitioning scheme and proved that it incurred a data traffic of \( O(n^2 \sqrt{P}) \) in factoring the sparse grid matrix described earlier. He also showed that a lower bound on the data traffic is \( \Omega(n^2 \sqrt{P}) \) under the condition of uniform load distribution, independent of the particular implementation or ordering scheme used. Thus, the data traffic is optimal for this block scheme.

Since our results for block partitioning pertain to the above block partitioning scheme, we briefly describe this partitioning scheme here in order to make this section self-contained. We then present our analysis of computation time, communication time and idle time for this partitioning scheme. We also derive a bound on the total number of messages for this partitioning scheme.
4.4.1 Sparse BLOCC scheme

Naik, in [54], proposes a scheme for dense Cholesky factorization called the block oriented column Cholesky factorization scheme, or BLOCC scheme for short. The scheme is as follows. Let $P$ processors be assigned to factorize a dense $m \times m$ symmetric positive definite matrix. Without loss of generality assume that $P = (r^2 + r)/2$ where $r$ is an integer. The lower triangular part of the matrix is divided into $P$ partitions by taking $r$ vertical and $r$ horizontal sections each of size $s$, where $s = m/r$. All except $r$ of the resulting $P$ partitions are square blocks of size $s \times s$. The remaining $r$ partitions which lie on the diagonal of the lower triangular matrix are $s \times s$ triangular blocks. Each of the partitions is assigned a single processor. The factorization then proceeds in block-wise fashion, using the column-oriented Cholesky factorization method as the underlying numerical algorithm. The scheme proposed is for shared memory multiprocessors, in which as soon as an element of the factor is computed in any block, it is written into the shared memory for access by other processors.

The dense BLOCC scheme is extended by the author, to the Cholesky factorization of sparse grid matrix. The key idea is the following. If $\Gamma$ is any $m$-vertex sub-separator, then, in the band of $m$ columns corresponding to the vertices of the separator in the Cholesky factor, (a) there is a dense $m \times m$ triangular diagonal block in the Cholesky factor, and (b) there are at most four off-diagonal rectangular blocks with non-zero elements. Each of the off-diagonal blocks is of size at most $(c_1 \cdot m + c_2) \times m$, where $c_1 \leq 2$ and $c_2 \leq 3$ are positive integer constants. Any non-zero in these columns of $\Gamma$ is either in the diagonal triangular block or in one of the four off-diagonal rectangular blocks.

Now consider the factorization of such an $m$-vertex sub-separator, for which $p$ processors have been assigned. Using all $p$ processors, the factorization of the $m \times m$ triangular diagonal block is first computed. For this part, the dense BLOCC scheme is used. Then the processors are used to compute the elements corresponding to the four off-diagonal rectangular blocks. For the purpose of factoring, these blocks are treated as if they were adjacent, and the resultant rectangular block is partitioned into $p$ sub
blocks each of size \((c \cdot m/\sqrt{P}) \times (m/\sqrt{P})\) where \(c \leq 6\) for a horizontal sub-separator and \(c \leq 4\) for a vertical sub-separator. Each partition is assigned a separate processor. For details of the factorization of the entire matrix, including the allocation of processors to separators, see [54].

4.4.2 Sparse BLOCC scheme for message-passing multiprocessors

We extend the sparse BLOCC scheme to message-passing multiprocessors. In order to understand the performance benefits and the limitations of the block scheme, we have developed a model to represent the parallel computations in factoring the sparse matrix. We analyze this model to get a bound on the performance of the block partitioning method. We then derive a bound on the total number of messages for the BLOCC scheme.

Performance model and time analysis

Without loss of generality, let \(P\) be equal to \(2^{2\ell}\), for some integer \(\ell\). Assume there are \(2\ell\) levels of separators in all. The levels are numbered from 0 to \(2\ell - 1\) as follows. The vertical separators are numbered \(2i\), \(0 \leq i < \ell\). The vertical separator of size \(n\) at the topmost level is given the level number 0, followed by level number 2 for the vertical separators of size \(n/2\) and so on. Analogously, the horizontal separators are numbered \(2i - 1\), \(1 \leq i \leq \ell\). The two horizontal separators of size \(n/2\) each are given the level number 1, the eight horizontal separators of size \(n/4\) each are given the level number 3, and so on. Refering to the separators in Figure 4.1(a), \(V_1\) is at level 0, \(H_1\) and \(H_2\) are at level 1, \(V_i\), \(i = 2 \ldots 5\) are at level 2 and the eight single-point horizontal separators are at level 3.

All \(P\) processors are assigned to compute the level-0 vertical separator. Of these, two disjoint sets of \(P/2\) processors each are assigned to compute the two level-1 horizontal separators. Next, four disjoint sets of \(P/4\) processors each are assigned to the four sub-grids that are formed by these vertical and horizontal separators. This process of allocation is repeated recursively for each of the sub-grids. The computations are modeled as follows. As stated earlier, all times are quantified in terms of number of
floating point operations.

1. The computations proceed level by level starting with the highest numbered level. At each level, all the separators at that level are computed in parallel. The computations are synchronized at the end of the computations on each level.

2. Each separator at any level is assigned a fixed number of processors, which compute the separator in parallel. Of these processors, there is one processor which is the last to finish the computations. For all separators at a given level, these “last” processors are all assigned an equal amount of work, i.e., the load is evenly balanced among these processors.

3. The parallel computation time at each level is the maximum number of operations performed by any processor at that level, which in turn is the number of operations performed by the processor which finishes last in any separator. A floating point multiply followed by an add constitutes an operation. This is the elementary operation in the Cholesky factorization. The total computation time, $T_{\text{comp}}$, is the sum of the parallel computation time over all levels.

4. An element of a block may be used in the computation of elements in other blocks only when all the elements of the block have been factored. The entire block is then sent to all processors which need any element in the block. When a block has to be sent to several processors, it will arrive at the same time at all the destination processors. The time taken from the moment a block of size $d$ is sent to any dependent processor to the moment it arrives at that dependent processor is given by $\alpha_g + \beta_g \cdot d$. Here, $\alpha_g$ is the latency cost, relative to the computation cost, in initiating a message and $\beta_g$ is the transmission cost per element transmitted, relative to the computation cost. In other words, if the absolute latency cost is $\alpha$, the absolute transmission cost is $\beta$ and the absolute computation cost is $w$, then, $\alpha_g = \alpha/w$ and $\beta_g = \beta/w$.

5. The idle time on a processor comprises of the delay time and the communication time.
6. The delay time is the time spent in waiting for predecessor blocks to finish their computations, i.e., the delay is due to precedence relationships. The delay at any level is the delay experienced by the processor which finishes last. The total delay time, $T_{delay}$, is the sum of the delay time over all levels.

7. The communication time at any level is the time contributed by the communication to the idle time of the processor which finishes last in that level. In other words, this is the communication cost in the critical path. All other communication is considered to overlap with this, and is not taken into account in the communication time. Thus the communication time only considers the contribution of these "non-overlapping" messages along the critical path. The total communication time, $T_{comm}$, is the sum of the communication time over all levels.

The total parallel execution time is given by $T_{tot} = T_{comp} + T_{delay} + T_{comm}$. We now present the analysis for the derivation of each of these parallel time components.

The analysis gives an upper bound on the actual parallel time. It examines a generic separator with $m$ elements and computes the computation time, delay due to precedence and delay due to communication for the processor which finishes last in that separator. Figure 4.3(a) shows the diagonal triangle and Figure 4.3(b) shows the off-diagonal rectangle of the clique corresponding to a separator with $m$ grid points.

Recall from the discussion in Section 4.4.1 that the diagonal triangular block is of size $m \times m$ and the off-diagonal blocks are treated as a single contiguous block. As shown in Figure 4.3, this off-diagonal block is of height at most $4m$ for a vertical separator and at most $6m$ for a horizontal separator.

A sub-grid at level $i$, $i = 2j$, is of size $n/2^i \times n/2^i$. A separator at level $i$ is assigned $P/2^i$ processors. Since $P = 2^{2l}$, it follows that the horizontal separators at level $2l - 1$ are each assigned two processors, and each of the $n/2^i \times n/2^i$ sub-grids at the level $2l$ is assigned one processor.

Consider a separator at level $i$, to which the number of processors assigned is $P_i$. The separator is partitioned as follows. The triangular diagonal block is divided into $P_i$ partitions, with $k^2/2 - k/2$ square sub-blocks and $k$ diagonal sub-blocks, each of
size $m/k$, where $P_i = k^2/2 + k/2$. Each of these $P_i$ partitions is assigned a unique processor. For the purpose of the analysis we will approximate the size of each partition by $(m/\sqrt{2P_i}) \times (m/\sqrt{2P_i})$. The rectangular block is divided into $P_i$ sub-blocks. Again, each sub-block is assigned a unique processor. Each sub-block is of size $(4m/\sqrt{P_i}) \times (m/\sqrt{P_i})$ for a vertical separator and $(6m/\sqrt{P_i}) \times (m/\sqrt{P_i})$ for a horizontal separator. In any separator, the processors first perform the computations for the triangular block, followed by the computations for the rectangular block.

In the rest of the analysis, we will use the term “separator” to mean both the separator in the grid and the corresponding clique in the grid matrix. The use will be clear from the context. We now prove three theorems: one for the computation time, one for the delay due to precedence relationships, and one for the delay due to the communication. In proving each of these theorems, we compute the required quantity for a “worst-case” processor assigned to a generic vertical separator and a generic horizontal separator at some level of the nested dissection. The result of the theorem is then obtained by summing this quantity over the horizontal and vertical separators at all levels, using the fact that a processor is assigned to exactly one separator at any level.

**Computation time**

Figure 4.3: Clique corresponding to separator of size $m$. 

(a) 

(b)
We derive the worst-case computation time for any processor, in computing the Cholesky factor using the block partitioning scheme. The “worst-case” processor for a generic separator of size $m$ is the one which performs the most number of operations. In turn, this is the processor which works on the rightmost triangular block in the $m \times m$ dense triangle and is one of the processors which works on the last column of blocks in the off-diagonal rectangular block. Even though the worst-case processor may not be the same for the triangle and the rectangle, we assume they are the same for the purpose of the worst-case analysis. In the following lemmas, we quantify the computation time for this processor for a triangular block and a rectangular block, respectively, in the generic separator. Assume that $P_i$ processors are assigned to the separator, which is at level $i$.

**Lemma 4.4.1:** The worst-case number of operations for a block in the triangle is $5m^3/(2P_i)$ for a vertical separator and $7m^3/(2P_i)$ for a horizontal separator.

**Proof:**

For the triangular block, the worst-case number of operations is performed by the processor which works on the triangular sub-block on the lower right corner, shown by the shaded triangle in Figure 4.3(a). The computational work for this block can be divided into internal work, i.e. contributions by the other blocks in the triangle and external work, i.e. contributions by blocks which are external to the triangle, to its left in the matrix.

Considering only the internal work, the number of operations required to compute an element in this block is at most $m$. Thus, the computational work in this block is $m \times m^2/(2P_i) = m^3/(2P_i)$.

For the external work computation, consider the sub-grid shown in Figure 4.4. We compute the external work for the generic vertical separator $V$ in (a) and the horizontal separator $H$ in (b), each of which is of size $m$.

Let $g$ be a grid point in the separator. Let $S$ be a separator in the grid which is ordered ahead of $V$ or $H$, with the property that every grid point in $S$ is connected to $g$ in the elimination graph, just prior to the elimination of $S$. Please refer to [28]
for a discussion of the elimination graph model. Let \( g_c \) be the column in the matrix corresponding to the grid point \( g \). Each such connection from a grid point \( g' \) in \( S \) to \( g \) will, at the time that \( g' \) is eliminated, induce an update operation on each non-zero element in the portion of \( g_c \) which is enclosed in the triangular block. This is because \( g' \) is connected to every grid point in the separator \( V \) or \( H \). Hence the total number of such operations on a non-zero in \( g_c \) is equal to the total number of such connections from grid points of separators which are ordered ahead of \( V \) or \( H \).

For the vertical separator \( V \), \( g \) will be connected to at most

\[
2 \cdot m/2 + 2 \cdot m/2 + 2 \cdot m/4 + 2 \cdot m/4 + \cdots = 4m
\]

grid points. Some of the separators responsible for these connections are labeled as \( A, A', B, B', C, C', D, D' \) in Figure 4.4(a). Thus, each non-zero element in a sub-block in the diagonal block for \( V \) will require at most \( 4m \) operations. In particular, the last block will require, in all, at most \( 4m \times m^2/(2P) = 2m^3/P \), operations.

For the horizontal separator \( H \), \( g \) will be connected to at most

\[
2 \cdot m + 2 \cdot m/2 + 2 \cdot m/2 + 2 \cdot m/4 + 2 \cdot m/4 + \cdots = 2m + 4m = 6m
\]

grid points. Some of the separators responsible for these connections are labeled as \( A, A', B, B', C, C', D, D' \) in Figure 4.4(b). Thus, each non-zero element in a sub-block
in the diagonal block for $H$ will require at most $6m$ operations. In particular, the last block will require, in all, at most $6m \times m^2 / (2P_i) = 3m^3 / P_i$ operations.

Adding the internal and external work for the worst-case block gives the result of the lemma. 

**Lemma 4.4.2:** The worst-case number of operations for a block in the rectangle is $12m^3 / P_i$ for a vertical separator and $24m^3 / P_i$ for a horizontal separator.

**Proof:**

For the rectangular block, the worst-case number of operations is performed by a processor which works a block in the last column of blocks, shown by the shaded rectangles in Figure 4.3(b). Again, the computational work for this block can be divided into internal work, i.e. contributions by the other sub-blocks in the rectangle and external work, i.e. contributions by blocks which are external to the rectangle, to its left in the matrix.

Considering only the internal work, the number of operations needed to compute any element in the last block is at most $m$. Thus, the computational work for this block is $m \times 4m^2 / P_i = 4m^3 / P_i$ for a vertical separator and $m \times 6m^2 / P_i = 6m^3 / P_i$ for a horizontal separator.

For the external work, consider the sub-grid shown in Figure 4.5.

We compute the external work for the generic vertical separator $V$ in (a) and the horizontal separator $H$ in (b), each of which is of size $m$.

Let $g$ be a grid point in the separator. Also, let $h$ be a grid point on one of the bordering edges; these border edges are separators which are ordered after $V$ or $H$. Let $S$ be a separator in the grid which is ordered ahead of $V$ or $H$. The following property holds for the chosen $S$ and $h$: every grid point in $S$ is connected to $g$ in the elimination graph, and every grid point in $S$ is connected to $h$ in the elimination graph, just prior to the elimination of $S$. When a grid point in $S$ is eliminated, an edge is created between $h$ and $g$ in the elimination graph. Let $g_c$ be the column in the matrix corresponding to the grid point $g$. Each such connection from a grid point $g'$ in $S$ to $g$ will induce an update operation on the non-zero element in row $h$ of $g_c$, $(h$ is in the
off-diagonal rectangular block of $V$ or $H$) at the time that $g'$ is eliminated. Hence the total number of such operations needed for to compute the non-zero in row $h$ in $g_c$ is equal to the total number of such connections from grid points of separators which are ordered ahead of $V$ or $H$.

For the vertical separator $V$, $g$ will be connected to at most

$$m/2 + m/2 + m/4 + m/4 + \cdots = 2m$$

grid points. Some of the separators responsible for these connections are labeled as $A$, $B$, $C$, $D$ in Figure 4.5(a). Thus, a non-zero element in row $h$ in the off-diagonal block for $V$ will require at most $2m$ operations. In particular, the last block will require, in all, at most $2m \times 4m^2/P_i = 8m^3/P_i$ operations.

For the horizontal separator $H$, $g$ will be connected to at most

$$m + 2 \times m/2 + 2 \times m/4 + \cdots = m + 2 \times m = 3m$$

grid points. Some of the separators responsible for these connections are labeled as $A$, $B$, $C$, $D$ in Figure 4.5(b). Thus, a non-zero element in row $h$ in the off-diagonal block for $H$ will require at most $3m$ operations. In particular, the last block will require, in all, at most $3m \times 6m^2/P_i = 18m^3/P_i$ operations.
Adding the internal and external work for the worst-case block gives the result of the lemma.

We now use these lemmas to prove the computation time theorem.

**Theorem 4.4.1:** In computing the Cholesky factor using the block partitioning scheme, the computation time for any processor, $T_{comp}$, is at most

$$(113n^3/4P)(1 - 1/\sqrt{P}) + (371/12)n^3/P^{3/2}.$$ 

**Proof:** Adding the results of Lemma 4.4.1 and Lemma 4.4.2, the worst-case work for a block is $29m^3/(2P_1)$ for a vertical separator and $55m^3/(2P_1)$ for a horizontal separator.

To obtain the total worst-case time for a processor for all vertical separators to which it is assigned, we use $i = 2j$, $0 \leq j < l$, $m = n/2^j$ and $P_i = P/2^{2j}$, and sum over all levels $i$ to get:

$$\frac{29}{2P} \sum_{j=0}^{i-1} n^3/2^j. \quad (4.2)$$

Similarly, to obtain the total worst-case time for a processor for all horizontal separators to which it is assigned, we use $i = 2j - 1$, $1 \leq j \leq l$, $m = n/2^j$ and $P_i = P/2^{2j-1}$, and sum over all levels $i$ to get:

$$\frac{55}{4P} \sum_{j=1}^{l} n^3/2^j. \quad (4.3)$$

Finally, recall that each of the $n/2^l \times n/2^l$ sub-grids at the level $2l$ is assigned one processor. The time taken to factor an $n \times n$ sub-grid at level $i$ which is bordered on all four sides is $371n^3/12 + O(n^2 \log_2 n)$, see [28]. Ignoring the lower order term and replacing $n$ by $n/2^l$, a processor takes time $(371n^3)/(12 * 2^{2l})$ to compute the lowest level sub-grid assigned to it.

To obtain the result of the theorem, we approximate the sum for the vertical separators in equation (4.2) by letting $j$ run from $1$ to $l$ instead of $0$ to $l-1$, use the fact that $P = 2^{2l}$, and add the results of equations (4.2) and (4.3), and the uniprocessor computational work, to obtain:
\[ T_{\text{comp}} = n^3 \cdot \left( \frac{29}{2P} + \frac{55}{4P} \right) \cdot \sum_{j=1}^d 1/2^j + \frac{371}{12} n^3 / P^{3/2} \]
\[ = (113n^3/(4P)) \cdot (1 - 1/2^i) + \frac{371}{12} n^3 / P^{3/2} \]
\[ = (113n^3/(4P)) \cdot (1 - 1/\sqrt{P}) + \frac{371}{12} n^3 / P^{3/2}. \]

\[ \blacksquare \]

**Delay due to precedence relations**

We derive the delay due to precedence relations seen by a processor on the critical path. Again, the "worst-case" processor for a generic separator of size \(m\) is the one which experiences the most delay. In turn, this is the processor which works on the rightmost triangular block in the \(m \times m\) dense triangle and is one of the processors which works on the last column of blocks in the off-diagonal rectangular block. Even though the worst-case processor may not be the same for the triangle and the rectangle, we assume they are the same for the purpose of the worst-case analysis. In the following lemmas, we quantify the delay time for this processor for a triangular block and a rectangular block, respectively, in the generic separator. Assume that \(P_i\) processors are assigned to the separator, which is at level \(i\).

**Lemma 4.4.3:** The worst-case delay seen by a processor assigned to a block in the triangle is \(m^3/(4P_i)\).

**Proof:**

The result is the same for a vertical or a horizontal separator. This is so because the delay computation is based only on the blocks which are internal to a separator, and for the triangle, the partitioning geometry is the same for a vertical or a horizontal separator. The processor which experiences the worst delay is the one which works on the triangular partition in the lower right corner. This is the block which incurs the greatest number of operations, shown in the proof of Theorem 4.4.1. The delay for this block is caused by each of the blocks to its left in the lowermost block row as follows. In the last row of blocks, the leftmost block is computed first. During this computation, all other processors in that block row see a delay. After the computation,
this block is broadcast to each block to its right in the last row block. It is used to then simultaneously update the recipient blocks. Then, the second block from left is computed during which time all the other processors again see a delay. So, in all, the last block is delayed during the internal computation of each block to its left in the triangle. There are $\sqrt{2P_i}$ such blocks. The number of operations in the internal computation of any such block is:

$$(m/\sqrt{2P_i}) \times (1 + 2 + 3 + \cdots + m/\sqrt{2P_i}) \approx (m/\sqrt{2P_i}) \times (m^2/(4P_i)),$$

and the delay is thus

$$(m/\sqrt{2P_i}) \times (m^2/(4P_i)) \times \sqrt{2P_i} = m^3/(4P_i).$$

\[ \square \]

**Lemma 4.4.4:** The worst-case delay seen by a processor assigned to a block in the rectangle is $2m^3/P_i$ for a vertical separator, and $3m^3/P_i$ for a horizontal separator.

**Proof:**

The delay seen by the last block is computed by a similar argument to the one for Lemma 4.4.3. The result follows in a straightforward manner. For a vertical separator, the delay is

$$(4m/\sqrt{P_i}) \times (m^2/(2P_i)) \times \sqrt{P_i} = 2m^3/P_i,$$

and for a horizontal separator, the delay is

$$(6m/\sqrt{P_i}) \times (m^2/(2P_i)) \times \sqrt{P_i} = 3m^3/P_i.$$

\[ \square \]

We now use the results of these lemmas to prove the theorem on delay due to precedence relations.

**Theorem 4.4.2:** In computing the Cholesky factor using the block partitioning scheme, the delay experienced by any of the processors due the precedence relations, $T_{delay}$, is at most $(31n^3/8P)(1 - 1/\sqrt{P})$. 


Proof:

Adding the results of Lemma 4.4.3 and Lemma 4.4.4, the worst-case delay seen by a processor is $9m^3/(4P)$ for a vertical separator and $13m^3/(4P)$ for a horizontal separator.

To obtain the total worst-case delay seen by a processor for all vertical separators to which it is assigned, we use $i = 2j, 0 \leq j < l$, $m = n/2^j$ and $P_i = P/2^{2j}$, and sum up $9m^3/(4P_i)$ over all levels $i$ to get:

$$ (9/(4P)) \cdot \sum_{j=0}^{l-1} (n^3/2^j). \quad (4.4) $$

To obtain the total worst-case delay seen by a processor for all horizontal separators to which it is assigned, we use $i, i = 2j - 1, 1 \leq j \leq l$, $m = n/2^j$ and $P_i = P/2^{2j-1}$, and sum up $13m^3/(4P_i)$ over all levels $i$ to get:

$$ (13/(8P)) \cdot \sum_{j=1}^{l} (n^3/2^j). \quad (4.5) $$

Finally, we approximate the sum for the vertical separators in equation (4.4) by letting $j$ run from 1 to $l$ instead of 0 to $l - 1$, use the fact that $P = 2^l$, and add the results of equations (4.4) and (4.5) to obtain:

$$ T_{\text{delay}} = n^3 \cdot (9/(4P) + 13/(8P)) \cdot \sum_{j=1}^{l} (1/2^j) $$

$$ = (31n^3/(8P)) \cdot (1 - 1/2^l) $$

$$ = (31n^3/(8P)) \cdot (1 - 1/\sqrt{P}). $$

Delay due to communication

We derive the delay due to communication seen by a processor on the critical path. All other communication is considered to overlap with the communication on the critical path, and is not taken into account in the delay computation. The “worst-case” processor for a generic separator of size $m$ is the one which experiences the most delay. In turn, this is the processor which works on the rightmost triangular block in the
$m \times m$ dense triangle and is one of the processors which works on the last column of blocks in the off-diagonal rectangular block. Even though the worst-case processor may not be the same for the triangle and the rectangle, we assume they are the same for the purpose of the worst-case analysis. In the following lemmas, we quantify the delay time for this processor for a triangular block and a rectangular block, respectively, in the generic separator. Assume that $P_i$ processors are assigned to the separator, which is at level $i$.

**Lemma 4.4.5:** The worst-case delay seen by a processor assigned to a block in the triangle is $(\alpha_g + \beta_g \cdot m^2/(2P_i)) \cdot \sqrt{2P_i}$.

**Proof:** The result is the same for a vertical or a horizontal separator. This is so because the delay is determined only by the blocks which are internal to a separator, and for the triangle, the partitioning geometry is the same for a vertical or a horizontal separator. The processor which experiences the worst delay is the one which works on the triangular partition in the lower right corner. This is the block which incurs the greatest number of operations, shown in the proof of Theorem 4.4.1. The delay due to communication seen by this block is caused by each of the blocks to its left in the lowermost block row as follows. In the last block row, the leftmost block is first computed. After the computation, this block is broadcast to each block to its right in the last row block. During this time, all the processors to the right are idle. This is repeated in sequence for all the blocks to the left of the last block, and we take the number of such blocks to be $\sqrt{2P_i}$. The communication time for each block's message is $\alpha_g + \beta_g \cdot m^2/(2P_i)$, since there are $\sqrt{2P_i}$ messages, the delay time for the last block is $(\alpha_g + \beta_g \cdot m^2/(2P_i)) \cdot \sqrt{2P_i}$.

**Lemma 4.4.6:** The worst-case delay seen by a processor assigned to a block in the rectangle is $(\alpha_g + \beta_g \cdot 4m^2/P_i) \cdot \sqrt{P_i}$ for a vertical separator, and $(\alpha_g + \beta_g \cdot 6m^2/P_i) \cdot \sqrt{P_i}$ for a horizontal separator.

**Proof:**
The delay seen by the last block, i.e. some block in the last column of blocks in the rectangle, is computed by a similar argument to the one for Lemma 4.4.4. The result follows in a straightforward manner.

\[ \alpha_g \cdot (3 + 2\sqrt{2})\sqrt{P} + \beta_g \cdot (9 + 7\sqrt{2})n^2/(2\sqrt{P})(1 - 1/\sqrt{P}). \]

**Proof:**

Adding the results of Lemma 4.4.5 and Lemma 4.4.6, the worst-case delay seen by a processor is \((\alpha_g + \beta_g \cdot m^2/(2P)) \cdot \sqrt{2P_i} + (\alpha_g + \beta_g \cdot 4m^2/P_i) \cdot \sqrt{P_i} \) for a vertical separator. Using \(i = 2j, 0 \leq j < l\), \(m = n/2^j\) and \(P_i = P/2^{2j}\), this gives

\[ (\alpha_g + \beta_g \cdot n^2/(2P)) \cdot (\sqrt{2P}/2^j) + (\alpha_g + \beta_g \cdot 4n^2/P) \cdot (\sqrt{P}/2^j) \]

for the delay. To obtain the worst-case delay for a processor over all levels of vertical separators, we sum the above over all \(j\):

\[ \left[ (\sqrt{2} + 1)\sqrt{P}.\alpha_g \right] \cdot \sum_{j=0}^{l-1} (1/2^j) + \left[ ((\sqrt{2} + 8)n^2.\beta_g/(2\sqrt{P}) \right] \cdot \sum_{j=0}^{l-1} (1/2^j). \] (4.6)

Analogously, adding the results of Lemma 4.4.5 and Lemma 4.4.6, the worst-case delay seen by a processor is \((\alpha_g + \beta_g \cdot m^2/(2P_i)) \cdot \sqrt{2P_i} + (\alpha_g + \beta_g \cdot 6m^2/P_i) \cdot \sqrt{P_i} \) for a horizontal separator. Using \(i = 2j - 1, 1 \leq j \leq l\), \(m = n/2^j\) and \(P_i = P/2^{2j-1}\), this gives

\[ (\alpha_g + \beta_g \cdot n^2/(4P)) \cdot (2\sqrt{P}/2^j) + (\alpha_g + \beta_g \cdot 3n^2/P) \cdot (\sqrt{2P}/2^j) \]

for the delay. To obtain the worst-case delay for a processor over all levels of vertical separators, we sum the above over all \(j\):

\[ \left[ (\sqrt{2} + 2)\sqrt{P}.\alpha_g \right] \cdot \sum_{j=1}^{l} (1/2^j) + \left[ ((1 + 6\sqrt{2})n^2.\beta_g/(2\sqrt{P}) \right] \cdot \sum_{j=1}^{l} (1/2^j). \] (4.7)
Finally, we approximate the sum for the vertical separators in equation (4.6) by letting $j$ run from 1 to $l$ instead of 0 to $l-1$, use the fact that $P = 2^l$, and add the results of equations (4.6) and (4.7) to obtain:

$$T_{comm} = \alpha_y \cdot (3 + 2\sqrt{2})\sqrt{P} + \beta_y \cdot (9 + 7\sqrt{2})n^2/(2\sqrt{P})(1 - 1/\sqrt{P}).$$

Number of messages for distributed sparse BLOCC

In Theorem 4.4.3 we derived communication time in a critical path of computations, which determines the parallel time, assuming that the processors are synchronized at the end of the computations at every level. We now derive the total number of messages sent during the factorization using the sparse BLOCC scheme, without using the performance model. We show that the total number of messages is $O(P\sqrt{P})$. The intuition for this result is as follows. Naik [54] showed that the data traffic for the BLOCC scheme is $O(n^2\sqrt{P})$. In this partitioning scheme, the size of a block is constant, namely $O(n^2/P)$. The number of messages is then the ratio of the data traffic to block size, which is $O(P\sqrt{P})$, and $O(\sqrt{P})$ per processor, if all the processors are used symmetrically.

To formally derive the number of messages, we first prove a lemma for number of messages sent to a processor assigned to a separator at level $i$, from any separator that is at a higher numbered level. Let the '+'-separator at the topmost level be the level-0 separator, the four '+'-separators at the next level be the level-1 separators, and so on, until level $l-1$. A level-$i$ separator could therefore be either a vertical separator or a horizontal separator. Consider a separator $S$ at level $i$. The separator is assigned $P_i$ processors where $P_i = P/2^i$ if it is a vertical separator, and $P/2^{i+1}$ if it is a horizontal separator. A separator $S'$ is said to update a block $B$ in another separator $S$ if any block in $S'$ updates $B$, and $S'$ is said to update $S$ if it updates any block in $S$.

**Lemma 4.4.7:** Any processor assigned to a level-$i$ separator $S$ receives $O(\sqrt{P_j})$ messages from any level-$j$ separator $S'$ which updates $S$. 
Figure 4.6: Separator $S'$ updates separator $S$.

Proof:

The proof is based on the geometry of the block partitioning in the Cholesky factor. Let $S$ be a level-$i$ separator and let $S'$ be a level-$j$ separator that updates $S$, $j > i$. In the sparse BLOCC scheme, each of the $P_i$ processors assigned to $S$ is assigned two partitions in $S$, one in the triangle at the diagonal and one in the off-diagonal rectangles. Let these partitions be $B_t$ and $B_r$, respectively. We show that the computation of each of $B_t$ and $B_r$ requires $O(\sqrt{P_j})$ blocks from $S'$. Since each block is sent in a separate message, it then follows that $P_i$ will receive at most $O(\sqrt{P_j})$ messages from $S'$.

See Figure 4.6 for an illustration of the updates. Figure 4.6(a) shows a partition $B_t$ in the triangle of $S$ being updated by $S'$, and Figure 4.6(b) shows a partition $B_r$ in one of the off-diagonal rectangles of $S$, being updated by $S'$. The updated block is shown by the solid rectangle while the actual updated portion is shown by the enclosed shaded portion. Henceforth we will refer to a partition as a block.

We refer to a row of blocks in the partitioning of either the triangle or the off-diagonal rectangles as a block row. A block row of $S'$ is aligned with the row extent of block $B$ of $S$, if the row extent of the block row intersects the row extent of $B$. A block row of $S'$ is aligned with the column extent of block $B$ of $S$, if the rightward projection of the block row intersects with the upward projection of the column extent of $B$. We will refer to the size of a row extent as its height. A block row of $S'$ has $\sqrt{P_j}$ blocks.
Consider the block $B_t$. $B_t$ can be either a triangular partition on $S$'s diagonal, or it can be a square partition as shown in Figure 4.6(a). $B_t$ is a rectangular partition. A pair of block rows of $S'$ will update a block $B$ in $S$, if, in general, one block row is aligned with the row extent of $B$ and the other block row is aligned with the column extent of $B$. In the case where $B_t$ is a triangular partition, a single block row may update $B_t$ by itself, without a pairing block row. However, for the worst case scenario, we will assume that updates are always computed by pairs of block rows.

Let $B_t$ and $B_r$ be assigned processor $p$. For every pair of block rows in $S'$ that updates $B_t$ or $B_r$, at most $2 \times \sqrt{P_j}$ messages are sent to $p$. How many such pairs of block rows of $S'$ can update $B_t$ or $B_r$? Note that the height and width of a block in a separator of size $m$, which is assigned $P_i$ processors, is within a constant factor of $m/\sqrt{P_i}$. Substituting $m$ with $n/2^i$ and $P_i$ with $P/2^{2i}$, we see that the height or width of a block in any separator is within a constant factor of $n/\sqrt{P}$. Clearly then, the number of pairs of block rows of $S'$ that can update either $B_t$ or $B_r$ is a bounded constant. It follows that the number of messages sent to $p$ from $S'$, in order to update $B_t$ or $B_r$, is $O(\sqrt{P_j}).$

We now prove the theorem on number of messages.

**Theorem 4.4.4:** The total number of messages sent during the factorization is $O(P \sqrt{P}).$

**Proof:**

Consider a level-$i$ separator $S$. From Lemma 4.4.7 we know that a level-$j$ separator $S'$ would send $O(\sqrt{P_j})$ messages to any processor which is assigned to $S$. Assume that processor $p$ is assigned to compute $B_r$ and $B_t$ in $S$. We compute the number of separators that update $B_r$, and therefore, the number of messages that are sent to $p$, in updating $B_r$. The messages to $B_t$, also assigned to processors $p$, will be no more than the messages to $B_r$.

To compute the separators that update $B_r$, we first identify the separators that update a non-zero $L_{g_c,h_c}$ in $B_r$, where column $g_c$ corresponds to grid point $g$ and column $h_c$ corresponds to grid point $h$. These separators are exactly the same as those...
which contribute to the external work of the non-zero element \( L_{y_r, b_r} \) in a rectangular block discussed in the proof of Theorem 4.4.1, referring to Figure 4.5. For an order of complexity analysis, we can assume that the heights and widths of the blocks are the same. Then, the same set of separators update all the non-zeros elements in \( B_r \). Counting the number of updating separators, we observe that there are at most four updating separators at each level \( k \), \( i \leq k < l \).

In addition, the number of messages sent to \( B_t \) or \( B_r \) from other processors assigned to the same separator \( S \) is \( O(\sqrt{P}) \). Thus, the total number of messages sent to \( p \), to update \( B_t \) or \( B_r \), is \( c \cdot \sum_{j=1}^{l-1} \sqrt{P_j} \) for some constant \( c \).

Processor \( p \) is assigned two blocks in some separator at every level \( i \), \( 0 \leq i < l \). So to get the total number of messages sent to \( p \), we have to sum the above over all levels \( i \). We have, therefore, number of messages is \( c \cdot \sum_{i=0}^{l-1} \sum_{j=1}^{l-1} \sqrt{P_j} \). This sum is \( c_1 \cdot \sqrt{P} (1 - \log p / \sqrt{P}) \) for some constant \( c_1 \). Summing this over all processors, asymptotically, we have that the total number of messages sent during the factorization is \( O(P\sqrt{P}) \), which proves the theorem.

\[ \Box \]

### 4.5 Conclusions

Schreiber, in [68], investigates the scalability of distributed sparse Cholesky factorization using column-mapped methods and two dimensional block mapping methods. The author defines an algorithm for this problem to be scalable if it maintains efficiency bounded away from zero as the number \( P \) of processors grows and the size of the data structures grows roughly linearly in \( P \). For dense Cholesky factorization, the author has shown that the column-based partitioning methods are not scalable. Using analytical results on the communication volume of the fan-out scheme for the grid problem, and with the help of simulation results, the author has argued that the column-based methods are not scalable in case of sparse matrices also.

The computational work in factoring the grid matrix is \( O(n^3) \) and the number of nonzero elements in the factored matrix is \( O(n^2 \log n) \) [28]. Thus, with both column
and block partitioning methods, the average computational work associated with each processor is $O(n^3/P)$ and the average number of nonzero elements assigned to each processor are $O(n^2 \log n/P)$. Using the best analytical results available for any of the three column-based methods, viz. fan-in, fan-out and multifrontal, the total communication volume is $O(n^2P)$, where $P$ is the number of processors and the total number of messages transmitted is $O(nP \log P)$. For the BLOCC scheme the total communication volume is $O(n^2 \sqrt{P})$, and the total number of messages is $O(P \sqrt{P})$. Thus, for the column-based methods, the average communication volume per processor is $O(n^2)$ and the average number of messages associated with each processor is $O(n \log P)$. The same quantities under the block partitioning scheme are $O(n^2/\sqrt{P})$ and $O(\sqrt{P})$, respectively. We use the number of nonzero elements as the problem size since it is directly related to the size of the data structures used in all the methods. Suppose that the grid size is increased so that the number of nonzeros in $L$ is increased by a factor of $c$. The total computational work increases by a factor linearly proportional to $c^{3/2}$. If we increase the number of processors by the same factor, so as to keep the size of the data structures on each processor the same, in both column and block approaches the average computation per processor increases roughly by a factor of $\sqrt{c}$. In the case of column-based methods, the average communication volume per processor increases by a factor that is roughly proportional to $c$ and the number of messages per processor increase by a factor of $\sqrt{c}$. In the case of block partitioning both quantities increase roughly in proportion to $\sqrt{c}$. Thus, in the block case both the computation work and the communication cost increase by a factor of $\sqrt{c}$, whereas in the column case, the communication cost increases by more than a factor of $\sqrt{c}$ while the computation cost per processor increases only by a factor of $\sqrt{c}$. In fact, it is not difficult to see that in the column-based methods, the number of processors cannot be increased by any significant fraction of $c$ if one were to maintain the same proportion between the computational work and the communication cost. In other words, to maintain performance of the column methods, the problem size must increase at a much faster rate than the number of processors - a clear indication of nonscalability. The block-based methods, on the other hand, scale well under the assumptions made here.
In other work, Rothberg and Gupta, in [60], propose a block fan-out algorithm for distributed sparse Cholesky factorization. For the grid problem, the communication volume is $O(n^2 \log n\sqrt{P})$ when parallelized on $P$ processors. Their block-to-processor assignment scheme assumes that the processors are laid out in the form of a $\sqrt{P} \times \sqrt{P}$ grid. Using domains [3, 4] reduces the volume to $O(n^2 \log P \sqrt{P})$.

Our partitioning strategy uses the BLOCC scheme as the building block. The BLOCC scheme is suitably modified and made more flexible, so that the partitioning method works for both structured and unstructured matrices, on architectures with different computation and communication characteristics. The important question that needs to be answered now is: how practical is a general block-partitioning method? We provide an answer to this question in the succeeding chapters.
Chapter 5

Block partitioning for sparse Cholesky factorization

5.1 Introduction

The theoretical results in Chapter 4 show that block partitioning is more scalable than the existing column partitioning schemes for factoring matrices arising from structured grid problems. The issue that needs to be addressed next is whether this block partitioning scheme can be made practical and if so, what are the issues that need to be addressed to realize high performance.

As we pointed out in Chapter 4, the theoretical results are based on a model which has certain limitations. One of the issues that we have to deal with in a practical block partitioning implementation is the architectural granularity, which is not accounted for in the model. The sparse BLOCC partitioning structure is solely determined by the number of processors. This can lead to two problems. If the blocks are larger than dictated by the granularity of the machine, the load balancing becomes a difficult task. On the other hand, if the blocks are too small to satisfy the machine granularity, communication and scheduling costs predominate. A practical implementation must allow parametric control in which the grain size of partitions can be adjusted to suit the target machine. A second issue we need to consider is the asynchrony of the algorithm. To derive the theoretical results we assumed that the computations go level by level, and no computation on a new level could start until all the computations at the previous level had been completed. While this is reasonable for structured matrices, it is an unnecessary imposition for unstructured matrices where synchronization may add significantly to the execution time. A general practical implementation should work equally well for both structured and unstructured matrices and general sparse Cholesky factorization
consists of data-driven computations. Thirdly, for a fixed machine granularity, single-
processor performance depends largely on the shape of blocks which is determined by
data access patterns in the algorithm, cache behavior, and vector lengths. A detailed
analysis of blocking for hierarchical memory is done in [59]. We use the BLOCC scheme
as the basis for our partitioning, but enhance it considerably to satisfy the requirements
pointed out by the preceding discussion.

In [75], experimental results from iPSC/860 are presented comparing the perfor-
mance of a column-based method with that of our 2-D block-based method for dis-
tributed sparse Cholesky factorization. The results indicate that 2-D block partitioning
reduces the total number of messages and the total volume of data transferred among
processors. In addition, it results in improved cpu performance because of better uti-
lization of cache and floating-point unit.

Although distributed sparse solvers based on 2-D partitioning have shown promise
for better performance and scalability, a major practical difficulty lies in partitioning
the sparse matrix and allocating and scheduling the partitions among processors so that
the resulting communication overhead is low and the load is well balanced. Manually
extracting different block partitions of such a matrix would be an extremely tedious and
error-prone task. Large structured and unstructured matrices encountered in real-life
applications may contain several million non-zero entries and, in such cases, any type
of systematic block partitioning by hand is simply impractical. To address these issues,
we started our investigation of general 2-D block partitioning and scheduling in [73],
where we introduced an automatic, general purpose block-based partitioning scheme
that takes into account the sparsity structure of the matrix in extracting the parti-
tions, and allows for control of the partition granularity. Using some of the principles
proposed in that paper, we have developed a parallel partitioner for sparse Cholesky
factorization on message passing multiprocessor systems, first described in [74]. The
partitioning is a mix of dense blocks and sparse columns, and allows for parametric
control to make the partitioning sensitive to both the matrix structure and the ma-
chine granularity. With such a hybrid partitioning method, higher performance and
scalability is achievable, provided a right set of parameters is used in partitioning the
sparse matrix [74, 72]. In [75], we examined, in detail, the effect of two partitioning parameters on the computation speeds, communication costs, extent of processor idling because of load imbalances, and bookkeeping overheads during the parallel numerical factorization using a sparse block code on an iPSC/860 machine. In [60], a block method has been suggested for partitioning sparse matrices in Cholesky factorization. Based on simulation results, the authors have claimed their method to be efficient.

In this chapter, we describe the design of the parallel partitioner. The efficiency, flexibility, and scalability of the partitioner, and the overall performance of the block-partitioned sparse Cholesky factorization for a variety of real-life, structured and unstructured test matrices is discussed in detail in Chapter 7. In order to generate the motivation for our 2D block-based approach to partitioning, we briefly survey the implementation and performance issues in existing distributed column-based methods for sparse Cholesky factorization.

The partitioning is preceded by pre-partitioning, which identifies the sparsity structure of the matrix. We first describe the pre-partitioner. Then we discuss block partitioning and give a sequential block-based sparse Cholesky factorization algorithm. Lastly, we discuss the design of the algorithms and data structures in the partitioner, and describe the parallelization of the partitioner. The input to the partitioning process is the symbolically factored lower triangular matrix $L$.

As a running example, we will use the matrix of of Figure 5.1, which shows the zero-nonzero structure of the lower triangular factor of a $49 \times 49$ matrix (a $\bullet$ represents a nonzero).

5.2 Distributed column sparse Cholesky factorization

In Section 2.1 and Section 2.2, we described sequential dense and sparse Cholesky factorization, and the tasks $c\text{mod}$ and $cd\text{iv}$ in column-based partitioning. The notion of elimination tree structure has been used extensively to describe and implement many aspects of sparse matrix computations (see e.g., [38, 44, 67]). Each node in the elimination tree corresponds to a column of $A$. The parent of any node corresponding to
column $i$, $i < N$, is the node corresponding to the column defined by $\min\{k|L_{k,i} \neq 0\}$; i.e., there is an edge $(i, k)$ in the tree if and only if $k$ is the row number of first non-zero element in column $i$ below the diagonal. The elimination tree represents a computation graph for the factorization process and can be defined using the structure of $L$ and hence from the structure of $A$. Figure 5.2(a) shows the sparsity structure of a lower triangular matrix, where an $\times$ indicates position of a nonzero in the lower triangular part of $A$ and an $\circ$ indicates a fill. The corresponding elimination tree is shown in Figure 5.2(b). Note that column 5 of $L$ cannot be computed until columns 1 and 4 are computed. In the elimination tree, this is indicated by the fact that node 4 is a child of node 5. Note also that, by definition, the elimination tree does not have transitive edges: column 6's computation depends on columns 3, 4 and 5 but the edges $(3, 6)$ and $(4, 6)$ are not included. We will use the concept of elimination tree to describe some of the parallelization techniques. Also, in the following discussion of the fan-out, fan-in and multifrontal methods, we use the term supernode which is defined as a set of contiguous columns $i, i+1, \ldots, j$ in $L$ such that each column in this set, except for
column $i$, has only one child in the corresponding elimination tree and the zero-nonzero structure of columns $i + k$ and $i + k + 1$ in $L$, $i \leq i + k < j$, differ only row $i + k$ where the former has a nonzero element. We will denote a supernode consisting of columns $i$ through $j$ by $S_{i,j}$. If the supernode is a single column $i$, we will denote it by $S_i$. Figure 5.3 shows the columns of the matrix of Figure 5.1, in which the columns are partitioned into supernodes by the vertical lines running through the matrix.

In describing the three methods, we assume that the columns of $L$ are mapped onto the processors according some scheme such as wrap mapping or subtree-to-subcube mapping. We say that a processor owns column $i$, if that processor performs the $cdiv(i)$ task. We denote a processor by $P_{(i)}$, if that processor owns column $i$. We denote by $pQ$, the processor which is numbered $Q$.

In the distributed fan-out method (see, for example [25]), for any column $j$, all $cmod(j,k)$ tasks and $cdiv(j)$ task take place in the processor $P_{(j)}$. Further, as soon as column $j$ is factored, it is sent to all the processors that perform a $cmod(i,j)$ task, for some column $i$, $i > j$; i.e., the factored column $j$ is fanned-out. The destination processor need to be sent only one copy of column $j$ even if more than one $cmod(i,j)$ task is performed at that processor. Also, only the nonzero values need to be transmitted. However, if the nonzero structure of column $j$ is not known a priori to the destination processor, then along with the nonzero values, the corresponding row index vector must also be sent, which adds to the communication cost. In performing a $cmod(*,j)$ task, the index vector of $j$ has to be matched with the index vector of every updated column.
Figure 5.3: Supernodes in the lower triangular factor.

on the destination processor. This index matching adds to the bookkeeping cost and is a major source of computational inefficiency in the fan-out algorithm.

In the distributed fan-in method (see, for example [5]), for any column $j$, the $\text{cm}od(j, k)$ tasks for different $k$'s are distributed among several processors. Typically, the product formation part of a $\text{cm}od(j, k)$ task is performed by processor $P_{(k)}$. Each such processor transmits the partial updates of column $j$ to processor $P_{(j)}$, i.e., the updates are fanned-in. If $P_{(k)}$ owns more than one updating column of $j$, the processor may compute these updates of $j$ and send only the aggregate sum vector. Furthermore, the partial updates from various processors, may themselves be combined while they are being fanned-in to processor $P_{(j)}$. Note that to perform the necessary computations, processor $P_{(k)}$ does not require the structure of column $j$. However, if $P_{(k)}$ does not use the structure of column $j$, then either it must send an index vector along with the
aggregate sum vector or it must use a dense vector (with zeros and nonzeros) of length $N - j + 1$ to compute and send the aggregate sum vector. As in the fan-out method, the $cdiv(j)$ task is executed in the processor $P_{(j)}$.

Compared to the fan-out method, the fan-in method has lower communication costs, both in terms of total number of messages and the amount of data transmitted during the factorization [33]. These gains are significant when the sparse matrix has some regularity such as those representing uniform grid graphs. The main drawback of the fan-in method as compared to the fan-out method, is the difficulty in balancing the computational work load among processors. Balancing the load becomes harder if supernodes are assigned to processors as single units in a bid to improve computational performance.

The main source of computational inefficiency in the two methods described above is in the use of indirect addressing. This difficulty is overcome in the multifrontal methods where dense matrix partial factorizations are performed [18, 45]. These algorithms make use of the structure associated with the columns of a supernode. Recall that the structure of the columns associated with a supernode $S_{i,j}$ is such that every column $k$, $i \leq k \leq j$, participates in the same set of $cmod(m,k)$, $m > j$, tasks and the resulting update vectors have the same nonzero structure. Thus, all $cmod$ tasks associated with the columns of a supernode can be computed using the same structure. In computing these partial updates, the affected columns are represented by a dense lower triangular matrix called frontal matrix. An obvious advantage is that these updates can be done efficiently without resorting to indirect addressing. As a further optimization, the frontal matrix of a supernode also includes the structure of the columns of that supernode, again in the compressed form. The $cdiv$ operations are performed in the corresponding part of the dense frontal matrix prior to performing the partial updates. A supernodal elimination tree is used in determining the order in which the supernodes are evaluated and the order in which the partial updates are merged to get the final updates. The key aspects to note here are that the columns are grouped together so that the corresponding $cdiv$ and $cmod$ operations can be performed on dense matrices. However, these methods are not without overheads. The inefficiencies arise in
the *assembly* and the *merge* steps. As a consequence, these methods yield the best computational efficiency when the frontal matrices are large.

In the distributed implementations of multifrontal methods, columns are distributed among processors, as in the fan-out and fan-in methods [48, 4, 57, 69]. To achieve reasonable balance of load, the work involved in the partial factorization and in the assembly steps of a frontal matrix may itself get distributed among processors. Columns of partial updates need to be communicated both in the assembly and merge steps. Good performance has been reported for sparse matrices that yield well distributed and large frontal matrices. Among the three column-based methods that are studied extensively, the distributed multifrontal methods have delivered the best overall performance. This is primarily because of the superior single processor performance resulting from the use of dense data structures. However, even with the multifrontal methods there are limitations to the expected parallel performance. Since these methods are column-based, the data locality is poorly utilized. This affects both the cache performance and the communication costs. As a result, they scale poorly with increasing number of processors. Note that grouping columns into larger units increases the data locality, but decreases the available parallelism. This gives rise to difficulties in load balancing and scheduling.

In Section 4 we showed a 2-D block based partitioning scheme that overcomes some of these drawbacks. We now discuss the design of a partitioner built around this partitioning scheme, beginning with design of the pre-partitioner which identifies naturally occurring 2-D blocks in the lower triangular factor, $L$.

### 5.3 Pre-partitioner

In partitioning the sparse matrix $L$, the partitioner makes use of some of the general properties observed in the sparsity structure of $L$. In most practical cases, the sparsity structure of $L$ is such that the nonzero elements conglomerate towards the right edge, forming dense blocks. Towards the left of the matrix, there is very little block structure. If the ordering used is a minimum-degree like ordering, then, overall, the sparsity
decreases gradually from left to right, i.e., the matrix becomes more dense. Moreover, one observes that there are triangular blocks of nonzero elements at the diagonal and rectangular or near-rectangular nonzero blocks below these triangles. See Figure 5.4. In that figure, the numbers along the diagonal are column numbers. Accurate identification of these blocks is the first step in partitioning the sparse matrix.

Figure 5.4: Clusters and rectangles.

Starting with the Lindez and Lstruct arrays, the pre-partitioner efficiently identifies the naturally occurring triangular and rectangular dense blocks in L. For this, the pre-partitioner first divides the matrix L into disjoint bands of columns called clusters. Informally, a cluster is characterized by a dense triangle at the diagonal and several dense rectangles below the triangle. In the degenerate case, a cluster is a column with the diagonal element representing the triangle. For instance, in Figure 5.4, columns 6 through 7 form a cluster, columns 15 through 18 form a cluster and so do columns 39
through 49. Columns 1 through 5 are not part of any cluster.

We define the following precise terms with reference to the structure of the symbolically factored lower triangular matrix $L$. We ignore any zeros that may be introduced into the matrix as a result of cancellations during the factorization.

**Definition 5.3.1:** A triangle $T_{m,n}$ is the block of elements $\{L_{i,j} \mid m \leq i, j \leq n, i \geq j\}$. We alternatively represent this triangle as $(m, n)$. $T_{m,n}$ is a dense triangle if none of its elements is zero. $T_{m,n}$ is maximally dense if neither $T_{m,n+1}$ nor $T_{m-1,n}$ is dense, i.e., it is not properly contained in another dense triangle.

For example, in Figure 5.5(a), $T_{c,c2}$ is a triangle which contains some zeros. $T_{c,c1}$ is a dense triangle, but it is not maximally dense. $T_{c,c1}$ and $T_{c',c2}$ are maximally dense triangles.

![Diagram](attachment:diagram.png)

**Figure 5.5:** Dense triangles and rectangles.

In Figure 5.4, $T_{10,13}$ is a triangle with one zero in $L_{23,19}$. $T_{19,22}$ and $T_{20,23}$ are maximally dense triangles.

**Definition 5.3.2:** A cluster $C_{m,n}$ is a band of columns $m$ through $n$ such that $T_{m,n}$ is a dense triangle. The cluster width of $C_{m,n}$ is $n - m + 1$. 
The following is a formal definition of a rectangle. A rectangle is always defined in the
context of a cluster.

**Definition 5.3.3**: A rectangle $R_{r_i,r_2,c,n}$ in a cluster $C_{m,n}$ is the block of elements
$\{L_{i,j} \mid n < r_1 \leq i \leq r_2, m \leq c \leq j \leq n\}$. We alternatively represent this rectangle as
$(r_1, r_2, c, n)$. $R$ is a dense rectangle if none of its elements is zero. $R$ is maximally dense
if none of the rectangles $(r_1 - 1, r_2, c, n)$ or $(r_2, r_2 + 1, c, n)$ or $(r_1, r_2, c - 1, n)$ is dense.

In Figure 5.5(b), we show a portion of some matrix below the triangle of cluster
$C_{1,5}$. There are five maximally dense rectangles: $R_{7,8,3,5}$, $R_{10,11,2,5}$, $R_{10,13,5,5}$, $R_{14,15,4,5}$
and $R_{15,15,2,5}$. Rectangles $R_{12,13,5,5}$ and $R_{14,14,4,5}$ are dense, but not maximally dense
rectangles. The significance of the solid-line and dotted-line boxes in that figure will
be clear shortly. Each cluster in Figure 5.4, except for $C_{39,49}$, shows one or more off-
diagonal rectangles.

The main functions of the pre-partitioner are:

1. To identify maximally dense triangles in $L$ given $L_{index}$ and $L_{struct}$. From these,
disjoint clusters are identified. We call this the cluster identification step.

2. To identify a set of disjoint rectangles in each cluster. We call this the rectangle
identification step.

We now present algorithms for the cluster identification and rectangle identification
steps and analyze their time complexities. In the rest of the discussion, when we refer
to a cluster, we mean a multi-column cluster.

### 5.3.1 Cluster identification

The following assertion is used to construct a fast algorithm for cluster identification.

**Claim 5.3.1**: Let $T_{k,m}$ be a dense triangle and $T_{k-1,m}$ not be a dense triangle. (i) if
$L_{m+1,k}$ is a nonzero element then $T_{k,m+1}$ is a dense triangle. (ii) if $L_{m+1,k}$ is zero then
$T_{k,m}$ is a maximally dense triangle.

**Proof**: We prove the first part of the claim by showing that $L_{m+1,j}$ is a nonzero
$\forall j$, $k \leq j \leq m + 1$. Since $T_{k,m}$ is a dense triangle, $L_{j,k}$, $k \leq j \leq m$, is a nonzero element.
Thus, when $L_{m+1,k}$ is a nonzero, in Cholesky factorization $L_{j,k}$ and $L_{m+1,k}$ combine to produce a nonzero element $L_{m+1,j}$, \( \forall j, k < j \leq m + 1 \).

The second part of the claim follows from the definition. \( \qed \)

Clusters are obtained from maximally dense triangles as follows. If a maximally dense triangle $T_{m,n}$ does not intersect another, then the cluster $C_{m,n}$ is reported. If two maximally dense triangles $T_{c,c1}$ and $T_{c',c2}$ intersect each other, $c < c' < c1$ (see Figure 5.5(a)), then the band of columns from $c$ to $c' - 1$ is treated as a cluster. In other words, of the two maximally dense triangles, the one to the left contributes a cluster which is obtained by appropriately truncating those columns which intersect with the maximally dense triangle to the right. To give another example, in Figure 5.4, the maximally dense triangles $T_{34,38}$ and $T_{36,45}$ intersect each other. $T_{34,38}$ contributes the cluster $C_{34,35}$.

Algorithm $Id.Clusters$ in Figure 5.6 gives a procedure to identify maximally dense triangles from which disjoint clusters are identified. The input to $Id.Clusters$ are the Lstruct and Lindex arrays representing the $N \times N$ matrix $L$. For clarity, in the following algorithm, we only show the identification of maximally dense triangles and reporting of the disjoint dense triangles. The disjoint cluster associated with each reported dense triangle can be identified in a straightforward manner.

The algorithm scans columns left to right, beginning with the first column, tracing an outline of the nonzero elements that form intersecting triangular blocks along the diagonal. We explain the rest of the algorithm by referring to Figure 5.5(a) and Figure 5.4. In Figure 5.4, the numbers along the diagonal are column numbers. In the $\textbf{while}$ loop of step 2, the selected column $c$ is scanned downwards, beginning at the diagonal or from the bottom edge of a predetermined triangle. For the example in Figure 5.5(a), $c$ is scanned from the diagonal. Take $c$ to be 34 in Figure 5.4. The column $c$ is scanned until a zero in that column is encountered or until all rows are exhausted. In the latter case, the entire block of the matrix below (and including) the diagonal and to the right of (and including) column $c$ is a dense triangle. In scanning $c$ in the Lstruct array, a zero is encountered when two successive nonzero elements in that column are more than one row apart. In the example of Figure 5.5(a), a zero is encountered in row
procedure Id_Cluster

0. $c \leftarrow 1$, $startrow \leftarrow 0$

1. while ($c \leq N$) do
   
   \hspace{1em} $index \leftarrow \text{Lindex}[c] + startrow$

2. \hspace{1em} while ($index < \text{Lindex}[c+1] - 1 \text{ and } \text{Lstruct}[index + 1] = \text{Lstruct}[index] + 1$) do
   \hspace{2em} $index \leftarrow index + 1$ endwhile
   \hspace{1em} $c1 \leftarrow \text{Lstruct}[index]$
   \hspace{1em} $entry \leftarrow index - \text{Lindex}[c]$
   \hspace{1em} if ($c1 = N$) then report triangle $T_{c,N}$, quit
   \hspace{1em} else $k \leftarrow c + 1$ endif

3. \hspace{2em} $done \leftarrow false$
   \hspace{1em} while ($k \leq c1 \text{ and } (\text{not } done)$) do
   \hspace{3em} $index \leftarrow \text{Lindex}[k] + entry$
   \hspace{3em} if ($\text{Lstruct}[index] = c1 + 1$) then $done \leftarrow true$
   \hspace{3em} else $k \leftarrow c + 1$, $entry \leftarrow entry - 1$ endif
   \hspace{2em} endwhile

4. \hspace{1em} if ($k = c1 + 1$) then report triangle $T_{c,c1}$, $c \leftarrow c1 + 1$, $startrow \leftarrow 0$
   \hspace{1em} else report triangle $T_{c,k-1}$, $c \leftarrow k$, $startrow \leftarrow entry$ endif

endwhile

Figure 5.6: Procedure for identification of clusters.
$c_{1+1}$. In Figure 5.4, $c_1$ would be 38. By claim 5.3.1, $T_{c,c_1}$ is a maximally dense triangle. Now the columns to the right of $c$ are scanned to identify any intersecting maximally dense triangles. This is done in step 3 of the algorithm, where nonzero elements are searched on row $c_1+1$ starting at column $c+1$. In the CSC representation, this would be equivalent to checking whether, in any column $k$, $c < k \leq c_1$, the row number of the entry immediately succeeding an entry in row $c_1$ is $c_1+1$. If the first nonzero element is on the diagonal, then the maximally dense triangle $T_{c,c_1}$ is reported in step 4 and a new triangle is searched starting at the diagonal element on column $c_1+1$. Otherwise, there is another maximally dense triangle $T_{c,c_2}$, $c < c' \leq c_1$, intersecting $T_{c,c_1}$, as in the example of Figure 5.5(a), and the search in step 3 ends when $k$ is equal to $c'$. Also, in Figure 5.4, $c'$ corresponds to 36. In this case the triangle $T_{c,c'-1}$ is reported and the next iteration of the algorithm begins at row $c_1+1$ in column $c'$. Another set of $c, c'$ and $c_1$ for which the above situation holds, in Figure 5.4, is $c = 36, c' = 39$ and $c_1 = 45$. In this case, the triangle $T_{36,38}$ is reported.

When the algorithm terminates, all the consecutive maximally dense triangles are identified; the non-intersecting maximally dense triangles are reported as such and when there is a sequence of intersecting maximally dense triangles, as in the example above, a list of disjoint dense triangles followed by a maximally dense triangle is reported.

**Claim 5.3.2:** The worst case time complexity of algorithm *Id_Clusters* is $O(N)$ comparisons for an $N \times N$ lower triangular matrix.

**Proof:** Consider one iteration of step 1. Let the scan begin in column $c_1$ at row $r_1$. See Figure 5.7.

Suppose step 2 scans down the column until row $r_2$. This scan makes $r_2 - r_1 + 1$ comparisons. Now let step 3 scan along row $r_2$ until column $c_2$, making $c_2 - c_1 + 1$ comparisons. In this iteration, therefore, the algorithm has traced an 'L'-shaped contour (a vertical section with a horizontal section at right angles to it). We will call this one *step*. In the next iteration, there will be another step starting at column $c_2 + 1$. The algorithm begins scanning at the element $L_{1,1}$ and when all the steps are traced, each row and column is scanned exactly once. When a row or column is scanned, one
comparison is performed. It is easy to see that the total number comparisons is $2N$ and hence the claim.

\[\text{\bf 5.3.2 Rectangle identification} \]

We describe the process of disjoint rectangle identification within each cluster reported by algorithm \textit{Id\_Cluster}. The following assertion is used to construct a fast algorithm for rectangle identification.

\textbf{Claim 5.3.3:} In a cluster $C_{m,n}$, if $L_{i,j}$, $j \leq n < i$, is a nonzero element, then $\forall k$, $j < k \leq n$, $L_{i,k}$ is also a nonzero element.

\textbf{Proof:} By definition, associated with cluster $C_{m,n}$ there is a dense triangle, $T_{m,n}$. Thus, $\forall k$, $j < k \leq n$, $L_{k,j}$ is a nonzero element. If $L_{i,j}$, $j \leq n < i$, is a nonzero element, then in Cholesky factorization, $L_{k,j}$ and $L_{i,j}$ combine to produce a nonzero element in $L_{i,k}$.

Claim 5.3.3 asserts that if there is a nonzero element $L_{i,j}$ below the triangle in a cluster, then all the elements to its right in row $i$ of the cluster will be nonzeros. Also, in a cluster $C_{m,n}$, any nonzero element below the triangle $T_{m,n}$ will lie in some dense rectangle with right edge on column $n$.\]
Given a cluster $C_{m,n}$, disjoint dense rectangles are obtained from maximally dense rectangles as follows. If a maximally dense rectangle does not intersect another, it is reported as a disjoint rectangle. For instance, this is true of the maximally dense rectangle $R_{7,8,3,5}$ in Figure 5.5(b). If a maximally dense rectangle $R$ intersects one or more maximally dense rectangles, a disjoint rectangle is obtained by extracting the largest sub-rectangle from $R$ in such a way that none of the intersecting rectangles is split vertically. For instance, the maximally dense rectangle $R_{10,15,5,5}$ in Figure 5.5(b) gives the disjoint rectangle $R_{12,13,5,5}$. Adding either $L_{11,5}$ or $L_{14,5}$ to the rectangle in an attempt to enlarge it would split $R_{10,11,2,5}$ or $R_{14,15,4,5}$ vertically, respectively. The other disjoint rectangles reported in Figure 5.5(b) are $R_{10,11,2,5}$, $R_{12,13,5,5}$, $R_{14,14,4,5}$ and $R_{15,15,2,5}$. In Figure 5.4, the disjoint dense rectangles reported in $C_{15,18}$ are $R_{34,35,15,18}$, $R_{36,37,15,18}$ and $R_{38,39,19,19}$.

Algorithm $Id RECTS$ in Figure 5.8 gives the procedure to identify disjoint rectangles in cluster $C_{m,n}$. The input to this algorithm are Lstruct and Lindex arrays and the cluster $C_{m,n}$.

The algorithm $Id RECTS$ identifies all disjoint dense rectangles that have right edge on column $n$. In each iteration of the loop of Step 1, the algorithm identifies a new dense rectangle stretching to the left. The disjoint rectangles are obtained by scanning column $n$ downwards and by breaking any intersecting maximally dense rectangles row-wise so that the resulting disjoint dense rectangles have their right edge on column $n$.

An auxiliary array, last[$m : n$] is maintained whose size is equal to the cluster width of $C_{m,n}$. This array acts as an offset for each column of the cluster $C_{m,n}$, indicating the first nonzero element in each column that is not part of an already identified rectangle. This array is useful in skipping over the nonzeros in the interior of each rectangle, without examining them. This speeds up the identification process.

We now explain how the algorithm $Id RECTS$ works, using the example shown in Figure 5.5(b) and the matrix of Figure 5.4. The main while loop of step 1 checks whether there is a nonzero in column $n$ that is not contained in any rectangle reported thus far. If there is such a nonzero element, say in row $r1$, the detection of a new rectangle, say $R$, begins at that nonzero element. In Figure 5.5(b), the first-non zero
procedure Id_Rects

0. for $i = m$ to $n$ do last[$i$] = $n - i + 1$ endfor

1. while $(\text{Lindex}[n] + \text{last}[n] < \text{Lindex}[n + 1])$ do 
   index $\leftarrow \text{Lindex}[n] + \text{last}[n]$, r1 $\leftarrow \text{Lstruct}[\text{index}]$, c $\leftarrow n$
   
2. while $(c > m$ and Lstruct[Lindex[c - 1] + last[c - 1]] = r1) do $c \leftarrow c - 1$ endwhile

3. index $\leftarrow \text{Lindex}[c] + \text{last}[c]$
   if $(c = m)$ then
     index’ $\leftarrow 0$
   else
     index’ $\leftarrow \text{Lindex}[c - 1] + \text{last}[c - 1]$
   endif
   while $(\text{index} < \text{Lindex}[c + 1] - 1$ and Lstruct[index] = Lstruct[index + 1] $- 1$ and
     Lstruct[index’] $\neq$ Lstruct[index]) do
     last[c] $\leftarrow$ last[c] + 1
     index $\leftarrow$ index + 1
   endwhile
   r2 $\leftarrow$ Lstruct[index]

4. report rectangle $R_{c_1, r_2, c_n}$
   for $i = c$ to $n$ do last[$i$] $\leftarrow$ last[$i$] + r2 $- r1 + 1$ endfor
   endwhile

Figure 5.8: Procedure for identification of rectangles.
entry in column 5 is in row $r_1 = 7$. In step 2, columns in the cluster are scanned going right to left across columns, checking for a nonzero in row $r_1$. Since all rectangles above row $r_1$ have already been reported, $Lstruct[Lindex[c] + last[c]] \geq r_1$, $m \leq c \leq n$. When a column which does not have a nonzero in row $r_1$ is encountered, the algorithm has detected the left edge of $R$, say in column $c$, and the nonzeros seen thus far in row $r_1$, between columns $n$ and $c$, form the top edge of $R$. For example, in Figure 5.5(b), scanning row $r_1 = 7$ results in detecting an edge in column 3.

Step 3 scans downward along the left edge $c$ to find the bottom edge of $R$. This is detected when (a) the bottom row of the matrix is encountered, or (b) two successive nonzeros in column $c$ are more than one row apart, or (c) two successive nonzero entries are only a row apart but the second entry (in row $r$, say) has a companion entry in row $r$ of column $c - 1$, indicating an intersection of maximally dense rectangles. When the scan terminates, by claim 5.3.3, the rectangle $R$ has been completely identified. In Figure 5.5(b), case (a) would be encountered in detecting rectangle $(15, 15, 2, 5)$ while scanning down column 2; case (b) would be encountered in detecting rectangle $(7, 8, 3, 5)$ while scanning column 3 or while detecting rectangle $(10, 11, 2, 5)$ while scanning column 2; and case (c) would be encountered in detecting rectangle $(12, 13, 5, 5)$ while scanning column 5 or in detecting rectangle $(14, 14, 4, 5)$ while scanning column 4. Refering to Figure 5.4, case (a) would occur in detecting rectangle $(43, 49, 20, 23)$ in cluster $C_{20, 23}$, while scanning column 20 or in detecting rectangle $(39, 42, 30, 33)$ in cluster $C_{30, 33}$ while scanning column 30; case (b) would occur in detecting rectangle $(32, 33, 25, 28)$ in cluster $C_{25, 28}$ while scanning column 25; and case (c) would occur in detecting rectangle $(24, 25, 9, 9)$ in cluster $C_{8, 9}$ while scanning column 9.

Step 4 reports $R$. All the columns that make up the rectangle are marked as having examined $w$ more entries, where $w$ is the breadth of $R$. This marking is achieved by advancing by $w$ the entries in $last$ corresponding to the component columns of $R$. This ensures that the nonzero elements in $R$’s interior are not examined.

**Claim 5.3.4:** The worst case time complexity of algorithm $Id.Rects$ is $O(Z)$ comparisons, where $Z$ is the number of nonzeros contained in the reported rectangles.
Proof: The algorithm traces over the top and left edges of every rectangle reported. For a rectangle $R$, the top edge is traced in step 2 and the left edge is traced in step 3. Step 5 is equivalent to tracing the bottom edge of $R$. In any edge, each element is examined exactly once. Thus the time to trace an edge is proportional to the number of nonzeros in the edge, or the length of the edge.

In the worst case, every rectangle could be of unit height, and $Id RECTs$ examines all the nonzeros which make up the rectangles, giving the above complexity. In the best case, there is a single rectangle of height $l$ and breadth $w$ and the time taken by $Id RECTs$ is $O(l + w)$.

5.3.3 Parameters

So far we have described the functioning of the pre-partitioner assuming that each cluster is a multi-column cluster. In practice, sparse matrices contain several columns that do not form multi-column clusters. The pre-partitioner reports these as single column clusters without any further partitioning. For example, in Figure 5.4, columns 1 through 5, and columns 12, 19, 24 and 29 remain as single columns at the end of the pre-partitioning process. An observation is that multi-column clusters with small width perform well if decomposed into single columns, i.e., it does not pay to partition small-width multi-column clusters into triangles and rectangles. The cutoff point depends on the overall structure of the sparse matrix as well as on the underlying architecture.

To maintain this flexibility in deciding the cutoff point, the pre-partitioner makes use of the input parameter $W$ in the cluster identification and rectangle identification steps. The parameter $W$ specifies the minimum acceptable cluster width and acts as a switch between column-based and block-based partitioning for different regions of the matrix. Only those clusters with width at least $W$ are accepted as multiple column clusters. The rest are decomposed into single column clusters. For instance, in Figure 5.4, if $W$ is set to 3, the clusters $C_{6,7}$, $C_{8,9}$, $C_{10,11}$, $C_{13,14}$ and $C_{34,35}$ are all discarded, and their constituent columns are treated as single columns.
5.4 Block sparse Cholesky

In this section we describe the blocks in our partitioning method and the different types of tasks that arise in this block-based sparse Cholesky factorization. We then present a sequential block sparse Cholesky factorization code. We show how the nature of the inter-block dependencies determine all the different types of tasks, and give the computational code for the tasks. Finally, we discuss the factors that influence the partitioning process.

5.4.1 Block partitions

In the partitioning step, the dense triangle and rectangles in each cluster, as identified by the pre-partitioner, are partitioned into triangular and rectangular shaped sub-blocks. We refer to these as the unit blocks, i.e., blocks which are atomic units for allocation and scheduling. When there is no ambiguity, we refer to these simply as blocks. For uniformity, a single column is also considered to be a unit block. Figure 5.9 shows block partitions in the matrix of Figure 5.4. For the partitioning, \( W \) has been set to 3, so that only those clusters that are at least three columns wide are accepted for block partitioning, while the other clusters are broken down into single columns. In Figure 5.9, the triangle \( T_{30,33} \) is partitioned into two unit triangles and one unit rectangle. The rectangle \( R_{39,45,36,38} \) in cluster \( C_{36,38} \) is partitioned into four unit rectangles.

Once the unit blocks are identified, block-based computational tasks and the inter-block dependencies which produce these tasks are defined. In the column-based partitioning methods, the column-updates-column tasks are implicitly defined by the structure of the non-zeros within the columns. For block partitioning, however, the blocks themselves are non-uniform in nature and the partitioning varies with the sparsity structure of the matrix. The partitioner, therefore, computes an explicit BCR, a representation that was proposed and described in Chapter 3.
Figure 5.9: Unit blocks.
5.4.2 Task types

In the column-based partitioning and scheduling schemes, there are two types of computational tasks, \texttt{cm}	exttt{od} and \texttt{cdiv}. The \texttt{cm}	exttt{od} task denotes the update of one sparse column using another, and the \texttt{cdiv} task denotes the scaling of the elements of a column by the square root of the diagonal element in that column.

<table>
<thead>
<tr>
<th>No.</th>
<th>Task Description</th>
<th>Tuple Notation</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Column Updates Column</td>
<td>(&lt;\text{Col}, \text{Update}, \text{Col}&gt;)</td>
<td>CC</td>
</tr>
<tr>
<td>2</td>
<td>Column Updates Triangle</td>
<td>(&lt;\text{Tri}, \text{Update}, \text{Col}&gt;)</td>
<td>CT</td>
</tr>
<tr>
<td>3</td>
<td>Rectangle Updates Column</td>
<td>(&lt;\text{Col}, \text{Update}, \text{Rect}&gt;)</td>
<td>RC</td>
</tr>
<tr>
<td>4</td>
<td>Rectangle Updates Triangle</td>
<td>(&lt;\text{Tri}, \text{Update}, \text{Rect}&gt;)</td>
<td>RT</td>
</tr>
<tr>
<td>5</td>
<td>Column Updates Rectangle</td>
<td>(&lt;\text{Rect}, \text{Update}, \text{Col}&gt;)</td>
<td>CR</td>
</tr>
<tr>
<td>6</td>
<td>Triangle and Rectangle Update Rectangle</td>
<td>(&lt;\text{Rect}, \text{Update}, \text{Tri}, \text{Rect}&gt;)</td>
<td>TRR</td>
</tr>
<tr>
<td>7</td>
<td>Two Rectangles Update Column</td>
<td>(&lt;\text{Col}, \text{Update}, \text{Rect}, \text{Rect}&gt;)</td>
<td>RRC</td>
</tr>
<tr>
<td>8</td>
<td>Two Rectangles Update Triangle</td>
<td>(&lt;\text{Tri}, \text{Update}, \text{Rect}, \text{Rect}&gt;)</td>
<td>RRT</td>
</tr>
<tr>
<td>9</td>
<td>Two Rectangles Update Rectangle</td>
<td>(&lt;\text{Rect}, \text{Update}, \text{Rect}, \text{Rect}&gt;)</td>
<td>RRR</td>
</tr>
<tr>
<td>10</td>
<td>Triangle Updates Rectangle</td>
<td>(&lt;\text{Rect}, \text{Update}, \text{Tri}&gt;)</td>
<td>TR</td>
</tr>
<tr>
<td>11</td>
<td>Scale Column</td>
<td>(&lt;\text{Col}, \text{Scale}&gt;)</td>
<td>SC</td>
</tr>
<tr>
<td>12</td>
<td>Scale Triangle</td>
<td>(&lt;\text{Tri}, \text{Scale}&gt;)</td>
<td>ST</td>
</tr>
</tbody>
</table>

Table 5.1: The twelve generic update and scale tasks.

There are twelve different types of computational tasks in our block partitioning scheme. Ten of these are variants of update operations and two are variants of scaling operations. These variants are defined by the shapes of the unit blocks that participate in these operations, and each variant results in a different computational code. Accordingly, there are twelve different generic tasks. These are summarized in Table 5.1 using the tuple notation \(<\mathcal{P}, \text{Oper}, [\mathcal{P}], [\mathcal{P}]>\), where \(\mathcal{P}\) defines a partition in \(L\), and \(\text{Oper}\) is either an update or a scale type operation on a partition. The partitions are identified by their shape: "Col" for column, "Tri" for triangle and "Rect" for rectangle. The last column in the table gives acronyms for each of these tasks. We will refer to the tasks by their respective acronyms in the rest of the paper. CC is the same as \texttt{cm}	exttt{od} and \texttt{SC}...
is the same as cdiv. Note that there is no explicit task for the scaling of a rectangle. The reason for this will be apparent when we discuss the code for the computational tasks in Section 5.4.6. The different types of update tasks, except CC are schematically shown in Figure 5.10. Using examples drawn from Figure 5.9, Table 5.2 gives one representative for each type of update task. In that table, we use the notation $C_k$ for column number $k$.

<table>
<thead>
<tr>
<th>Name</th>
<th>Tuple notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>$&lt; C_{14}, \text{update}, C_{13} &gt;$</td>
</tr>
<tr>
<td>CT</td>
<td>$&lt; T_{20,21}, \text{update}, C_{19} &gt;$</td>
</tr>
<tr>
<td>CR</td>
<td>$&lt; R_{32,33,25,26}, \text{update}, C_{24} &gt;$</td>
</tr>
<tr>
<td>TR</td>
<td>$&lt; R_{32,33,27,28}, \text{update}, T_{25,26} &gt;$</td>
</tr>
<tr>
<td>TRR</td>
<td>$&lt; R_{32,33,27,28}, \text{update}, T_{25,26}, R_{32,33,25,26} &gt;$</td>
</tr>
<tr>
<td>RC</td>
<td>$&lt; C_{34}, \text{update}, R_{34,35,15,16} &gt;$</td>
</tr>
<tr>
<td>RRC</td>
<td>$&lt; C_{34}, \text{update}, R_{34,35,17,18}, R_{36,37,15,18} &gt;$</td>
</tr>
<tr>
<td>RT</td>
<td>$&lt; T_{39,41}, \text{update}, R_{39,40,36,38} &gt;$</td>
</tr>
<tr>
<td>RRT</td>
<td>$&lt; T_{39,41}, \text{update}, R_{39,40,36,38}, R_{41,42,36,38} &gt;$</td>
</tr>
<tr>
<td>RRR</td>
<td>$&lt; R_{43,43,32,33}, \text{update}, R_{32,33,27,28}, R_{43,43,25,26} &gt;$</td>
</tr>
</tbody>
</table>

Table 5.2: Examples of update tasks.

5.4.3 Block sparse Cholesky factorization

In Figure 5.11, we present a modified sparse Cholesky factorization algorithm that uses block partitions and the tasks defined on these blocks. We refer to this algorithm as block sparse Cholesky factorization. In this algorithm, $Q$ is the number of clusters in $L$ as identified by the pre-partitioner, where each cluster is at least $W$ columns wide. For each cluster, computations are performed on each unit block in that cluster. For
Figure 5.10: Block update tasks.
\begin{figure}
procedure Block\_Sparse\_Cholesky

\textbf{for} $C = 1$ to $Q$ \textbf{do}
  \begin{itemize}
    \item \textbf{foreach} block $B$ in $C$ \textbf{do}
      \begin{itemize}
        \item $U_B \leftarrow$ list of updaters of $B$
        \item \textbf{foreach} $(B', B'') \in U_B$ \textbf{do}
          \begin{itemize}
            \item $< B, Update, B' >$
          \end{itemize}
        \end{itemize}
    \end{itemize}
    \begin{itemize}
      \item \textbf{foreach} $(B', B'') \in U_B$ \textbf{do}
        \begin{itemize}
          \item $< B, Update, B', B'' >$
        \end{itemize}
    \end{itemize}
  \end{itemize}
  \begin{itemize}
    \item \textbf{if} (B is not a rectangle) then
      \begin{itemize}
        \item $< B, Scale >$
      \end{itemize}
    \end{itemize}
\end{figure}

Each block, the list of updating unit blocks is extracted from the BDAG. This list of updating blocks is divided into two sets: one set containing single blocks that update $B$, and the other set containing pairs of blocks that update $B$.

In order to understand the block computations, it is necessary to get a more formal view of the dependencies that bring about the twelve different types of tasks. The next subsection gives the basis for computing these inter-block dependencies, using the element-level dependencies in Cholesky factorization as a building block.

### 5.4.4 Block dependencies

The partitioner establishes the inter-block dependencies by building on the element-level dependencies shown in Figure 2.3. These inter-block dependencies can be classified according to the shape and type of the unit blocks involved. We describe how these dependencies are constructed, using the idea of projection, and the intersection of projections.

For the purpose of illustration, we use the example shown in Figure 5.12. In that figure, $A$, $B$ and $C$ are clusters. Cluster $A$ has two rectangular blocks, $P_1$ and $M_1$, cluster $B$ has two rectangular blocks, $P_2$ and $M_2$. The triangular block of cluster $C$
Figure 5.12: Inter-block dependencies in sparse Cholesky factorization.
is partitioned into six unit blocks; three triangular units and three rectangular units. The off-diagonal rectangular blocks in $C$ are each partitioned into three unit blocks; the upper rectangle is partitioned into $R_1$, $R_2$ and $R_3$ and the lower rectangle is partitioned into $R_4$, $R_5$ and $R_6$.

**Projection**

**Definition 5.4.1:** The *horizontal projection* of a rectangular block $R_{r_1,r_2,c_1,c_2}$, formally denoted as $HP(R)$, is the interval $[r_1,r_2]$, obtained by extending the top and bottom edges of $R_{r_1,r_2,c_1,c_2}$ to the diagonal. The horizontal projection of a triangular block $T_{m,n}$ is $[m,n]$. This is denoted as $HP(T)$.

**Definition 5.4.2:** The *vertical projection* of a rectangular block $R_{r_1,r_2,c_1,c_2}$, formally denoted as $VP(R)$, is the interval $[c_1,c_2]$, obtained by extending the left and right edges of $R_{r_1,r_2,c_1,c_2}$ to the diagonal. The vertical projection of a triangular block $T_{m,n}$ is $[m,n]$ and is denoted as $VP(T)$.

A single point $P : (i,j)$ is defined to be the degenerate rectangle $P_{i,i:j}$ if $i \neq j$; otherwise, it is defined to be the degenerate triangle $T_{i,i}$. In the first case, $HP(P)$ is $[i,i]$ and $VP(P)$ is $[j,j]$. In the second case, $HP(P)$ and $VP(P)$ are both equal to $[i,i]$. In both cases the projections are degenerate zero-length intervals.

In Figure 5.12, $HP(P_1) = [I_1, I'_1]$, $HP(P_2) = [I_2, c1]$ and $VP(R_2) = [c2, c3]$.

Projections are used by the partitioner in order to compute the inter-block dependencies. When a cluster is partitioned, all the rectangles in that cluster are horizontally projected to the diagonal, just before they are partitioned into unit blocks. The horizontal projection of each rectangle finds a *match* with one or more clusters. A cluster $C_{m,n}$ matches a horizontal projection if the projection interval overlaps the interval $[m,n]$. For example, in Figure 5.12, the cluster $C$ matches the projection $HP(P_1)$, and the matching interval is $[c1, I'_1]$. When a rectangle is projected, the matching interval, along with the projecting rectangle's row and column extents, is stored in the matching cluster(s). When a single-column cluster is processed, each of its non-zeros is
horizontally projected. The row number of the non-zero element along with the column number is stored in the matching cluster(s).

**Characterization of dependencies**

To characterize all the inter-block dependencies, we start by giving two basic definitions that are extensions of the element-level operations presented in Section 2.2. Note that all elements of $L$ referred to in the following definitions are nonzeros.

**Definition 5.4.3**: Block $B$ is updated by a base block $B'$ and a multiplier block $B''$ iff $\exists i, j, k$ such that $L_{i,j} \in B \land L_{j,k} \in B' \land L_{i,k} \in B''$. The update operation is referred to as block base-multiplier update. We say that $B$ depends on $B'$ and $B''$, denoted as $B', B'' \delta B$.

**Definition 5.4.4**: Block $B$ is updated by a base block $B'$ only iff $\exists i, j$ such that $L_{i,j} \in B' \land L_{i,j} \in B$. The update operation is referred to as block base update. We say that $B$ depends on $B'$, denoted as $B' \delta B$.

Note that a block base update could, in general, include element-level base-multiplier updates.

We use Definition 5.4.3 and Definition 5.4.4 and the following two definitions to characterize the different generic task dependency types in block sparse Cholesky factorization.

**Definition 5.4.5**: If $B' \delta B$ then $\delta_{B',B}$ denotes the portion of $B'$ that is transferred from $B'$ to $B$, i.e., $\delta_{B',B} = (r, r', c, c')$ means that the sub-part of $B'$ enclosed in the row extent $r - r'$ and column extent $c - c'$ is used to update $B$.

**Definition 5.4.6**: If $B', B'' \delta B$ then $\delta_{B',B'',B}$ gives the portions of $B'$ and $B''$ that are used to update $B$, i.e., $\delta_{B',B'',B} = (r, r', i, i', c, c')$ means that that the sub-part of $B'$ enclosed in the row extent $r - r'$ and column extent $c - c'$ is combined with the sub-part of $B''$ enclosed in the row extent $i - i'$ and column extent $c - c'$ to update $B$. 
Each inter-block dependency results in a task, and will be denoted by using the tuple notation for tasks, \( <P, \text{Oper}, |P|, |P| > \) introduced in Section 5.4.2. The following two rules are used to determine inter-block dependencies or updates.

1. **Rule 1: Base update** Block \( B \) is updated by a base block \( A \) iff \( VP(B) = HP(B) \) and \( HP(A) \cap HP(B) \neq \emptyset \).

   By applying this rule, we obtain the following generic dependencies based on block shapes. In each case, the corresponding task in the form of a tuple is also indicated. The block dependencies are illustrated in Figure 5.10.

- **Column updates column, \(<\text{Col}, \text{Update}, \text{Col}>\)**
  Let \( A \) be column \( k \) and \( B \) be column \( j \). For \( A \delta B \) to be true, \( \exists (P_1 \in A \land P_2 \in B) \) such that \( VP(P_2) = HP(P_2) \) and \( HP(P_1) = HP(P_2) \). This is only satisfied for \( P_1 = L_{j,k} \) and \( P_2 = L_{j,j} \). This is the base case as seen in Figure 2.3. Also, \( \delta_{A,B} = (j, N, k, k) \), i.e., all non-zeros below and including the non-zero in row \( j \) of column \( k \) are used in the update.

- **Column updates triangle, \(<\text{Tri}, \text{Update}, \text{Col}>\)**
  Let block \( A \) be column \( k \) and block \( B \) be triangle \( T_{r,r} \). By definition, \( VP(T) = HP(T) = [r, r'] \). For \( A \delta B \) to be true, \( \exists (P_1 \in A) \) such that \( HP(P_1) \cap HP(T) \neq \emptyset \). This is only satisfied for any \( P_1 = L_{i,k}, r \leq i \leq r' \). Also, \( \delta_{A,B} = (i, r', k, k) \), i.e., all non-zeros in column \( k \) from row \( i \) to row \( r' \) are used in the update. In Figure 5.10(a), the non-zero elements of column \( k \) that are involved in the update are in rows \( i_1, i_2 \) and \( i_3 \). The points of intersection of the dotted lines with each other and of the dotted lines with the diagonal are the points of triangle \( T \) that are updated by column \( k \).

- **Rectangle updates column, \(<\text{Col}, \text{Update}, \text{Rect}>\)**
  Let block \( A \) be rectangle \( R_{r,r',c,c'} \) and block \( B \) be column \( k \). For \( A \delta B \) to be true, \( \exists (P \in B) \) such that \( HP(P) \cap HP(R) \neq \emptyset \). This is only satisfied for \( P = L_{k,k}, r \leq k \leq r' \). Then, \( \delta_{A,B} = (k, r', c, c') \), i.e., the sub-part of the rectangle enclosed between rows \( k \) and \( r' \) is used to update the column. In Figure 5.10(e),
the shaded portion of the rectangle between rows \( k \) and \( r_2 \) update the column elements between rows \( k \) and \( r_2 \).

- **Rectangle updates triangle, \(< Tri, Update, Rect >\)**

  Let block \( A \) be rectangle \( R_{r,r',c,c'} \) and block \( B \) be triangle \( T_{m,n} \). By definition, 
  \[ VP(T) = HP(T) = [m, n] \]. For \( A \neq B \) to be true, then, \( HP(A) \cap HP(B) \neq \emptyset \).
  This is satisfied if and only if \([r, r'] \cap [m, n] \neq \emptyset \). Let this intersection be \([i, i']\), then \( \delta_{A,B} = (i, i', c, c') \). In Figure 5.10(g), the shaded portion of \( R \) updates the shaded portion of \( T \).

2. **Rule 2: Base-multiplier update**

  Block \( C \) is updated by blocks \( A \) and \( B \) \((A, B \neq C)\) iff \( HP(A) \cap VP(C) \neq \emptyset \) and 
  \( VP(A) \cap VP(B) \neq \emptyset \) and \( HP(B) \cap HP(C) \neq \emptyset \).

  The following block dependencies are obtained by applying this rule:

- **Column updates rectangle, \(< Rect, Update, Col >\)**

  Let \( A \) and \( B \) be column \( k \) and \( C \) be rectangle \( R_{r,r',c,c'} \). For \( A, B \neq C \) to be true, 
  \( \exists (P_1 \in A \land P_2 \in A) \) such that \( HP(P_1) \cap VP(C) \neq \emptyset \) and \( HP(P_2) \cap HP(C) \neq \emptyset \). This is only satisfied for \( P_1 = L_{i,j,k}, P_2 = L_{i,k}, c \leq j \leq c' \) and \( r \leq i \leq r' \). Then, 
  \( \delta_{A,B,C} = (c, c', r, r', k, k) \). In Figure 5.10(b), the non-zero elements in rows \( i_1 \) and \( i_2 \) of column \( k \) combine with the non-zero elements in rows \( j_1, j_2 \) and \( j_3 \) to update a portion of \( R \). This updated portion is the set of points given by the intersection of the dotted lines in \( R \)'s interior. This update can be treated as a base update for practical purposes, since both the base and the multiplier are the same block.

- **Triangle and rectangle update rectangle, \(< Rect, Update, Tri, Rect >\)**

  Let block \( A \) be triangle \( T_{m,n} \), block \( B \) be rectangle \( R_{r,r',c,c'} \) and block \( C \) be rectangle \( R_{i,i',j,j'} \). For \( A, B \neq C \) to be true, 
  \([m, n] \cap [j, j'] = [v, v'] \neq \emptyset \) and 
  \([m, n] \cap [c, c'] = [u, u'] \neq \emptyset \) and \([r, r'] \cap [i, i'] = [t, t'] \neq \emptyset \). Then, \( \delta_{A,B,C} = (v, v', t, t', u, u') \).

  In Figure 5.10(d), the shaded rectangular portion of \( T \) combines with the entire shaded rectangle \( R_1 \) to update the entire shaded rectangle \( R_2 \).
- Two rectangles update column, <Col, Update, Rect, Rect>

Let block $A$ be rectangle $R_{r',r',c',c'}$, block $B$ be rectangle $R_{i,i',j,j'}$, and block $C$ be column $k$. For $A, B \delta C$ to be true, $[r, r'] \cap [k, k] \neq \emptyset$, i.e., $r \leq k \leq r'$; and $[c, c'] \cap [j, j'] = [u, u'] \neq \emptyset$; and $\exists L_{q,k}$ such that $[i, i'] \cap [q, q] \neq \emptyset$, i.e., $i \leq q \leq i'$. The fill-in property guarantees that if $[r, r'] \cap [k, k] \neq \emptyset$ then all the elements $L_{q,k}, i \leq q \leq i'$ are non-zero. Thus $\delta_{A,B,C} = (k, r', i, i', u, u')$. In Figure 5.10(f), the elements of $R_1$ which are in the row $k$ between the vertical dotted lines combine with the entire shaded rectangle $R_2$ to update the elements between rows $r_3$ and $r_4$ in column $k$.

- Two rectangles update triangle, <Tri, Update, Rect, Rect>

Let block $A$ be rectangle $R_{r,r',c,c'}$, block $B$ be rectangle $R_{i,i',j,j'}$, and block $C$ be triangle $T_{m,n}$. For $A, B \delta C$ to be true, $[r, r'] \cap [m, n] = [v, v'] \neq \emptyset$; $[c, c'] \cap [j, j'] = [u, u'] \neq \emptyset$; and $[i, i'] \cap [m, n] = [t, t'] \neq \emptyset$. Then, $\delta_{A,B,C} = (v, v', t, t', u, u')$. In Figure 5.10(h), the shaded portion of $R_1$ combines with the entire shaded rectangle $R_2$ to update the shaded rectangular portion of $T$.

- Two rectangle update rectangle, <Rect, Update, Rect, Rect>

Let block $A$ be rectangle $R_{r,r',c,c'}$, block $B$ be rectangle $R_{i,i',j,j'}$, and block $C$ be rectangle $R_{p,p',q,q'}$. For $A, B \delta C$ to be true, $[r, r'] \cap [q, q'] = [v, v'] \neq \emptyset$ and $[c, c'] \cap [j, j'] = [u, u'] \neq \emptyset$ and $[i, i'] \cap [p, p'] = [t, t'] \neq \emptyset$. Then, $\delta_{A,B,C} = (v, v', t, t', u, u')$. In Figure 5.10(i), the shaded portion of $R_1$ combines with the shaded portion of $R_2$ to update the shaded part of $R_3$.

- Triangle updates rectangle, <Rect, Update, Tri>

Let block $A$ be triangle $T_{m,n}$ and blocks $B$ and $C$ be rectangle $R_{r,r',c,c'}$. Then for $A, B \delta C$ to be true, $[m, n] \cap [c, c'] = [v, v'] \neq \emptyset$. This implies that triangle and rectangle must belong to the same cluster and hence $[v, v'] = [c, n]$. Also, $[m, n] \cap [c, c'] = [u, u'] \neq \emptyset$, which by the same argument as above implies that $[u, u'] = [c, n]$. Finally, $HP(B) \cap HP(C) \neq \emptyset$ is trivially true by virtue of $B$ and $C$ being the same rectangle. $HP(B) = [r, r']$. We thus have $\delta_{A,B,C} = (c, n, r, r', c, n)$. In fact this can be effectively treated as a base update.
In Figure 5.10(c), the shaded portion of $T$ updates the shaded portion of $R$.

Referring to Figure 5.12, suppose we want to compute the dependencies for the triangular block $T : (c_1, c_2)$. $HP(P_1) = [I_1, I'_1]$ and $VP(T) = [c_1, c_2]$. $HP(P_1) \cap VP(T) = [c_1, c_2]$ and therefore, the base block $P_1$ updates the block $T$. The intersection also gives complete information about the portions of all the blocks involved in each update. In above case, the upper shaded extent of $P_1$ between rows $c_1$ and $c_2$ updates all of $T$. Now consider block $R_2$. $VP(R_2) = [c_2, c_3]$ and $HP(P_1) = [I_1, I'_1]$, hence $VP(R_2) \cap HP(P_1) = [c_2, I'_1]$. $VP(M_1)$ matches $VP(P_1)$. $HP(M_1)$ intersects $HP(R_2)$. Thus, the lower shaded portion of the base $P_1$, enclosed between rows $c_2$ and $I'_1$, and all of the multiplier $M_1$ are used in updating the shaded portion of $R_2$. Figure 5.12 also shows $R_5$ being updated by the combination of the upper shaded portion of $P_2$ and the whole of $M_2$ and $R_6$ being updated by the lower shaded portion of $P_2$ and the whole of $M_2$.

5.4.5 Block structure

The shape and size of the dense blocks extracted by the partitioner are influenced by several factors, including the sparsity structure of the matrix, the parallelism in the problem, and the characteristics of the target architecture. We discuss how these factors influence the design of the partitioner.

We choose the dense blocks to be either triangular or rectangular in shape. The zero-nonzero structure of the symbolic factor of most sparse matrices shows naturally occurring triangular blocks at the diagonal and polygonal blocks off the diagonal. Manipulating polygonal blocks is complex and makes the coding process extremely difficult. Apart from this, computing the inter-block dependencies is much more difficult than with rectangular blocks. Thus, for the sake of simplicity, we restrict unit blocks to "regular" triangular and rectangular shapes. A unit triangle in our partitions is an equilateral triangle. The triangular blocks at the diagonals are partitioned into dense equilateral unit triangles and dense unit rectangles. The off-diagonal polygonal blocks are partitioned into dense unit rectangles. The pre-partitioner does some of this work
by carrying out the rectangulation of these polygons.

The sizes of the unit blocks are determined by the \textit{grain size}, which is fixed by the parameter $G$. The value of $G$ specifies the minimum block size in partitioning the triangle and rectangles in a cluster. The value of $G$ is based on two factors: the computation to communication ratio of the target architecture and the need to load balance the computations across processors. $G$ determines the size of a block on the basis of either the number of nonzeros in that block and/or the amount of computational work required to factor that block. Typically, the computations associated with nonzeros in the factor increase towards the right of the matrix, whereas the elements towards the left tend to be required in many more computations than those towards the right. Thus, in partitioning certain regions of the matrix, communication overhead must be given higher priority and in some other regions load imbalance overhead must be given higher priority. In the communication overhead prone regions, $G$ is used to partition the matrix so that the locality is improved and in the load imbalance prone regions, $G$ is used to gain more flexibility in load balancing. In the former case, the parameter $G$ is used in determining the minimum number of nonzero elements per block and in the latter case, it is used in determining the amount of computational work per block. In its two forms, the value of $G$ is scaled appropriately to take into account the architectural characteristics.

A third factor, the \textit{data access} within blocks, is used by the partitioner to further refine the partitioning process. Data access is the order in which the individual elements of data in a block are accessed during computation. Uniprocessor computational performance on any machine depends on two major factors - the locality of computations and the vector lengths used. Increased locality leads to improved cache reuse and better performance. The locality of data is dictated by the size and shape of the blocks. Large vector lengths generally result in better performance, due to enhanced utilization of vector processors and/or pipelined floating point units. The vector lengths are also dictated by the size and the shape of the blocks, and also the data access used during the computations. Given a certain amount of computation to be performed, few iterations with long vectors result in better performance than several iterations with short
vectors, if the vectors are no longer than can be fitted into the cache. As far as possible, data in the blocks should be organized in such a manner that the computational tasks satisfy the former condition.

We consider two forms of data access in computations: column-major and row-major. In the former case, the data values within a unit triangular or unit rectangular block are stored in column-major order, while in the latter, they are stored in row-major order. We now compare these methods using the computations involved in Cholesky factorization, in order to highlight the tradeoffs that need to be considered by the partitioner in preferring one method over the other.

The basic element-level update in Cholesky factorization is of the form $y \leftarrow y - \alpha x$. A basic block update computation which illustrates both column-major access and row-major access in Cholesky factorization is shown in the Figure 5.13. The block $M$ and the vector $x$ are in the same column range, and the block $M$ and the vector $y$ are in the same row range. $M$, $x$ and $y$ are all dense. The computation is a sequence of the following element-level updates, given by:

$$y_i \leftarrow y_i - M_{i,j} \cdot x_j, \quad i \in [r, r'], \quad j \in [c, c']$$

In Figure 5.13, the shaded portion of $M$ shows the part that has been already used in computing $y$. In (a), the progression is from left to right, column-wise, and for every column of $M$ used up, one element of $x$ is also used and the vector $y$ is updated once. In (b), the progression of computation is from top to bottom, row-wise, and for
every row of $\mathbf{M}$ used up, the vector $\mathbf{x}$ is used once, and one element of $\mathbf{y}$ is computed.

The code for column-major access is given in Figure 5.14(a). After loop-interchange, this results in the code for row-major access, shown in Figure 5.14(b). The right-hand

\[
\begin{align*}
\text{for } j &= c \text{ to } c' \text{ do} \\
& \text{for } i = r \text{ to } r' \text{ do} \\
& \quad y_i \leftarrow y_i - M_{i,j} \times x_j \\
& \text{endfor} \\
\text{endfor} \\
\text{for } j &= c \text{ to } c' \text{ do} \\
& \text{for } i = r \text{ to } r' \text{ do} \\
& \quad y_i \leftarrow y_i - M_{i,j} \times x_j \\
& \text{endfor} \\
\text{endfor}
\end{align*}
\]

(a) Column-major element and vector updates

\[
\begin{align*}
\text{for } i &= r \text{ to } r' \text{ do} \\
& \text{for } j = c \text{ to } c' \text{ do} \\
& \quad y_i \leftarrow y_i - M_{i,j} \times x_j \\
& \text{endfor} \\
\text{endfor} \\
\text{for } i &= r \text{ to } r' \text{ do} \\
& \text{for } j = c \text{ to } c' \text{ do} \\
& \quad y_i \leftarrow y_i - M_{i,j} \times x_j \\
& \text{endfor} \\
\text{endfor}
\end{align*}
\]

(b) Row-major element and vector updates

Figure 5.14: Column-major and row-major major access.

side of Figure 5.14(a) is an $\text{axpy}$ enclosed in the $j$ loop, while the right-hand side of Figure 5.14(b) is a $\text{dot}$ product enclosed in the $i$ loop.

The decision of whether to use $\text{axpy}$ or dot product would depend on the relative efficiencies of $\text{axpy}$ and dot product operations on the machine, and the structure of the blocks under consideration. If we assume that given a vector length, the $\text{axpy}$ and dot operations execute with the same efficiency, then the decision depends purely on the block structure. It would be better to use row-major access and storage if there is a relatively large number of horizontally long and flat blocks in the partitioning. On the other hand, if there is a relatively large number of vertically tall and thin blocks, then it would be better to use column-major access. Clearly, the number of blocks of either type depends on the sparsity structure of the matrix. In all the test matrices that we have studied, the sparsity structure is more suited for row-major access. We have, therefore, chosen to implement the partitioner so that the rectangular block partitions
are horizontally long and flat, instead of square in shape.

To clarify further, consider the matrix of Figure 5.9. Let the grain size \( G \) be set to 4, and assume that all blocks are partitioned on the basis of number of non-zeros i.e. each block is required to contain at least 4 non-zeros. In that matrix, the rectangle \( R_{43,49,20,23} \) is partitioned into seven horizontally flat unit rectangles, while rectangle \( R_{32,33,25,28} \) is partitioned into two squares. Consider the update of \( T_{32,33} \) using the rectangles obtained by partitioning \( R_{32,33,25,28} \). If we partition the rectangle into square units, we get vector lengths of size 2, whether we use axpy or dot product. If we partition the rectangle into two units which are horizontally flat and use dot product, we get vectors of length 4. In this case, it is better to partition horizontally.

To summarize, the block partitions are sparse columns or dense triangles or unit rectangles. The sizes of the rectangles and triangles are determined by the grain size parameters \( G \). The rectangles are horizontally long and flat, in preference to a square shape. All data within triangles and rectangles are stored in row-major order.

### 5.4.6 Code for task computations

The code for the different types of tasks, based on the row-major access for dense blocks, is summarized in the following discussion. We do not give the code for \( CC \), \( CT \) and \( CR \) since these are essentially \( cmod \) tasks. In all the other cases, we assume that the base block \( B \), the multiplier block \( M \) and the updated block \( D \) have been identified as the appropriate sub-portions of the blocks in consideration. Let the row extent of \( B \) be \( h \) to \( h' \). This is the same as the column extent of \( D \). Let the row extent of \( M \) be \( r \) to \( r' \), which is the same as the row extent of \( D \). Let the column extent of \( M \) be \( w \) to \( w' \), which is the same as the column extent of \( B \). We refer to the row-major update code in Figure 5.13(b) as \( \text{upd} \), a procedure which will accept vectors \( x \) and \( y \) and a block \( M \) as parameters. The code for \( RRR \), \( TRR \), \( RRT \), \( RC \) and \( RRC \) are essentially the same, and is shown in Figure 5.15. For both \( RC \) and \( RRC \), \( h = h' \). In addition, for \( RC \), \( M = B \).

The code for \( RT \) is shown in Figure 5.16. \( B = M \) for this task.

The code for \( TR \) is shown in Figure 5.17. In this task, \( M = D \), and the calls to \( \text{upd} \)
for $s = h$ to $h'$ do
  $x \leftarrow B_{s,s',w,w'}$
  $y \leftarrow D_{r',r,s,s}$
  call $upd(x, y, M_{r',r,w,w'})$
endfor

Figure 5.15: Code for RRR, RRT, TRR, RRC and RC.

for $s = r$ to $r'$ do
  $x \leftarrow B_{s,s,w,w'}$
  $y \leftarrow D_{s,h,s,s}$
  call $upd(x, y, B_{s+1,r',w,w'})$
endfor

for $s = h$ to $h'$ do
  $x \leftarrow B_{s,s,h,s-1}$
  $y \leftarrow D_{1,r,s,s}$
  call $upd(x, y, D_{1,r,1,s-1})$
  $scale(D_{r,r',s,s})$
endfor

Figure 5.16: Code for RT. Figure 5.17: Code for TR.

are interspersed with the scaling of the columns of $D$. Each column of $D$ is scaled by dividing the elements of the column by the diagonal element in that column, which is in the triangular block $B$.

5.4.7 BCR for block sparse Cholesky factorization

We describe how the BCR representation works for our block partitioning scheme.

**AVR for blocks**

The representation for sparse columns is identical to the description for CSC storage in Section 3.2.1. For a dense triangle $T_{m,n}$, the attributes are $(m, n)$, the data is stored in contiguous fashion, and the size of the triangle is $(n - m + 1) \times (n - m + 2) \times 0.5$.

For a dense rectangle $R_{r1,r2,c1,c2}$, the attributes are $(r1, r2, c1, c2)$, the data is stored in contiguous fashion, and the size of the rectangle is $(r2 - r1 + 1) \times (c2 - c1 + 1)$.

**Representation for tasks**

The correspondence between the notation used here and the general BCR representation is as follows.

- $T = CC, CT, CR, TR, TRR, RC, RRC, RT, RRT, RRR, SC, ST$
• A task of type $<B,\text{Oper},B'>$ corresponds to, say, $<\mathcal{R}^*_u,\mathcal{R}^*_{u,1}>$, and a task of type $<B,\text{Oper},B',B''>$ corresponds to, say, $<\mathcal{R}^j_u,\mathcal{R}^j_{u,1},\mathcal{R}^j_{u,2}>$. In the first case, $\mathcal{R}^*_{u,1}$ corresponds to $\delta_{B',B}$; $\Gamma(t^*_u)$, the program module for this task, is implicitly defined by the partition shapes of $B$ and $B'$, and by $\text{Oper}$. For example, if $B$ is a triangle and $B'$ is a rectangle, and $\text{Oper}$ is update, then $\Gamma(t^*_u)$ is the module RT. In the second case, $\mathcal{R}^j_{u,1}$ and $\mathcal{R}^j_{u,2}$ together correspond to $\delta_{B',B'',B}$, and $\Gamma(t^j_u)$ is implicitly defined by the partition shapes of $B$, $B'$ and $B''$, and by $\text{Oper}$. For example, if $B$ is a triangle, $B'$ and $B''$ are rectangles, and $\text{Oper}$ is update, then $\Gamma(t^j_u)$ is the module RRT. In either case, the portion of the updated block which is affected by the update, $R'_u$ or $R'_u$, respectively, is determinable from the portions of the updating blocks and the type of task. It need not be explicitly represented.

The actual computational code for each of the program modules is discussed in Section 5.4.6.

**BDAG for precedence relationships**

Let us consider the precedence relationships of each type of block in turn. Let the block be labeled as $B_k$. If the block is a sparse column, the updates to it can be executed in any order. The only restriction is that the scaling task SC must be executed only after all other updates are complete. Let there be $r$ updates to $B_k$ and let the scaling task be identified as the $r$-th update. Then, in the BDAG, the vertex set of the subgraph corresponding to $B_k$, say $V_k$, consists of vertices $n^k_i, \ 1 \leq i \leq r$, and the vertex $n^k_{r+1}$. The set of edges coming into any of the vertices in $V_k$ includes, first of all, all the updates to $B_k$ except SC. These are all type I edges. Apart from these, there is an edge $(n^k_{r+1}, n^k_i)$ of type I, for the SC task, and the edges $(n^k_i, n^k_j), \ 1 \leq i \leq r - 1$ of type II, expressing the constraint stated above. If the block is a dense triangle, the precedence relationships are similar. The updates to the triangle can be executed in any order, so long as the task ST is executed only after all other updates are complete. The BDAG sub-graph for the triangle is built in a manner analogous to that for a column.

If the block is a dense rectangle, the same precedence relationship holds - the scaling task is performed only after all the other updates are done. However, recall that the
scaling of a rectangle, which is the final step in its factoring, is done as a part of
the TR tasks that update that rectangle. This leads to more precedence constraints.
We explain this using the example of Figure 5.9. Consider the cluster \( C_{30,33} \) and the
rectangle \( R_{30,30,33} \) in it. For simplicity, we will abbreviate this rectangle as \( R_u \). Again,
assume there are \( r \) updates to \( R_u \), which include the three updates due to the triangular
block above it. Let these updates be labeled by \( C_u^{r-2} < R_u, updates, T_{30,31} >, C_u^{r-1} < R_u, updates, T_{32,33} >, \) and \( C_u^r : < R_u, updates, R_{32,33,30,31}, R_u > \). In \( C_u^r \), the left half of
\( R_u \) is used in updating the right half. But this may only be done if the left half has been
factored. In turn, factoring the left half of \( R_u \) is completed only when \( C_u^1 \) is completed.
Finally, the right half of \( R_u \) is factored in the course of the execution of \( C_u^{r-1} \). These
constraints set up a total precedence order among these tasks: \( C_u^{r-2} < C_u^r < C_u^{r-1} \). The
three tasks listed above are responsible for scaling the rectangle \( R_u \). Note that each
TR task is, in effect, a TRR task in which the updating rectangle is the same as the
updated rectangle. This leads to the following additional dependencies: \( C_u^{r-2} < C_u^{r-2} \)
and \( C_u^{r-1} < C_u^{r-1} \). In the BDAG, the vertex set of the sub-graph corresponding to
\( R_u \), say \( V_u \), consists of vertices \( n_u^i, 1 \leq i \leq r \), and the vertices \( n_u^{r-2} \) and \( n_u^{r-1} \). The
set of edges coming into any of the vertices in \( V_u \) includes, first of all, all the updates
to \( R_u \) except the three updates listed above. These are all type I edges. There are
two type II edges: \( (n_u^{r-2}, n_u^i) \) and \( (n_u^{r-1}, n_u^i) \). Apart from these there are the edges
\( (n_u^i, n_u^i), 1 \leq i \leq r - 3 \), \( (n_u^i, n_u^{r-1}), 1 \leq i \leq r - 3 \), and \( (n_u^i, n_u^{r-2}), 1 \leq i \leq r - 3 \)
expressing the precedence constraint that the tasks \( C_u^{r-2}, C_u^{r-1} \) and \( C_u^r \) must execute
only after all the other updates to \( R_u \) are done. Lastly, there are the type I edges
\( (n_u^{r-1}, n_u^{r-1}) \) and \( (n_u^{r-2}, n_u^{r-2}) \) which are, respectively, the tasks \( C_u^{r-1} \) and \( C_u^{r-2} \).

The precedence relationships may be generalized as follows. Let \( R_u \) be a unit rect-
angle in cluster \( C_{m,n} \). For ease of understanding, we assume that the rectangle extends
across the entire width of the cluster. If it does not, we consider only that portion of
the triangle above the rectangle, whose left edge is in the same column as the left edge of
the rectangle. (See the characterization of the TR task in 5.4.4). Let the triangle \( T_{m,n} \)
be partitioned into unit triangles and rectangles numbered as \( B_{i,j}, i \geq j, 1 \leq i, j \leq k \).
There are \( k \) rows and \( k \) columns of unit blocks in \( T_{m,n} \). The numbering is such that
block \( B_{r,j} \) is above \( B_{s,j} \), in the same column of blocks, if and only if \( r < s \) and similarly, block \( B_{i,p} \) is to the left of \( B_{i,q} \), in the same row of blocks, if and only if \( p < q \). The tasks \( < R_s, \text{update}, B_{r,j} >, 1 \leq k \) are the TR tasks and the tasks \( < R_s, \text{update}, B_{i,j}, R_u > \) are the RRR tasks. We denote the former set of tasks by \( T_i,i \) and the latter by \( T_i,j, i > j \). The partial order of precedence among these tasks is \( T_{i,j} \prec T_{i,i} \prec T_{k,i} \). The BDAG is built for this general case by simple extension of the specific instance discussed in the preceding paragraph.

### 5.5 Partitioner

Figure 5.18 gives a schematic view of the partitioner. The lower triangular symbolic

---

![Partitioner schematic](image_url)

---

Figure 5.18: Partitioner schematic.

factor, with clusters and columns identified by the pre-partitioner, is the input to the partitioner. The partitioner sweeps left to right across this input matrix, processing the columns and clusters as they are encountered in left to right order. When a column
is processed, the non-zeros in the column are projected to the diagonal and stored in the matching clusters and columns. Next, the dependencies for the column due to blocks to the left of the column, are computed. The dependencies are computed using the previously stored projections due to the columns and rectangles to the left of the column being processed. When a cluster is processed, the off-diagonal rectangular blocks are projected to the diagonal and stored in the matching clusters and columns. The triangles and rectangles are then partitioned into unit blocks. The partitioner uses two variants of $G$. The variant $T$ is the grain size used to partition the triangular block into unit blocks and the variant $R$ is used to partition each off-diagonal rectangular block into unit rectangles. The dependencies for each of unit block in the cluster are then computed. This is done by first building a data structure called the interval tree. The tree is used to store the projections from the columns and rectangles that are to the left of the cluster. Dependencies for the unit blocks are computed efficiently by making queries on the interval tree. Details of the entire process are described in the rest of this section. The computed inter-block dependencies are stored in the BDAG.

5.5.1 Triangle and rectangle partitioning

Let the width of the triangle in a cluster be $w$. Note that $w$ is both the height and the base of the triangle. The partitioner divides the triangle into, say, $k$ horizontal and vertical strips. In Figure 5.19(a), $k = 4$. All the strips are of the same width except,
possibly, the bottom-most horizontal strip and the rightmost vertical strip which may be smaller in width. So all rectangular unit blocks are of the same size and all triangular unit blocks are of the same size, except, possibly, those at the “edges”. The triangle is thus partitioned into $k(k+1)/2$ unit blocks: $k$ triangular unit blocks and $k(k-1)/2$ rectangular unit blocks. The size of each partition (and hence $k$) is controlled by the grain size parameter $T$. The parameter $T$ imposes a lower bound on the block size and hence an upper bound on the number of partitions. Let $T$ allow at most $t$ parts in the given triangle. Then the partitioner will choose the largest $k$ for which $k(k+1)/2$ is at most $t$. Procedure Partition_triangle in Figure 5.20 shows how a triangle of width $w$ is partitioned into at most $t$ parts.

\begin{verbatim}
procedure Partition_triangle
1. $k \leftarrow (\sqrt{1+8*t} - 1)/2$
2. $v \leftarrow w/k$
3. if $w$ is not an integer multiple of $k$ then $v \leftarrow v + 1$ endif
4. $k' \leftarrow w/v$
5. if $w$ is not a multiple of $v$ then $k' \leftarrow k' + 1$ endif
\end{verbatim}

Figure 5.20: Procedure for partitioning of a triangle.

All divisions in that procedure are integer divisions. Step 1 computes the maximum number of horizontal and vertical strips given that there can be at most $t$ partitions. In step 2, $v$, the size of a strip, is initialized to $w/k$. If $w$ is not an integer multiple of $k$, $k-1$ strips are of width $v$ each and one strip is of width $v + (w \mod k)$. Steps 3, 4 and 5 ensure that the width of the “edge” strip is less than that of the others and the number of strips, $k' \leq k$.

Now consider a rectangle of size $m \times n$, where $m$ is the height and $n$ is the breadth. Let the parameter $R$ be such that a maximum of $r$ partitions are to be made for this rectangle.

**Square rectangles**

We first give the code for the situation in which the rectangle is to be divided into units that are approximately square in shape. The rectangle is divided into $p$ horizontal
strips and \( q \) vertical strips. All the horizontal strips are of equal height except, possibly, the bottom-most one and all the vertical strips are of equal breadth except, possibly, the rightmost vertical strip. These "edge" strips may be narrower than the rest. Thus all unit blocks are of the same size except, possibly, at the edges. Let \( p \times q \) be equal to \( k \). The partitioner chooses the largest \( k \) which is at most equal to \( r \). The factors \( p \) and \( q \) are chosen in such that each unit block is approximately square in shape. This ensures high degree of locality for a given amount of work. For example, in Figure 5.19(b), \( p \) is equal to three and \( q \) is five. Procedure \textit{Partition\_rectangle} in Figure 5.21 describes the simple heuristic used to achieve this. Let \( h \) be the height of a horizontal partition and \( v \) be the breadth of a vertical partition. The algorithm will choose a \( h \) and a \( v \) for which \( |h - v| \) is minimum. At the end of the process, \textit{bestv} and \textit{besth} give the vertical partition breadth and the horizontal partition height respectively.

The algorithm proceeds by picking possible factors \( i \) and \( j \) as follows. The factor \( i \) is chosen to be \( 1, 2, \ldots, \sqrt{r} \) in turn and for each \( i \), the corresponding \( j = r / i \). This is coded in the \textbf{for} loop of step 2. For each such pair of factors, the following action is performed in step 4. Two temporary pairs of values are obtained for the horizontal and vertical partition widths: one pair is obtained using \( i \) for the horizontal and \( j \) for the vertical and the other is obtained using \( j \) for the horizontal and \( i \) for the vertical. In step 5, the difference of the vertical and horizontal partition widths is compared for the two pairs. The lesser difference indicates a "more square" shape; this is then compared with the "best" difference so far. The best difference, along with the corresponding horizontal and vertical partition widths are reset. Steps 6, 7, 8 and 9 ensure that the "edge" horizontal and vertical strips are no bigger in size than the other strips.

\textbf{Horizontally flat and long rectangles}

In the sparse matrices that we have experimented with, the matrix non-zeros are structured in such a manner that partitioning the structure into horizontally flat and long vectors, and using row-major access in computations leads to high performance. To be able to partition the rectangles in this fashion we specify a vector length \( L_h \), along with the grain size \( R \), \( L_h \le R \). The rectangle is partitioned in such a manner that except for the "edge" unit rectangles, all other unit rectangles are of width \( L_h \). The code is a
procedure Partition_rectangle

1. \( l \leftarrow \sqrt{r} \), \( bestv, besth, bestd \leftarrow \text{maxval} \)
2. for \( i = 1 \) to \( i \) do

3. \( j \leftarrow r/i \)
4. if \( m < i \) then \( h \leftarrow 1 \) else \( h \leftarrow m/i \) endif
   if \( n < j \) then \( v \leftarrow 1 \) else \( v \leftarrow n/j \) endif
   if \( n < i \) then \( v' \leftarrow 1 \) else \( v' \leftarrow n/i \) endif
   if \( m < j \) then \( h' \leftarrow 1 \) else \( h' \leftarrow m/j \) endif
5. if \( (|v - h| < |v' - h'|) \) then
   if \( (|v - h| < bestd) \) then \( bestv \leftarrow v, besth \leftarrow h, bestd \leftarrow |v - h| \) endif
   else
   if \( (|v' - h'| < bestd) \) then \( bestv \leftarrow v', besth \leftarrow h', bestd \leftarrow |v' - h'| \) endif
   endif
   endiffor
6. \( p \leftarrow m/besth, q \leftarrow n/bestv \)
7. if \( m \) is not a multiple of \( p \) then
   \( besth \leftarrow besth + 1, p \leftarrow m/besth \)
   if \( m \) is not a multiple of \( besth \) then \( p \leftarrow p + 1 \) endif
   endif
8. if \( n \) is not a multiple of \( q \) then
   \( bestv \leftarrow bestv + 1, q \leftarrow n/bestv \)
   if \( n \) is not a multiple of \( bestv \) then \( q \leftarrow q + 1 \) endif
   endif
9. \( r \leftarrow p \times q \)

Figure 5.21: Procedure for partitioning of a rectangle.
fairly straightforward specific instance of the code for the square rectangles described above. The vertical partition width and hence the number of vertical partitions is set a priori, using $L_b$. Using these, and $R$, the horizontal partition width and the number of horizontal partitions are computed.

If, instead, vertically tall and thin rectangles are needed, we can specify a vector length $L_c$, $L_c < R$ in order to partition the rectangles in such a manner that except for the "edge" unit rectangles, all other unit rectangles are of height $L_c$. The code is symmetrical to that for computing horizontally flat and long partitions.

5.5.2 Computation of dependencies

In Section 5.4.4 we formulated rules, based on the intersection of projections, to compute the inter-block dependencies. In this section, we describe the algorithms and data structures used to compute these dependencies. In particular, we discuss the interval tree data structure which is used to compute the intersection of projections and is central to the efficient functioning of the partitioner.

Interval Tree

To construct the BDAG, the intersections of projections need to be computed first. For this purpose, we make use of the interval tree data structure. The interval tree allows storage of $n$ intervals in linear space such that intersection queries can be answered in logarithmic time. Let $U \subseteq R$ be a finite set and let $S = \{[x_i, y_i] ; x_i \in U, y_i \in R, 1 \leq i \leq n\}$ be a set of $n$ closed intervals on the real line. Note that the left endpoints of the intervals in $S$ are in $U$. An interval tree for $S$ (with respect to the universe $U$) is a leaf-oriented search tree for set $U$ where each node of the tree stores a node list. The node list $NL(v)$ of a node $v$ is the set of intervals in $S$ containing the split value of $v$ but of no ancestor of $v$. The split value of a leaf representing $x_i$ is $x_i$. Let $\max(y)$ be the maximum split value of a leaf node in the subtree rooted at $y$ and $\min(y)$ be the minimum split value of a leaf node in the subtree rooted at $y$. The split value of a non-leaf node $v$ with left child $l(v)$ and right child $r(v)$ is $(\max(l(v)) + \min(r(v)))/2$. The node list of a node $v$ is stored as as two sequences: the ordered list of left endpoints
and the ordered list of right endpoints.

Figure 5.22 shows an interval tree for the set of intervals

\[ S = \{[1, 8], [2, 5], [3, 7], [4, 5], [6, 8], [1, 3], [2, 3], [1, 2]\} \]

with respect to the universe \( U = \{1, 2, 3, 4, 5, 6, 7\} \). In that Figure, the root node's

split value is 4.5. The intervals stored at this node are all and only those intervals of
\( S \) which contain the split value. These intervals are \([1, 8], [2, 5], [3, 7] \) and \([4, 5]\), stored
in two sequences: one sequence stores them in order of left endpoints, i.e., in the order
\([1, 8], [2, 5], [3, 7], [4, 5]\) and the other sequence stores them in order of right endpoints,
i.e., in the order \([2, 5], [4, 5], [3, 7], [1, 8]\). These sequences (represented by the left and
right endpoints, respectively) are shown to the left and right of the root node in the
figure.

Let \(|U| = N\) and let \( I = [x_c, y_c] \) be a query interval. Let \( A = \{[x, y] \in S; [x, y] \cap
[x_c, y_c] \neq \emptyset\} \) be the set of intervals in \( S \) intersecting \( I \). The following properties of the
interval tree are important in making the computation of the intersections of projections,
and hence the detection of dependencies, efficient in space and time.
i. An interval tree for $S$ uses space $O(n + N)$.

ii. An interval tree for $S$ of depth $O(\log N)$ can be constructed in time $O(N + \n \log N + n \log n)$.

iii. $A$ can be computed in time $O(\log N + |A|)$.

See [50] for a detailed description of the interval tree and its uses.

**Grouping and storage of intervals**

Clusters are partitioned and the dependencies involving blocks in the cluster are computed by proceeding left to right along the matrix, processing clusters one by one. When a cluster $C_{m,n}$ is taken up for processing, all the clusters to its left have been processed. Projections of the blocks to the left of $C_{m,n}$ which fall within the range $[m,n]$ have already been stored at the time of processing previous clusters. An interval tree is then constructed with the universe $U$ being the set of integers in the closed interval $[m,n]$.

Before storing intervals in the tree, they are divided into *groups* to ensure that every reported intersection in a query on the tree results in at least one dependency. The updates to any unit block within a rectangular block of a cluster can be only of the base-multiplier type. Within the triangular block of a cluster, the updates to any rectangular unit block can be only of the base-multiplier type, but the updates to any triangular unit block can be of either the base type or the base-multiplier type. To facilitate the computation of these updates, the blocks in $C_{m,n}$ are given group numbers as follows: triangle $T_{m,n}$ is numbered "-1" (for the base updates) and "0" (for the base-multiplier updates), followed by numbering the rectangular blocks starting with "1" and going top to bottom. In Figure 5.12, cluster $C$ has the following group numbering associated with blocks: "-1", "0" (triangle), "1" (the rectangle composed of $R_1$, $R_2$ and $R_3$) and "2" (the rectangle composed of $R_4$, $R_5$ and $R_6$).

The projection intervals are then divided into groups. First, every base block’s projection is in group "-1", corresponding to the update of some unit triangle(s) in $T_{m,n}$ by the base block. Next, the intervals corresponding to base-multiplier pairs are
divided into groups depending on the updated block's group number. The interval for a base-multiplier pair belongs to group "i", \( i \geq 0 \) if the multiplier's horizontal projection intersects the horizontal projection of a block in group \( i \) in \( C_{m,n} \).

In Figure 5.12, consider the interval \([I_1, I'_2]\) which has been stored in cluster \( C_{m,n} \) as a result of a previous projection, and for which the projecting rectangle is \( P_1 \). First of all, the projection is in group "-1" (\( P_1 \) updates a unit triangle). The projection is also in group "0" (the unit rectangle \((c2, c3, c1, c2)\) is updated by the upper shaded portion of \( P_1 \) as base and the lower shaded portion of \( P_1 \) as multiplier) and in group "1" (\( M_1 \)'s horizontal projection intersects that of rectangle number 1). Thus the projection of \( P_1 \) is split into three intervals, one in each group, and the intervals in groups "1" and "2" are annotated with the respective multiplier information.

Each node in the interval tree stores \( 2g \) interval lists, where \( g \) is the number of groups in the cluster. There are four groups in the cluster \((c1, c4)\) of Figure 5.12. Each group has two interval lists, one list in ascending order of left end-points and the other in descending order of right end-points. Hence, there will be eight interval lists at each node of the interval tree.

Procedure Build_interval_tree in Figure 5.23 gives the algorithm for grouping and storing the intervals in the interval tree. The main loop of step 1 goes through all updaters of the cluster \( C_{m,n} \) or \( CI \). The projection interval for each updater has been already stored with the cluster. If the updater is a column \( c \), \( z' \) is the non-zero with the least row number which is contained in the extent \([m, n]\). The column updates the unit triangle in \( T_{m,n} \) which contains \( L_{z',c} \) and the interval \([c, c]\) is therefore stored in group "-1". Then, starting with the first non-zero following \( L_{z',c} \) in column \( c \), non-zeros are matched against the rectangles in \( C_{m,n} \) to identify groups. Suppose the non-zero \( L_{z,c} \) matches with a rectangle in group \( g \), whose bottom-most row number is \( r \); \( z \) is the least row number of a non-zero in \( c \) for which such a match can occur. Then, the non-zeros \( L_{i,c}, z \leq i \leq r \) are all used in updating this rectangle and can be skipped over in looking for the next matching group. This is achieved by doing a binary search of \( c \) for a non-zero with the least row number > \( r \). If the updater is a rectangle \( R \), it becomes the base block for each multiplier block below it. Each of these multiplier blocks is
procedure $Build\_interval\_tree$

1. foreach updater $u$ do
   
   if ($u$ is a column $c$) then
      store interval $[c, c]$ in group “-1”
      $z' \leftarrow$ row number of first non-zero in $c$ s.t. $z' \in [m, n]$
      $z \leftarrow$ row number of non-zero immediately following $z'$ in $c$
      do a binary search of $Cl$ to find group $g$ for $z$
      store interval $[c, c]$ in group $g$
      $r \leftarrow$ bottom row of matching rectangle or triangle
      while (there are unprocessed non-zeros in $c$) do
         do a binary search of $c$ for first non-zero beyond row $r$
         $z \leftarrow$ row number of this non-zero
         do a binary search of $Cl$ to find group $g$ for $z$
         store interval $[c, c]$ in group $g$
         $r \leftarrow$ bottom row of matching rectangle
      end while
   
   else
      $u$ is a rectangle updater $R$, $[r, r']$ is interval or projection
      store interval $[r, r']$ in group “-1”
      foreach rectangle below $R$ do
         do a binary search of $Cl$ to find group $g$ for $R$
         store interval $[r, r']$ in group $g$
      end foreach
   
   end if

end foreach

2. sort the stored intervals in ascending order with left endpoint
   as primary key and group number as secondary key

3. store the intervals in the interval tree

---

Figure 5.23: Procedure for building an interval tree.
matched against some rectangle in $C_{m,n}$ by doing a binary search of the rectangles in $C_{m,n}$. Steps 2 and 3 build and store the intervals in the interval tree. For details on how to store the intervals in an interval tree, refer [50].

**Querying the interval tree**

When the dependencies for a unit block are to be constructed, the interval tree can be queried with (a) the query interval which is the vertical projection of the unit block and (b) the group number, $g$, which is the group number of the block to which the unit belongs. The intersections reported will be either a set of base updaters (if $g$ is "-1") or a set of base-multiplier pairs (if $g$ is not "-1") since a unit rectangle needs a base-multiplier pair for its update, group number "-1" will be only used for the unit triangle.

When $g$ is "-1", the base updaters returned will each update the unit triangle for which the query was made. Since the base updaters themselves may be partitioned into unit blocks, each of the unit blocks in each base updater will update the unit triangle. Thus every intersection reported results in at least one dependency. For other values of $g$, there is potential inefficiency if queries are handled naively. Suppose a query is made for a unit rectangle $R$ which belongs to a rectangular block $D$ whose group number is $g$. Let $S$ be the set of base-multiplier pairs returned. Corresponding to each pair of intervals in the intersection, there is a pair of base and multiplier blocks, $(S, M)$, such that $HP(S)$ intersects $VP(R)$, $VP(S)$ intersects $VP(M)$ and $HP(M)$ intersects $HP(D)$. The inefficiency arises in the situation where $HP(M)$ does not intersect $HP(R)$ and therefore the pair $(S, M)$ does not update $D$, resulting in a wasted output interval.

To overcome this inefficiency by guaranteeing that every intersection reported on a query results in at least one dependency, one query is made for each *vertical strip* in the triangle and each vertical strip in each of the rectangles. In fact, this also reduces the number of queries as compared to making one query for each unit block. Using one query for a vertical strip, dependencies for all the unit blocks in that strip can be computed efficiently.
Figure 5.24 shows a general query situation.

Figure 5.24: Query with a vertical strip.

Algorithm Query\_interval\_tree in Figure 5.25 shows how the dependencies are computed for a given vertical strip.

**procedure** Query\_interval\_tree

1. Query the interval tree with \([c, d]\) and \(y\)
2. if \(y = "1"\) then
   \(S \leftarrow\) set of base updaters, \(T \leftarrow\) query triangle
   **foreach** \(U \in S\) do Base\_deps\((U, T)\) **end foreach**
else
   \(S \leftarrow\) set of base-multiplier pairs of updaters, \(V \leftarrow\) vertical strip of query
   **foreach** \((U, M) \in S\) do Base\_mult\_deps\((U, M, V)\) **end foreach**
**end if**

Figure 5.25: Procedure for querying an interval tree.

In step 1, a query is made on the interval tree with the appropriate group number. If the group number is “1”, the query returns a set of base updaters for the unit triangle at the top of the vertical strip. In step 2, \(S\) is the set of base updaters which correspond to the intervals of intersection. Each base updater block \(U\) may be itself partitioned into unit blocks (if \(B\) is not a column). In this case, the procedure Base\_deps, shown
in Figure 5.26, sets up dependencies $B \delta T$ for all unit blocks $B \in U$ for which $HP(B)$ intersects $HP(T)$.

\begin{verbatim}
procedure Base_deps

0. $T \leftarrow$ input triangle
1. if $U$ is a column then $U \delta T$
   else
2.   compute $S$, the topmost horizontal strip in $U$ for which $HP(S) \cap HP(T) \neq \emptyset$
3.   while $(HP(S) \cap HP(T) \neq \emptyset)$ do
    foreach $B \in S$ do $B \delta T$ endfor
    $S \leftarrow$ next horizontal strip after $S$
   end while
end if
\end{verbatim}

Figure 5.26: Procedure for computing base dependencies.

For example, in Figure 5.24, the block containing the $P_i$'s updates the unit triangle $T$. The topmost horizontal strip in this block whose horizontal projection intersects $HP(T)$ is the second horizontal strip i.e. the one which is partitioned into $P_1, P_2$. This strip gives rise to the dependencies $P_1 \delta T$ and $P_2 \delta T$. The next horizontal strip also updates $T$, and gives rise to the dependencies $P_3 \delta T$ and $P_4 \delta T$.

Before we discuss how the dependencies are computed for group numbers which are not "-1", we describe a procedure which we refer to as $Merge$-$scan$ that is repeatedly used in the computation of these dependencies. This procedure takes two vertical (resp. horizontal) strips of blocks and returns a set of pairs of blocks whose horizontal (resp. vertical) projections intersect each other. Let $S_1$ and $S_2$ be two strips, with extents $(e_1, e_1')$ and $(e_2, e_2')$ respectively. If $S_1$ and $S_2$ are horizontal strips, $e_1$ and $e_2$ are the leftmost edges and $e_1'$ and $e_2'$ are the rightmost edges; if they are vertical strips, $e_1$ and $e_2$ are the topmost edges and $e_1'$ and $e_2'$ are the bottommost edges. If one of $S_1$ or $S_2$ (say $S_1$) is a part of a column then $e_2$ is the row number of the first non-zero and $e_1'$ is the row number of the last non-zero in the part under consideration. Each of $S_1$ and $S_2$ (if they are not columns) has been partitioned into unit blocks.

The procedure essentially scans the strips in parallel, much like the algorithm for merging two sorted lists of values. The strips are matched against each other starting at
$e_1$ and $e_2$ respectively. One pointer is maintained for each strip, pointing to the current unit block in the scan of that strip. The pointers are initially respectively set to $p_1 = b_{1i}$ and $p_2 = b_{2i}$, where $b_{1i}$ and $b_{2i}$ are the first unit blocks in $S_1$ and $S_2$ respectively whose projections intersect each other. This initial pointer setting can be done in constant time. The loop invariant ensures that the projections of the respective blocks to which the pointers point intersect each other.

Suppose in the $i$th iteration, the pointers are at $p_1 = b_{1i}$ and $p_2 = b_{2i}$. Then the pair $(b_{1i}, b_{2i})$ is reported. Next, there are three cases for advancing the respective pointers. We will describe these cases assuming horizontal strips; the discussion is symmetric for vertical strips. Let $r_1$ and $r_2$ be the locations of the right edges of $b_{1i}$ and $b_{2i}$ respectively.

a) If $r_1 < r_2$, $p_1$ is advanced to the unit block immediately to the right of $b_{1i}$ in $S_1$;
b) If $r_1 > r_2$, $p_2$ is advanced to the unit block immediately to the right of $b_{2i}$ in $S_2$;
c) If $r_1 = r_2$, $p_1$ and $p_2$ are both advanced to respectively point to the unit block immediately to the right of $b_{1i}$ in $S_1$ and the unit block immediately to the right of $b_{2i}$ in $S_2$.

The scan ends when both pointers are advanced beyond $e'$, where $e'$ is the minimum of $e'_1$ and $e'_2$. Obviously, the pairs of blocks are reported in time linear in the total number of pairs reported.

If one of the strips is a column, a unit block is replaced by a row extent. Suppose $S_1$ is a column. Let $(b_1, b_2)$ be one of the reported pairs. Then $b_2$ is essentially the strip of non-zeros in the column such that the row numbers of the first and last non-zeros are within the top and bottom edges of $b_2$, respectively. ($S_2$ must be a vertical strip). Also, one or both of $S_1$ and $S_2$ could be a strip of strips, i.e., instead of being comprised of unit blocks, they could be comprised of strips of unit blocks. For instance $S_1$ could be a vertical strip, with each horizontal partition in it being a horizontal strip instead of a unit block, and $S_2$ could be a vertical strip. A pair $(b, b')$ reported by Merge_scan would mean that the horizontal projection of the horizontal strip $b$ intersects the horizontal projection of the unit block $b'$. 


If the group number is other than "-1", the query returns a set of base-multiplier updater pairs. For each such base-multiplier pair, the procedure $\textit{Base\_mult\_deps}$, shown in Figure 5.27, is called to compute the dependencies for the vertical strip.

\begin{verbatim}
procedure $\textit{Base\_mult\_deps}$

1. if \(U = M\) is a column then
   \[P \leftarrow \text{Merge\_scan}(U, V)\]
   \[\text{foreach } (b, b') \in P \text{ do } U \delta b' \text{ end foreach}\]
else
2. compute $S_u$, the topmost horizontal strip in $U$ for which $HP(S_u) \cap VP(V) \neq \emptyset$
3. while \((HP(S_u) \cap VP(V) \neq \emptyset)\) do
   \[P_s \leftarrow \text{Merge\_scan}(M, V)\]
   \[\text{foreach } (S_m, B) \in P_s \text{ do}\]
   \[P_u \leftarrow \text{Merge\_scan}(S_u, S_m)\]
   \[\text{foreach } (b, b') \in P_u \text{ do } b, b' \delta B \text{ end foreach}\]
end foreach
   \[S_u \leftarrow \text{next horizontal strip after } S_u \text{ in } U\]
end while
end if
\end{verbatim}

Figure 5.27: Procedure for computing base-multiplier dependencies.

The working of this procedure is best described by means of an example. In Figure 5.24, suppose we want to compute the dependencies for the vertical strip which contains $R$. On doing a query, let one of the pairs of base-multipliers returned be the rectangle containing the $P_i$'s and the rectangle containing the $M_i$'s respectively.

In step 2 of procedure $\textit{Base\_mult\_deps}$, $S_u$ corresponds to the strip containing $P_1$ and $P_2$. In step 3, one of the pairs returned in $P_s$ is $(Q, R)$, where $Q$ is the horizontal strip containing $M_1, M_2, M_3$. The corresponding $P_u$ contains the pairs $(P_1, M_1)$, $(P_1, M_2)$, $(P_2, M_1)$, $(P_2, M_3)$, all of which update $R$. Another pair in $P_s$ is $(Q', R)$ where $Q'$ is the horizontal strip containing $M_4, M_5, M_6$. The corresponding $P_u$ contains the pairs $(P_1, M_4)$, $(P_1, M_5)$, $(P_2, M_2)$, $(P_2, M_6)$ which update $R$.

In the next iteration of the \texttt{while} loop, $S_u$ corresponds to the strip containing $P_3$ and $P_4$. By a similar process in the \texttt{for} loop of statement 3, the pairs $(P_3, M_1)$, $(P_3, M_2)$, $(P_3, M_4)$, $(P_3, M_3)$, and $(P_4, M_2)$, $(P_4, M_3)$, $(P_4, M_5)$, $(P_4, M_6)$ are identified and all of these update $R$. 
Dependencies internal to cluster

The discussion above described how dependencies of the type $A \delta C$ or $A, B \delta C$ are computed, where $C$ is in a cluster and $A$ and $B$ are external to the cluster to which $C$ belongs. We now describe how dependencies between blocks entirely within a cluster are computed. These dependencies are computed without using the interval tree.

Consider first the triangle $T_{m,n}$ in a cluster $C_{m,n}$. Let some unit triangle $B$ in $T_{m,n}$ belong to the horizontal strip $H$ in $T_{m,n}$. Then, $A \delta B$ for all unit rectangles $R$ such that $R \in H$. This is computed in step 1 of the procedure $InternalTri Deps$, shown in Figure 5.28.

---

**procedure** $InternalTriDeps$

1. foreach unit triangle $T$ do
   
   $H \leftarrow$ horizontal strip to which $T$ belongs
   
   foreach unit rectangle $R$ in $H$ do $R \delta T$ end foreach

2. foreach horizontal strip $H$ do
   
   foreach unit rectangle $R$ in $H$ do
   
   $V \leftarrow$ vertical strip to which $R$ belongs
   
   $T \leftarrow$ unit triangle contained in $V$
   
   $T \delta R$
   
   $H_t \leftarrow$ horizontal strip to which $T$ belongs
   
   foreach unit rectangle $R_m$ to the left of $R$ in $H$
   
   $R_s \leftarrow$ unit rectangle in $R_m$’s vertical strip and in $H_t$
   
   $R_s, R_m \delta R$

end foreach

end foreach

---

Figure 5.28: Procedure for computing intra-cluster triangle dependencies.

To compute the dependencies for unit rectangles in $T_{m,n}$, let $R$ be one such unit rectangle in $T_{m,n}$, and let $R$ belong to the vertical strip $V$ and horizontal strip $H$. Then, $A \delta R$ where $A$ is the unit triangle in $V$. Let $A$ belong to the horizontal strip $H_a$. For every vertical strip $V_i$ to the left of $V$, $B, C \delta R$ where $B \in H_a, B \in V_i$ and $C \in H, C \in V_i$. The dependencies for $R$ are computed in step 2 of the procedure $InternalTriDeps$. 
Next, consider the rectangles below $T_{m,n}$. For each of these rectangles, the following dependencies are computed. Let $R$ be a unit rectangle which belongs to the vertical strip $V$ and horizontal strip $H$ in some rectangle below $T_{m,n}$. Let $H_s$ be a horizontal strip in the triangle $T_{m,n}$ such that $HP(H_s) \cap VP(V) \neq \emptyset$. Then, $A,B \in R$ where $A \in H_s, B \in H$ and $VP(B) \cap VP(A) \neq \emptyset$. $B$ is in some vertical partition to the left of $V$ in the rectangle.

**Procedure Internal_rect_deps**

1. compute $S_u$, the topmost horizontal strip in $T$ for which $HP(S_u) \cap VP(V) \neq \emptyset$
2. while $(HP(S_u) \cap VP(V) \neq \emptyset)$ do
3.   foreach unit rectangle $R \in V$ do
   - $H \leftarrow$ part to the left of $R$ of the horizontal strip to which $R$ belongs
   - $P_u \leftarrow$ Merge_scan($S_u, H$)
   - for each $(b,b') \in P_u$ do $b,b' \in R$ end foreach
   - end foreach
   - $S_u \leftarrow$ next horizontal strip after $S_u$ in $U$
   - end while

---

**Figure 5.29:** Procedure for computing intra-cluster rectangle dependencies.

Procedure Internal_rect_deps, in Figure 5.29, gives the algorithm to compute these dependencies. $S_u$ in step 1 can be computed in constant time since the sizes of the partitions in the triangle and rectangle are known.

**Top-level procedures**

Procedures Process_column and Process_cluster are the top-level partitioning routines which tie everything together. These are respectively shown in Figure 5.30 and Figure 5.31. Columns and clusters are processed going left to right in the matrix. When a column is encountered in this left-to-right scan, the procedure process_column is called to process it; when a cluster is encountered, the procedure process_cluster is called. The data structure ColTree is a dynamic balanced binary search tree, which stores those columns which depend on at least one block, but which have not yet been processed. A column is independent if it has no updaters. The data structure ColTree is an array
which stores the clusters in left to right order of their column extents.

procedure \textit{Process\_column}

\begin{itemize}
\item \textbf{if} \quad \textit{c} is not an independent column \textbf{then}
\item \hspace{1cm} delete \textit{c} from \textit{ColTree}
\item \hspace{1cm} \textbf{foreach} \ \textit{u} in \textit{c}'s list of updaters \ \textbf{do} \ \textit{u} \ \& \ \textit{c} \ \textbf{end foreach}
\item \textbf{end if}
\item \textbf{foreach} non-zero in row \textit{j} in column \textit{c} \ \textbf{do}
\item \hspace{1cm} do a binary search of \textit{ColTree} for \textit{j}
\item \hspace{1cm} \textbf{if} \quad \textit{match is found} \textbf{then}
\item \hspace{2cm} \textit{Cl} \leftarrow \textit{matching cluster}
\item \hspace{2cm} store interval \([j, j]\) in \textit{Cl}
\item \textbf{else}
\item \hspace{2cm} do a binary search of \textit{ColTree} for \textit{j}
\item \hspace{2cm} \textbf{if} \quad \textit{found} \textbf{then}
\item \hspace{3cm} add \textit{c} as \textit{j}'s updater
\item \textbf{else}
\item \hspace{3cm} insert \textit{j} (with \textit{c} in it's updater list) into \textit{ColTree}
\item \textbf{end if}
\item \textbf{end if}
\item \textbf{end if}
\item \textbf{end foreach}
\end{itemize}

\begin{figure}
\begin{center}
\begin{tabular}{|c|c|c|}
\hline
\textit{Column} & \textit{Cluster} & \textit{Column}\textit{Cluster} \\
\hline
\textit{ColA} & \textit{Cl1} & \textit{ColB} \\
\hline
\textit{ColC} & \textit{Cl2} & \textit{ColD} \\
\hline
\end{tabular}
\end{center}
\caption{Procedure for processing a column.}
\end{figure}

\subsection{Parallelizing the partitioner}

The partitioner has been parallelized with the aim of distributing data to achieve scalability. The emphasis is not on efficient parallelization of the computational work.

Let there be \(p\) processors, numbered 0 through \(p\). Let the columns of the matrix be numbered 1 through \(N\), and the clusters be numbered 1 through \(Q\), where \(N\) is the order of the matrix and \(Q\) is the number of clusters identified by the pre-partitioner. Note that a column is treated as a single column by the partitioner if and only if it is not contained in any cluster of acceptable width. Let the set of such single columns be \(SC\). The columns in \(SC\) are ordered as follows. A column \(i\) is ordered ahead of column \(j\) if and only if \(i < j\). Let the set of clusters be \(CL\). The clusters in \(CL\) are ordered as follows. A cluster \(C_{m,n}\) is ordered ahead of cluster \(C_{p,q}\) if and only if \(m < p\).

The clusters in \(CL\) and the columns in \(SC\) are distributed among the processors in
procedure $Process\_cluster$

1. call $Build\_interval\_tree$

2. $W \leftarrow$ work in triangle, $S \leftarrow$ size of triangle
   $Z \leftarrow$ minimum non-zeros, $A \leftarrow$ minimum work
   $P \leftarrow S/Z$ or $P \leftarrow W/A$
   call $Partition\_triangle$

3. foreach vertical partition in the triangle do
   call $Query\_interval\_tree$, group “1”
   call $Query\_interval\_tree$, group “0”
   call $Internal\_tri\_deps$
end foreach

4. foreach rectangle $R$ below triangle do
   $g \leftarrow g + 1$
   $W \leftarrow$ work in $R$, $S \leftarrow$ size of $R$
   $P \leftarrow S/Z$ or $P \leftarrow W/A$
   call $Partition\_rectangle$
   foreach vertical partition in $R$ do
     call $Query\_interval\_tree$, group $g$
     call $Internal\_rect\_deps$
   end foreach

5. do a binary search of $CITree$ to find match with $R$
   if $R$ (projects to a set of columns $S$) then
     foreach column $c \in S$ do
       do a binary search of $c$ in $ColTree$
       if (found) then
         add $R$ as $c$’s updater
       else
         insert $c$ (with $R$ as an updater) in $ColTree$
       end if
     end foreach
   else
     $S \leftarrow$ set of clusters matching $R$’s projection
     foreach cluster $C \in S$ add $R$ as updater end foreach
   end if
end foreach

---

Figure 5.31: Procedure for processing a cluster.
wrap-around fashion as follows. The $k$th column in $SC$, $1 \leq k \leq |SC|$, is assigned to processor $(k - 1)mod p$ and $r$th cluster in $CL$, $1 \leq r \leq |CL|$, is assigned to processor $(r - 1)mod p$. A processor “owns” the clusters and columns that are mapped to it.

Each processor computes the dependencies for the clusters and columns it owns, in parallel with the other processors. A primary reason for not attempting control parallelism is that the number of dependencies per cluster or column is not known beforehand, since these dependencies themselves are computed by the partitioner. Thus it is not possible to assign balanced workload to the processors. The parallel code schematic is shown in Figure 5.32.

The parallel code differs from the sequential version in two respects:

1. A processor computes the dependencies for a column or a cluster only if it owns that column or cluster. This is shown by the “My column?” and “My cluster?” checks in Figure 5.19.
2. Consider the projection, onto the diagonal, of a non-zero in a column or a rectangle in a cluster. Let this projection be computed in processor $q$. Such a projection is stored in the matching cluster or column, say $C$, only if $C$ is also owned by $q$. Otherwise, the projection is discarded. This is because $q$ needs the projections onto $C$ to compute dependencies for $C$, only if does $q$ owns $C$.

5.6 Conclusions

In this chapter we introduced a novel block partitioning scheme for sparse Cholesky factorization based on a hybrid mixture of sparse columns, dense triangles and dense rectangles. Tuning knobs in the form of parameters are provided in order that the partitioner may be ported across problems with widely ranging sparsity structures and across architectures with different granularities.

We noted that the partitioning geometry has a powerful influence on the utilization of the cache, vector processors and pipelined floating point units of current RISC-based architectures. This leads us to believe that it is extremely important for a partitioner to closely identify the structure of the data for a given problem, which in turn necessitates the construction of problem-specific partitioners.
Chapter 6
Scheduling and communication optimization

Following the partitioning of the algorithm, the next phase is to schedule the computations and communication on the processors of the target machine, using the BCR. This phase is divided into two steps: (a) allocation and (b) scheduling of computations and inter-processor communication. In the allocation step, the data blocks are distributed among the processors. A processor owns the data blocks which have been allocated to it. All computations that modify a block of data are performed in the processor that owns that block. Thus, the allocation of tasks to processors is completely determined by the allocation of data blocks to processors.

The scheduling of computations and communication is broken up into two stages. The first stage is a one-time pre-pass, which results in a static schedule on every processor. In this stage, the scheduler executes the algorithm in parallel, using the BCR. It uses a dynamic, data-driven strategy to locally schedule, on every processor, the execution of tasks and the messages sent from and received on that processor. However, the actual numerical computations associated with the tasks are performed only if required. If the pre-pass is done in the symbolic mode, a task is reported to be executed without actually performing the numerical computations, i.e. effectively, the cost of a floating point operation is assumed to be zero. If the pre-pass is done in the non-symbolic mode, each task is executed in the true sense, by performing all the numerical floating point operations that constitute that task. In either case, actual data messages are sent among processors. Thus, if the pre-pass is done in the non-symbolic mode, the numerical answers to the problem are available at the end of the pre-pass itself. In either mode, during the course of the pre-pass, the scheduler maintains a record of the order in which the tasks are executed and the order in which messages are sent and
received on every processor. Apart from this, it records, for every message, the size of the message, and its destination if the message is sent out from the processor. The output of the pre-pass is, therefore, a static task and communication schedule for each processor.

The second stage of the scheduling consists of the subsequent passes or iterations in which the static schedule generated by the pre-pass is followed to execute the algorithm, given that the symbolic structure of the problem does not change, while the data values change between iterations. Unlike the dynamic scheduling used in the pre-pass, the static scheduling in the second stage avoids both the overheads and the complexity of manipulating data structures incurred in order to manage the execution of tasks depending on the arrival of data on a processor. Apart from this, the complete a priori knowledge of inter-processor communication gathered from the pre-pass allows for aggressive communication optimization in which the the latency of message passing may be reduced significantly. We have designed a deadlock-free, parallel communication optimization algorithm for unstructured computations, as a part of this second stage of the scheduler. The complexity of code, the distribution of data among processors and the details of message-passing are all hidden from the application programmer by providing an easy-to-use interface consisting of three primitives called SET_UP, READY_BLOCK and GET_BLOCK.

The allocation and scheduling pre-pass are the additional preprocessing steps after partitioning. The scheduling support in the second stage of iterative computations, including the communication optimization algorithm, is referred to as execution-time support.

In this chapter we first describe an allocator which is a sequential program specific to block sparse Cholesky factorization. We then present the general scheduler pre-pass algorithm for unstructured computations. The algorithm is designed in the style of Single Program Multiple Data, or SPMD, code. Next, we describe the design of the primitives for block-based unstructured computations. The application of the scheduler to block sparse Cholesky factorization is presented. This includes both the pre-pass algorithm, and the code using primitives for computations in the subsequent iterations
of the factorization. Finally, we present the communication optimization algorithm.

6.1 Allocation for block sparse Cholesky factorization

The allocation of blocks to processors proceeds in a cluster by cluster manner. We describe two allocation strategies. These differ in the order in which the clusters are chosen for allocation. In the following, we first describe the allocation policy for blocks within a given cluster and then describe the two strategies used for ordering the selection of clusters.

6.1.1 Allocation of cluster

If a cluster \( Cl \) is not a single column, processors are assigned to the component partitions using the procedure \( Allocate\_cluster \) of Figure 6.1, with the aim of balancing the load among processors. In that procedure, \( P \) is the total number of available processors. A priority queue, \( Pheap \), stores all processors, with the highest priority given to the processor which has the least amount of computational work assigned so far. Let \( k \) be the number of partitions in \( Cl \)'s triangle. Let \( r \) be \( \min(P, k) \). Then the top \( r \) processors in \( Pheap \), i.e., those with the least computational work, are picked for assignment to the unit blocks in the cluster. Let the set of these processors be \( PA \).

The processors in \( PA \) are maintained in a temporary priority queue, \( CHeap \), with the same priority criterion as for \( Pheap \). Using \( CHeap \), first, the unit blocks in the triangle are allocated by going through the unit triangles first and then going through the unit rectangles top to bottom, and left to right along each horizontal strip, i.e., in block row major order. For every unit, the processor in \( PA \) with current minimum load is assigned to it. Next, for each rectangle below the triangle, the unit blocks are traversed in block row major order, and for every unit block, the processor in \( PA \) with current minimum workload is assigned to it.
procedure allocate_cluster

0. \( k \leftarrow \text{number of partitions in triangle}, \ r \leftarrow \min(P, k), \ C\text{Heap} \leftarrow \text{empty} \)

1. \( PA \leftarrow \text{top} \ r \text{ processors from } P\text{Heap} \text{ in heap order} \)
   delete these from \( P\text{Heap} \) and insert into \( C\text{Heap} \)

2. \textbf{foreach} unit block in triangle \textbf{do}
   \( p' \leftarrow \text{top of } C\text{Heap} \)
   assign \( p' \) to the block
   increase the key for \( p' \) in \( C\text{Heap} \) with work in block and reheap \( C\text{Heap} \)
\textbf{end foreach}

3. \textbf{foreach} rectangle below the triangle \textbf{do}
   \textbf{foreach} unit block in rectangle \textbf{do}
   \( p' \leftarrow \text{top of } C\text{Heap} \)
   assign \( p' \) to the block
   increase the key of \( p' \) in \( C\text{Heap} \) with work in block and reheap \( C\text{Heap} \)
   \textbf{endfor}
\textbf{end foreach}

4. \textbf{while} \( (C\text{Heap} \text{ is not empty}) \) \textbf{do}
   \( p' \leftarrow \text{next processor in } C\text{Heap} \)
   delete \( p' \) from \( C\text{Heap} \)
   insert \( p' \) into \( P\text{Heap} \) with current work
\textbf{endwhile}

---

Figure 6.1: Allocation of blocks in cluster.

6.1.2 Selection of cluster and allocation of column

As mentioned earlier, the clusters may be selected for allocation in two ways. In the first, which is a simpler version, the clusters are sorted by workload in a decreasing order and then are picked for allocation in this order. If the cluster is a column, then it is assigned the processor with current minimum work load.

The second version uses the cluster dependency DAG (CDAG). The CDAG is obtained from the BDAG by collapsing all nodes that belong to the same cluster into a "clusternode". All the edges among the nodes which are collapsed to form a clusternode are discarded. The remaining edges in the BDAG form the directed edges among the clusternodes. All duplicate edges between any pair of clusternodes are merged into a single edge. Each clusternode in the CDAG is annotated with a weight equal to the
total workload in the cluster represented by that clusternode. This is the sum of
the weights of all the nodes that collapsed to form that clusternode. Each directed edge in
the CDAG is annotated with a measure of the volume of data communicated from the
source clusternode to the dependent clusternode, as a result of the dependencies from
the unit blocks of the source cluster to the unit blocks of the dependent cluster.

Procedure $GlobalAllocate$ of Figure 6.2 is the the allocation procedure using the
CDAG. A cluster is ready for allocation only after all its predecessors in the CDAG

```plaintext
procedure global_allocate
0. Insert all ready clusters (no predecessors) into $Clheap$
1. while ($Clheap$ is not empty) do
   $Cl$ — cluster at top of $Clheap$, delete top of $Clheap$
   if ($Cl$ is a column) then $allocate\_column$
   else $allocate\_cluster$
   endif
   foreach child $Ci$ of $Cl$ in the CDAG
      decrement predecessor count $pred(i)$ by 1
      if ($pred(i) = 0$)
         insert $Ci$ in $Clheap$
      endif
   endforeach
   endwhile
```

Figure 6.2: Selection of cluster for allocation.

have been allocated processors. A heap, $Clheap$, maintains a priority queue for the
ready clusters, in descending order of work in the clusters. Along with $Pheap$, it works
towards balancing the workload among the processors. In each iteration of the allocation
process, the cluster, $Cl$, at the top of $ClHeap$ (i.e., the ready cluster with maximum
work) is picked next for allocation. After allocation, the new ready clusters are inserted
in $Clheap$.

In the above scheme, if $Cl$ is a column, the procedure $process\_column$ identifies the
predecessor $Cr$ which has maximum communication with $Cl$. If $Cr$ is a column, the
same processor $p$ that is assigned to $Cr$ is assigned to $Cl$. Otherwise, if $Cr$ is a cluster,
one processor is arbitrarily picked from the processors that are assigned to $Cr$, and is
assigned to $C_l$.

Among other parameters, the structure of $L$ affects the performance of two strategies presented here. We leave the quantification of this effect to future study. The performance results presented in this dissertation use the first version where prior to allocation the clusters are sorted in decreasing order of workload.

6.2 Scheduling for unstructured computations

The scheduling is divided into two stages: (a) the scheduling pre-pass, which is a parallel dynamic scheduling of the BDAG and (b) the static scheduling, which is the scheduling of numerical computations of the algorithm using the schedule generated by the pre-pass. The static scheduling is supported by primitives, including a communication optimization algorithm. In this section, we first describe the pre-pass. We then give the parallel code that obeys the static schedule using primitives, and lastly, we describe the functionality of the primitives.

6.2.1 Scheduling pre-pass

The scheduling pre-pass is done at run-time, but before the numerical computations of the actual algorithm. The goals of the pre-pass are the following.

- To generate, for each processor, a complete ordering of the tasks to be executed on that processor.

- To generate the complete inter-processor communication structure for the algorithm.

The pre-pass results in a static schedule, which is subsequently used to execute the computations of the algorithm.

The motivation for static scheduling arises from the inefficiency of on-the-fly scheduling of tasks and communication during algorithm execution. The first cause of inefficiency is attributable to the data-driven nature of computations, in which the activation of tasks at execution time depends on the arrival of data and is not implicitly derivable. To keep track of the arrival of data and the resultant activation of tasks, with
possible prioritization of task execution, several data structures need to be maintained and manipulated. This adds to the execution time. Secondly, communication is more expensive when the communication structure is not known beforehand. With a priori knowledge of the spatial and temporal communication pattern, the communication cost can be reduced by aggressive optimization during the execution of the algorithm. In the rest of the dissertation we will use the terms communication pattern, communication structure, and communication profile interchangeably.

In the pre-pass stage, the BDAG is traversed and “executed” in parallel. Execution proceeds dynamically, in parallel, obeying all dependency constraints among tasks and performing all inter-processor communication as required. However, the numerical computations required by the computational tasks are not carried out. Hence the execution is merely symbolic. In the rest of this section, when we say execution, we mean symbolic execution.

The scheduler needs the following information on a processor $p$ before it starts executing; $B_p$ denotes the set of blocks mapped to processor $p$.

- The AVR for each block in $B_p$.
- The task information of all the tasks that update the blocks in $B_p$. For each task, this information is the task number, the tuple representation for the task. Note that the program module to which the task is mapped can be obtained by applying the mapping function $\Gamma$ to the task number.
- The edges $(x, y)$ in the BDAG for which $\Psi(x) \in B_p$.

We call this information the part of the BCR that is local to $p$.

During the execution of the BDAG, a task can be in one of three states. In the executable state, the precedence constraints for the task have been satisfied and all the blocks that participate in the task are available on the processor i.e. the task can be executed at any time thereafter. In the waiting state, at least one precedence constraint for the task has not been satisfied. In the sleeping state, none of the precedence constraints for the task have been satisfied.
We assume that each block is assigned a priority of execution before the scheduler starts executing. All tasks that update a block get the same priority. The tasks in the executable state are stored in a dynamic priority queue called exec.q. Each entry in the queue is in fact a set of tasks that update the same block, and the queue is keyed on the block priorities. The tasks in this set may be executed in any relative order. Let the function \( \rho : B \to N \), from the set of blocks to the set of natural numbers, define the block priorities. The block \( B_i \) gets a higher priority than block \( B_j \) if and only if \( \rho(B_i) > \rho(B_j) \). If there are two executable tasks, one which updates \( B_i \) and the other which updates \( B_j \), the former is executed before the latter.

At the time of construction of the BDAG, the predecessor count of each node is computed. Each node \( x \) has two predecessor counts, \( predI(x) \) and \( predII(x) \), for the number of incoming type I and type II edges, respectively. A priority queue, ready.q is maintained for nodes. The priority of a node \( x \) is \( \rho(\Psi(x)) \). A task, represented by the edge \((x, y)\) in the BDAG, in the waiting state, waits "on" node \( y \). Recall from the BDAG representation that an edge \((x, y)\) of type I represents a part of a task, i.e. there may be several type I edges, \((x_i, y)\), sharing a task, and all of these edges share the node \( y \). For every waiting task \( \tau \) we keep a count \( predI(\tau) \), which is the number of type I edges that share that task. Task \( \tau \), all of whose representative edges share node \( y \), goes from the sleeping state to the waiting state when its first representative edge is "released". It then waits on node \( y \). It goes from the waiting state to the executable state when all its remaining representative type I edges are released, and all type II edges into \( y \) are released.

During the parallel execution of the BDAG, the predecessor counts of nodes are decremented as and when precedence relationships are satisfied and tasks executed. When a node's predecessor count goes to 0, all edges going out from that node are released. The semantics of the release of an edge is described in procedure Release.edge of Figure 6.3.

Steps 1 and 2 carry out the book-keeping associated with the satisfaction of precedence relationships, and are self-explanatory. In the course of decrementing the number
procedure \(\text{Release}_\text{edge}(x,y)\)

1. if \(((x, y)\text{ is a type I edge})\) then \(\text{predII}(y) \leftarrow \text{predII}(y) - 1\) endif

2. if \(((x, y)\text{ is a type II edge})\) then
   \(\text{predII}(y) \leftarrow \text{predII}(y) - 1\)
   if \(((x, y)\text{ represents task } \tau)\) then
     \(\text{predI}(\tau) \leftarrow \text{predI}(\tau) - 1\)
     if \((\tau\text{ is in sleeping state})\) then
       transfer task \(\tau\) to waiting state
     endif
   endif
endif
endif

3. if \((\text{predII}(y) = 0)\) then
   foreach (task \(\tau\) waiting on \(y\) s.t. \(\text{predI}(\tau) = 0\))
   add \(\tau\) to \(\text{exec.q}\)
   endforeach
endif

---

Figure 6.3: Procedure to release an edge in a BDAG.

of precedences, one or more tasks may go from the waiting to the executable state, as detected in step 3. A necessary condition for a task which is waiting on \(y\) to become executable is that there are no unreleased type II edges into \(y\). Then, if the edge \((x, y)\) is the last unreleased shared edge for task \(\tau\), when \((x, y)\) is released, all the \(\text{predI}\) constraints on \(\tau\) are satisfied and \(\tau\) becomes executable. It is then added to \(\text{exec.q}\), in proper prioritized order, with its priority of \(\rho(\Psi(y))\).

We now describe the mechanics of inter-processor communication. Consider an edge \((x, y)\) of type I which is released. If the block represented by \(x\), \(\Psi(x)\) and the block represented by \(y\), \(\Psi(y)\), are mapped to the same processor, i.e. \(\Pi(\Psi(x)) = \Pi(\Psi(y)) = P_x\), then there is no communication, and the edge is released in \(P_x\). Otherwise, the edge must be released in processor \(\Pi(\Psi(y))\), since satisfaction of precedence relationships for node \(y\), or change of state of tasks waiting on \(y\) have to be effected on \(\Pi(\Psi(y))\). Since the edge \((x, y)\) is stored in \(P_x\), it must be sent to \(P_y\). Any data that flows from \(x\) to \(y\) must be extracted from \(\Psi(x)\) and sent along with the edge \((x, y)\).
There may be several edges going out from $x$ that are all released at the same time. If more than one pair of released edge and accompanying data have to be sent to some processor, they are consolidated into one message to that processor. The edges $(x, y_i)$ going out of node $x$ are classified according to the processor $P_i$ which owns node $y_i$, for different $i$. The set of destination processors is then $D = \bigcup P_i$. For each destination processor $P_i$, the following message packet is constructed. One part of the packet consists of all the edges $(x, z)$, $II(\Psi(z)) = P_i$. The second part is the block $\Psi(x)$. We choose to send the entire block instead of just the the portions of the block $\Psi(x)$ required by the edges for the following reasons. Firstly, there are overheads involved in unioning the block portions in the sending processor $P_x$. Secondly, each destination processor has to pay the added cost of address arithmetic and storage manipulation in order to break up a received union-of-portions of a block into the respective updating portions that participate in different local tasks. Lastly, sending appropriate union-of-portions of $\Psi(x)$ to different destinations requires that a new message packet be created for each destination processor, which further increases the total communication cost. The cumulative overheads involved in these are typically greater than the additional cost of sending the entire block instead of a sub-portion.

The scheduling pre-pass algorithm is shown in algorithm Pre_schedule of Figure 6.4.

The pre-pass produces four sets of information on each processor: $fetch\_set$, $send\_set$, $local\_set$ and $task\_list$. The $fetch\_set$ is a list of the off-processor blocks that arrive at the processor during the scheduling process, in the order in which they are received. Each member of the set is a tuple of the form $<blk, size, deps>$, where $blk$ is the global unique number of the block that is fetched, $size$ is the block size computed from the AVR of the block, and $deps$ is the number of local tasks in which the block is a participant. Each member of the $send\_set$ is a tuple of the form $<blk, p1, p2, \ldots, pk>$ where $blk$ is the global unique number of a local block that is sent to the processors $p1, p2, \ldots, pk$. The set is ordered according to the sequence in which the blocks are sent off-processor during the scheduling process. The $local\_set$ is simply the AVR for all the locally mapped blocks. The $task\_list$ is a complete ordering of the tasks on
algorithm Pre_schedule
inputs: Local BCR /* part of BCR local to me */
        Functions Ψ, II, and Γ
return: task_list, fetch_set, send_set, local_set

done ← 0, ready-q ← empty, exec-q ← empty
foreach (node x s.t. predI(x) = 0 and predII(x) = 0) do
    add x to ready-q
end foreach

while (done < myblks) do

1. x ← node at the front of ready-q
   delete x from ready-q
   if (x is not a designated node) then
      foreach edge (x, y) do
         call release_edge(x,y)
         if (predI(y) = 0 and predII(y) = 0) then add y to ready-q endif
      end foreach
   else
      done ← done + 1
      foreach edge (x, y) do
         if (II(Ψ(y)) = me) then
            call release_edge(x,y)
         if (predII(y) = 0 and predII(y) = 0) then add y to ready-q endif
         else
            add edge (x, y) to message for II(Ψ(y)), D ← D ∪ II(Ψ(y))
         endif
      end foreach
      foreach (P ∈ D) do
         add Ψ(x) to message for P and send message to P
      end foreach
      append < Ψ(x), D > to send_set
   endif

2. L ← list of tasks at front of exec-q, delete these tasks from exec-q
   foreach (τ ∈ L) do
      append Γ(τ) with appropriate calling arguments to task_list
   end foreach

3. while (there is a message in the message buffer) do
   B ← block retrieved from message, relset ← set of to-be-released edges
   foreach ((x, y) ∈ relset) do
      call release_edge(x,y)
      if (predI(y) = 0 and predII(y) = 0) then add y to ready-q endif
      append < B, sizeof(B), |relset| > to fetch_set
   end foreach
endwhile

endwhile

Figure 6.4: Main procedure to schedule a BDAG.
the processor, in the sequence in which they were executing during scheduling. Each
member of the list is a program module, with the appropriate arguments with which
the module is called for execution.

The variable $myblks$ is the number of blocks allocated to a processor. The initial-
ization step (0) adds all independent nodes to the $ready.q$. A node is independent if
there are no type I or type II edges coming into it. The main while loop iterates until
the computations to all local blocks are completed, and the local blocks are sent to the
processors that need them for updates. Each iteration of the main loop is divided into
three steps: (1) releasing the edges going out from the next ready node, (2) extracting
the next set of executable tasks and (3) clearing messages in the message buffer.

In step 1, the node $x$ with highest priority is extracted from $ready.q$. If $x$ is not a
designated node, any edge going out of $x$ is an intra-block edge, and can be released
locally. During the process of releasing edges, nodes for which all precedence constraints
are satisfied are added to $ready.q$ in order of priority. If $x$ is a designated node, any edge
going out of $x$ is an inter-block edge, and the node is ready when all the computations
to the corresponding block, $\Psi(x)$, have been executed. A ready node, $x$, is “done”, as
soon block $\Psi(x)$ is sent to the processors that need it for updates. The foreach loop
in the else part goes through each of the edges out of $x$. An edge is released locally
if the destination block is mapped to this processor, identified by the variable $me$. If
the destination block is in a different processor, the edge needs to be released on that
processor, and the block $\Psi(x)$ has to sent to that processor for updates there. The next
foreach loop does all the “sending-out” communication.

In step 2, the list of tasks that update the block with highest priority is extracted
from $exec.q$. The numerical computations for these tasks are not performed. For each
task, the program module to which it is mapped, along with the appropriate calling
arguments, is appended to $task.list$.

In step 3, all the incoming messages are cleared. Each message is composed of an
updating block and a set of edges to be released on this processor. These edges are
released, and the block is stored until it is no longer required for any updates. Each
edge corresponds to a different task which requires the block $B$. Thus, the number of
tasks that use $B$ is given by the number of edges in the \textit{reset}, which is stored as the entry \textit{deps} in the member for block $B$ in the \textit{fetch.set}.

6.2.2 Parallel static scheduling using primitives

Once the pre-pass is done, the static scheduling code required to be written by the user is fairly simple and looks like a sequential program. In Figure 6.5, we prepend the call to the pre-pass using the primitive \texttt{SET.UP}, to the static scheduling code. The code shown in that figure is generic for any algorithm that has been partitioned and allocated, using the BCR, and the mapping, II of tasks and blocks to processors. The

\begin{verbatim}

\texttt{task.list} ← \texttt{SET.UP}(BCR,II)

\textbf{while} (\texttt{task.list} not empty) do
  \texttt{τ} ← \textbf{front of} \texttt{task.list}
  delete \texttt{τ} from \texttt{task.list}
  if (\texttt{τ} is a compute task) then
    \textbf{foreach} (participating block $B$) do
      \texttt{GET.BLOCK}($B$)
    \textbf{end foreach}
    execute \texttt{τ}
  else /* it is a communication task */
    $B$ ← block to be sent out
    \texttt{READY.BLOCK}($B$)
  endif
\textbf{endwhile}

\end{verbatim}

Figure 6.5: Distributed algorithm with primitives.

The main \textbf{while} loop iterates over the \texttt{task.list}, one task at a time. A task can either be a computational task or a communication task. If it is a computational task, the updated block and each of the updating blocks are made available through \texttt{GET.BLOCK}. Of these blocks, the non-local ones require inter-processor communication to fetch them from their respective owning processor. \texttt{GET.BLOCK} hides this data distribution, and therefore the communication, from the user. If the task is one in which a block is to be sent to depending tasks in which it participates, the primitive \texttt{READY.BLOCK} is
called. Again, communication to destination processors is hidden from the user. The necessity and functionality of the primitives is discussed next.

6.2.3 Primitives for scheduling

In an unstructured data-driven algorithm, the activation of computational work is driven by the availability of data. The correct parallelization of such an algorithm, by scheduling computational work and communication messages while obeying all dependency constraints and preventing message buffer overflow, requires sophisticated management of several data structures and sufficient knowledge of the architecture. The scheduling pre-pass algorithm of Figure 6.4 is a case in point. Writing a correct parallel program for such an algorithm, therefore, is a very difficult task for the general user. The primitive SET_UP takes the burden of scheduling away from the user. It calls the scheduler, instead, which generates the static task schedule and communication profile. Thus, SET_UP is included in the preprocessing step, along with the partitioning and allocation. The scheduler has to do the management of data structures, but this can be done more efficiently by a systems programmer, who is aided by an intimate knowledge of the architecture. Besides, the cost of scheduling is now a one-time preprocessing expense, and may be amortized by using the static schedule several times.

During the subsequent execution of the algorithm, the user simply walks through the complete schedule of computation and communication on each processor, without having to manipulate any data structures for scheduling. The computations are localized, and the user can deal with them as on a single processor machine. However, using a communication schedule to conduct message passing still needs an understanding of the capabilities and limitations of communication in the architecture. Correct and efficient message-passing is a difficult task, and the user should not be expected to deal with it. The primitives GET_BLOCK and READY_BLOCK provide a communication interface to the user. They use the communication schedule from GET_BLOCK to conduct optimized message-passing at execution time.
All these primitives have been implemented for the column sparse Cholesky factorization on the iPSC/860 [76]. To explain the functionality of the primitives in detail, we first describe the structures used in interface between the program and the primitives. We then describe the functional details of each primitive.

The program-primitives interface uses the AVR representation for blocks, described in Section 3.2.1. Figure 6.6 gives the C code for these structures.

```c
typedef struct ad {
  char *ptr;
  int size;
} ATTR_DATA, *AD_LIST;

typedef struct bc {
  AD_LIST all_attr;
  int numattrs;
  AD_LIST all_data;
  int numdata;
} BLOCK_CHAR, *BC_LIST;
```

Figure 6.6: AVR structures.

**Primitive SET_UP**

```c
int *SET_UP(BCR, II)
```

The primitive SET_UP is first called from the user’s program to generate a task schedule and set up the appropriate environment for carrying out the communication during numerical factorization. To achieve this, the primitive calls the scheduler. The primitive returns the ordered task list in the form of an array of integers comprising of the task tuples. The `fetch_set` and `send_set` are used to set up the necessary structures for efficient use by by READY_BLOCK and SEND_BLOCK, respectively, during execution. The local_set is used to distinguish between local and remote blocks. It is also used by READY_BLOCK to construct message packets.

**Primitive READY_BLOCK**

```c
void READY_BLOCK(blk)
int blk;
```

When a block is available to update its dependents the primitive READY_BLOCK
is called. The parameter $blk$ is the global number of the block which needs to be sent to the processors which "own" its dependents i.e. the processors to which its dependents are mapped. The primitive first retrieves the attribute and data value information from the structures created in SET_UP, and packs them into a single message. The attributes precede the data in the message and the message carries sufficient additional information to enable the receiving processor to unpack the message back into attributes and data values. Then, to each processor in the send_list of this block, a message is sent. Note that the entire block is sent to each of the receiving processors, irrespective of the portion of the block actually required.

**Primitive GET_BLOCK**

BC_LIST GET_BLOCK($blk$)

int $blk$;

GET_BLOCK is called to obtain a block required before the execution of a task in the user code. The parameter $blk$ is the global number of the block to be obtained, and the structure pointed to by BC_LIST is passed back, filled in with the attributes and data values of the block. The primitive first checks whether the block is local or non-local. If local, it sets up the appropriate pointers to the attributes and data values stored by the SET_UP primitive. If non-local, it checks whether the block has already been received. If the block has not been received, it posts a receive for the block and waits until the block arrives. It then unpacks the message and sets up the appropriate pointers to the attributes and data values.

The attributes and values of the block are returned in the BLOCK_CHAR structure. It is assumed that every time a block is fetched using GET_BLOCK, it is used in one update. If the block is non-local, the number of local dependents (or updates) is decremented before returning from the procedure call. If the number of local dependents goes to zero, the space for the block is freed on the next call to GET_BLOCK.
6.3 Scheduling for block sparse Cholesky factorization

In this section, we illustrate the implementation of scheduling for block sparse Cholesky factorization. The scheduling pre-pass follows the strategy of the general BCR scheduling pre-pass of Figure 6.4. However, certain problem-specific changes are made in order to enhance the efficiency of the data structures. The distributed code using primitives is a simple specific instance of the general code described in Figure 6.5.

6.3.1 Scheduling pre-pass for block sparse Cholesky factorization

The scheduling pre-pass can be run in two modes. In the first mode, the BDAG is traversed without actually executing the numerical computations, thereby doing symbolic scheduling. In the second, alternative mode, the numerical computations are also performed for the factorization. All the priority queues are simple FIFO queues. The scheduling pre-pass for sparse block Cholesky factorization is shown in Figure 6.7. The code omits several low-level details of data structures and only presents a top-level view. The following data structures are used for scheduling: (a) scale_q, a queue of blocks to be scaled; (b) base_update_q, a queue of local blocks to be used in updating other local blocks; (c) base_mult_update_q : a queue of (base,multiplier) block pairs to be used in updating local blocks; (d) ready_q : a queue of blocks ready to be sent off; (e) wait_list : list of blocks that are waiting for one or more blocks to pair with for base-multiplier updates. The variable myblks is the number of blocks assigned to the processor. A block is independent if it has no updaters. The outputs task_list, send_set and fetch_set are described in Section 6.2.

The updates are divided into base updates and base-multiplier updates, as described in Section 5.4.4. In the symbolic mode, the computations associated with the scaling and update tasks are not carried out - the tasks are simply added to the output task_list at the appropriate time. Blocks are sent off to the processors in the order in which they become ready to be sent, so the priority is simple FIFO.
**Algorithm Pre_schedule_Cholesky**

```
done ← 0; task_list ← empty

0. foreach (independent block k) do add k to scale_q, ready_q end foreach

while (done < myblks) do

1. k ← delete_front(scale_q)
   if (not symbolic) then execute < k, scale > endif
   append < k, scale > to task_list

2. k ← delete_front(ready_q); D ← processors that have blocks dependent on k
   send k to all processors in D; append < k, D > to send_set; done ← done + 1
   if (k is a base block used to update local blocks) then
     add k to base_update_q endif
   if (k pairs with j to update local blocks and j ∈ wait_list) then
     add (k, j) to base_mult_update_q else add k to wait_list endif

3. k ← delete_front(base_update_q)
   foreach local block j that is updateable by k do
     if (not symbolic) then execute < j, Update, k > endif
     append < j, Update, k > to task_list
     if (all updates to j are done) then add j to scale_q, ready_q endif
   end foreach

4. while (there is an unreceived block k in the message buffer) do
   receive block k from message buffer; deps ← 0
   foreach (j s.t. ∃ a local block updateable by base-multiplier pair (k, j)) do
     if (j ∈ wait_list) then add (k, j) to base_mult_update_queue
     else wait_list ← wait_list ∪ k endif
     deps ← deps + 1
   end foreach
   foreach (local block j updateable by k) do
     if (not symbolic) then execute < j, Update, k > endif
     append < j, Update, k > to task_list
     if (all updates to j are done) then add j to scale_q, ready_q endif
     deps ← deps + 1
   end foreach
   append < k, sizeof(k), deps > to fetch_set
endwhile

5. while (base_mult_update_q not empty) do
   (k, j) ← delete_front(base_mult_update_q)
   foreach (local block i updateable by (k, j)) do
     if (not symbolic) then execute < i, Update, k, j > endif
     append < i, Update, k, j > to task_list
     if (all updates to i are done) then add i to scale_q, ready_q endif
   end foreach
endwhile
```

Figure 6.7: Scheduling pre-pass for block sparse Cholesky factorization.
6.3.2 Parallel static scheduling of block sparse Cholesky factorization using primitives

The parallel block sparse Cholesky factorization algorithm using primitives is shown in Figure 6.8. We assume that the primitive SET_UP has been called with the local BCR on each processor and the mapping, II of blockto processors. The returned task_list is used by the code. In this version, a block is sent out as soon as it is scaled. Note that

\[
\text{while (task_list not empty) do}
\]
\[
\tau \leftarrow \text{delete_front(task_list)}
\]
\[
\text{if } (\tau \text{ is } \langle B, \text{Scale} \rangle) \text{ then}
\]
\[
\text{GET_BLOCK(B)}
\]
\[
\text{execute scale(B)}
\]
\[
\text{READY_BLOCK(B)}
\]
\[
\text{else if } (T \text{ is } \langle B, \text{Update}, A \rangle) \text{ then}
\]
\[
\text{GET_BLOCK(A)}
\]
\[
\text{GET_BLOCK(B)}
\]
\[
\text{execute } \langle B, \text{Update}, A \rangle
\]
\[
\text{else if } (T \text{ is } \langle C, \text{Update}, A, B \rangle) \text{ then}
\]
\[
\text{GET_BLOCK(C)}
\]
\[
\text{execute } \langle C, \text{Update}, A, B \rangle
\]
\[
\text{endif}
\]
\[
\text{ endwhile}
\]

Figure 6.8: Parallel block sparse Cholesky factorization.

depending on whether the block is a column, triangle or rectangle, one of the twelve specific types of tasks listed in Table 5.1 is executed in each the “execute” steps in the code. Also, \(\langle B, \text{Scale} \rangle\) is only a dummy task when \(B\) is a rectangle. It is a no-op, whose only function is to indicate that rectangle is ready to be sent out.

6.4 Communication optimization

This section discusses communication optimization on a message-passing architecture which is characterized by the following communication cost behavior, when a message is sent from a source processor to a destination processor. The time to send the message is significantly lower when the user’s program on the destination processor has already
posted a receive in anticipation, after allocating the space required for the message, than when the user's program has not posted an anticipatory receive. In the former protocol, the sending processor's message can directly go to the program space at the receiving processor. In the latter protocol, the sending processor's message is stored in the system message buffer, and copied into the receiving processor's program space only when the receive is posted at some later point of time. This leads to overheads of system message buffer management and recopying. We refer to the first protocol as the direct protocol, and the second as the indirect protocol.

The Intel iPSC/860 supports the direct protocol. The Thinking Machines CM5 provides both blocking and non-blocking message passing functions [70, 71]. A non-blocking function returns as soon as the processor has announced its readiness to send or receive. The processor can then perform other work, while waiting for the other processor to announce its readiness. When both processors have announced readiness, they receive interrupt messages telling them to begin transmission. Apart from this, the CM5 provides store and fetch primitives that allow a program running on one node to unilaterally read or write to the memory of another node. The remote node does not directly participate in this process. These primitives are useful when the memory layout of remote processors is known, and the handshaking aspect of the general message-passing function is considered to incur unacceptable overhead. Thus these store and fetch primitives provide a direct protocol for message passing.

Our goal is to schedule the communication among processors using only the direct protocol, and thereby reduce the communication time on every processor by a significant margin. The direct protocol can be implemented without much effort for programs in which the communication is structured, as for instance, those in which communication is restricted to occur between every pair of computational iterations. In this case, there is a software barrier between iterations, at which the sending and receiving processors can carry out the communication in synchrony. However, in the applications we consider, the communication between any pair of processors is not simultaneously co-ordinated either by hardware or by barriers in the program. A receiving processor need not immediately use the message sent to it. This makes the implementation of the direct
protocol much harder.

To illustrate, consider the following naive method of implementing the direct protocol. Post all the receives for each processor before commencement of execution. This guarantees that any message sent from one processor to another will always be successfully received. There are severe limitations to this naive approach. First, there is always a limitation placed by the architecture on the number of receives that can be pending at any time, without being used by the program. Second, posting receives beforehand requires that we allocate buffer space apriori for all the incoming messages. For large applications the limited amount of available memory would place a limit on the number of pending receives, and would render this approach impractical.

To overcome these limitations and thereby make the message passing scalable, we propose an algorithm which implements the direct protocol by incrementally posting receives during the execution of the computations. In the rest of this section, we will only look at communication between a pair of processors, $P_1$ and $P_2$. The communication between any pair of processors is independent of the communication between any other pair as far as the optimization techniques discussed in this section go.

We first describe the communication model to which our results apply. The model assumes that there is a complete static ordering of the sends and receives available to the optimization algorithm on every processor. A scalable deadlock-free communication strategy is then presented, followed by an approximation which is easier to implement. Finally, we describe the advantage of using our communication optimization algorithm on the iPSC/860. We believe that the essential idea of the communication strategy can also be used to advantage on the CM5.

### 6.4.1 Communication model using a static schedule

We define a sequential process to mean a thread of serial execution in which there are three types of processing activities - computational tasks, sending messages out of the processor, and receiving incoming messages from other processors. We assume that these activities are all scheduled or completely ordered within the process. Each processor runs one sequential process and the processes communicate in an asynchronous
manner i.e. the processes are not co-ordinated either by hardware or by software barriers, in order to send a message from one process to another. Each of the activities of a process is handled by the cpu, and therefore contributes to the total process execution time. Since each processor runs a single process, we use the terms processor and process synonymously.

On each processor, there is a completely ordered list of tasks, which are executed in sequence. Each task needs certain blocks of data for execution. Either a block is available locally or it has to be obtained from another processor. When a block is to be obtained from another processor, space for the block is allocated by the process, a receive is posted for that block, and a handle for that receive is obtained from the system. A posted receive is cleared when the process retrieves the message corresponding to that receive, using the handle as a means of identification. This handle is then is released for reuse. A receive is pending if it has been posted.

A message is sent to a processor only when the sending processor is informed that the receiving processor has posted a receive for that message. This guarantees delivery of the message to the receiving processor. In other words, whenever a processor posts a certain number of receives, the sending processor is informed about this by sending it a message token. A message token is a word or two in length.

A communication schedule between a pair of processors $P_1$ and $P_2$ is feasible if and only if, for any $m$, the number of receives in the first $m$ messages on $P_1$ is at most equal to the number of sends in the first $m$ messages on $P_2$. It is easy to see that a contradiction of this condition leads to a physically impossible situation in which a message that is not sent from the sender is received on the receiving processor.

A send run on $P_i$ with respect to $P_j$ is an ordered set of consecutive sends from $P_i$ to $P_j$ without any intervening receive. A receive run is defined analogously. The send and receive runs are defined for a given pair of processors. The message passing on a processor, with respect to a given “pairing” processor to which it sends messages and from which it receives messages, is a sequence of alternating send and receive runs.
Specifically, the message passing on $P_1$ with respect to $P_2$ is denoted by the sequence

$$S_1^1, R_1^1, S_2^1, R_2^1, \ldots, S_u^1, R_u^1$$

and the message passing on $P_2$ with respect to $P_1$ is denoted by the sequence

$$S_1^2, R_1^2, S_2^2, R_2^2, \ldots, S_u^2, R_u^2$$

Any of $S_1^1$, $S_1^2$, $R_u^1$ or $R_u^2$ could be an empty run, but both $S_1^1$ and $S_1^2$ cannot be empty in a feasible communication schedule.

The length of a run $Q$ is the number of messages in that run, denoted by $|Q|$. We use the notation

$$\mathcal{M}^i = (|S_1^i|, |R_1^i|, |S_2^i|, |R_2^i|, \ldots, |S_u^i|, |R_u^i|)$$

to denote the sequence of lengths of the send and receive runs in processor $P_i$.

Let $S_i^j$ be the $i^{th}$ message sent by $P_j$ and let $R_i^j$ be the $i^{th}$ message cleared by $P_j$. A processor blocks on a send if it is waiting for a token from the receiving processor to guarantee that a receive for the send has been posted. This token releases the processor, which then completes the send. A processor blocks on a receive when it waits for the sender to send the corresponding message, and is released when the message arrives.

Two processors are in a deadlock if each processor blocks on a send or a receive, waiting to be released by the other. We assume that as long as there is at least one more message to be received on a processor, there is at least one pending receive in that processor at any given point in the execution.

### 6.4.2 A scalable deadlock-free strategy

In this sub-section, we only consider the following type of communication: on any processor, receives for messages are cleared in the order in which they are posted, which in turn is the order in which the messages are sent from the sending processor. Considering $P_1$ and $P_2$, the constraints are: $R_1^1 = S_1^2$ and $R_1^2 = S_1^1$.

**Claim 6.4.1**: In a feasible communication schedule, the only possible deadlock situation between a pair of processors is one in which each processor blocks on a send.
Proof:

Assume there is a deadlock situation in which \( P_1 \) blocks on a send, \( S^1_i \) and \( P_2 \) blocks on a receive, \( R^2_j \). First, note that \( i = j \), i.e. the receive on which \( P_2 \) is assumed to block is for the message that \( P_1 \) is trying to send. The receive obviously cannot be for a message that \( P_1 \) has already sent, i.e. \( i \neq j \). Also, the receive cannot be for a message that would be sent from \( P_1 \) after the message on which it is assumed to block, i.e. \( i \neq j \), since that would mean that \( S^1_j \) has been sent successfully, and \( P_2 \) cannot therefore block on \( R^2_i \). So \( i = j \). Since there is at least one pending receive in \( P_2 \), and \( P_2 \) is ready to clear the next receive, \( P_1 \) cannot block. Also, since the receives for messages in \( P_2 \) are cleared in the order in which the messages are sent from \( P_1 \), \( P_1 \) can send the message to \( P_2 \) and release both itself and \( P_2 \), which contradicts the deadlock assumption.

Next, assume that there is a deadlock situation in which each of \( P_1 \) and \( P_2 \) blocks on a receive. Let \( P_1 \) block on its \((m+1)\)st message and let \( P_2 \) block on its \((n+1)\)st message. Without loss of generality let \( m \leq n \). Let there be \( \lambda \) sends in the first \( n+1 \) messages in \( P_2 \). Since these sends have been completed, the \( \lambda \)-th message from \( P_2 \) must be available on \( P_1 \). A feasible schedule requires that in the first \( n+1 \), and therefore the first \( m+1 \) messages of \( P_1 \), there be at most \( \lambda \) receives. Specifically, the \((m+1)\)-th receive can thus be cleared by \( P_1 \) without blocking, and the deadlock assumption is contradicted.

To lead to our strategy for scalable deadlock-free message-passing, we start by attempting maximum reuse of message handles and memory space, by never exceeding more than one pending receive at any time. Only when a receive is cleared and the corresponding handle released, is a new receive posted. Every time a new receive is posted, the sending processor is sent a message token. Each processor starts by posting one initial receive, for the first message required on that processor.

Unfortunately, this maximum reuse strategy does not guarantee deadlock-free execution. Figure 6.9 illustrates a deadlock situation in a feasible communication schedule between \( P_1 \) and \( P_2 \). In that figure, the circled numbers are the lengths of the receive
runs and the oblique numbers are the lengths of the send runs. $M^1 = (2, 5, 3, 5, 2, 2, 1, 0)$ and $M^2 = (0, 2, 6, 2, 2, 1, 4, 3)$ This results in a deadlock situation with $P_1$ blocked in $S^1_3$ and $P_2$ blocked in $S^2_4$. Each processor starts with one initial receive, and after that, every time a receive is cleared in a receive run, a new receive is posted. At the end of $R^k_3$, $P_1$ has posted a total of 11 receives, and at the end of $R^k_3$, $P_2$ has posted a total of 6 receives. Before beginning run $S^1_3$, $P_1$ has sent a total of 5 messages. It sends the first message in $S^1_3$, and then blocks on the second send, waiting for $P_2$ to post more receives. $P_2$, on the other hand, has sent a total of 8 messages before beginning run $S^2_4$. It sends the first 3 messages in $S^2_4$, and then blocks on the fourth send, waiting for $P_1$ to post more receives. This is a deadlock situation. This example generalizes to the following condition for deadlock.

Let $T^k_i$ be the total number of receives posted at the end of $R^k_i$, $k = 1, 2$. The following result assumes at least one pending receive on either processor at any time.

**Claim 6.4.2** A deadlock, with $P_1$ blocking in $S^1_\alpha$ and $P_2$ blocking in $S^2_\beta$ occurs if and only if

$$T^{1}_{\alpha - 1} < \sum_{j=1}^{\beta} |S^2_j| \quad \text{and} \quad T^{2}_{\beta - 1} < \sum_{j=1}^{\alpha} |S^1_j|$$

**Proof:** The sufficiency of the condition is easy to see. When $P_1$ begins the send run $S^1_\alpha$ and $P_2$ begins the send run $S^2_\beta$, neither processor has sufficient number of posted receives to satisfy all the sends in the other processor’s send run. On the other hand, no fresh receives can be posted unless a send run is terminated. Therefore, both $P_1$ and $P_2$ block, causing a deadlock. Now we show the necessity of this condition by contradiction. Let $T^{1}_{\alpha - 1} \geq \sum_{j=1}^{\beta} |S^2_j|$ (the argument is symmetric for $T^{2}_{\beta - 1} \geq \sum_{j=1}^{\alpha} |S^1_j|$).
Then, $P_2$ will complete the run $S_β^3$ without blocking. Since, by Claim 6.4.1, there cannot be a deadlock with $P_2$ in a receive run, $P_2$ will complete $R_β^2$, and enter $S_β^2$. Set $β ← β + 1$ and we are back to the invariant in the claim. Continuing this process, eventually both $P_1$ and $P_2$ will complete all send and receive runs.

In the special case of a new receive being posted every time (and only when) a pending receive is cleared, the deadlock condition can be more specifically stated as

$$
λ^1 + \sum_{j=1}^{α-1} |R_j^1| < \sum_{j=1}^{β} |S_j^2| \quad \text{and} \quad λ^2 + \sum_{j=1}^{β-1} |R_j^2| < \sum_{j=1}^{α} |S_j^1|,
$$

where $λ^1$ and $λ^2$ are the number of receives posted before entering the first send run in $P_1$ and $P_2$, respectively. In other words, these are startup receives. If, in addition, each processor has exactly one pending receive at all times, $λ^1 = λ^2 = 1$. This condition holds for the example of Figure 6.9 for $α = 3$ and $β = 4$.

To get around the deadlock situation, we propose the following. Instead of posting a new receive when and only when a receive is cleared, we will (a) post receives in batches and send, as confirmation of these posted receives to the sender, one message token for the entire batch of messages; (b) post a fresh batch at the beginning of every send run i.e. just before sending the first message in any run. We first formalize this strategy and then show it guarantees deadlock-free message passing.

Consider the processor $P_1$ (a symmetrical argument may be made for $P_2$). Before sending the first message in any $S_i^1$, we post a new batch of receives consisting of the next $M_i^1$ messages. This batch size is sufficient to guarantee that the next send run, $S_{i+1}^1$ is entered. Inductively, each processor must eventually terminate execution. To quantify the batch size, we view the processors as moving “in step” with each other in the send and receive runs. In other words, $M_i^1$ and $M_i^2$ are influenced only by $S_i^1$, $R_i^1$, $S_i^2$ and $R_i^2$. Thus, the requirement that $P_1$ enter $S_{i+1}^1$ translates into the following conditions. (1) $P_1$ completes its send run, $S_i^1$, which implies that there are at least $|S_i^2|$ pending receives in $P_2$ when it enters $S_i^2$. (2) $P_1$ completes its receive run, $R_i^1$, which implies that when $P_1$ enters $S_i^1$, there are at least $|R_i^1|$ pending receives in $P_1$. Complementarily, when $P_1$ posts receives, it must ensure that $P_2$ can complete its send run $S_i^2$. Thus, we need that the total number of receives posted in $P_1$ at the beginning
of send run $S_i^1$:

$$\sum_{j=1}^{i} M_j^1 \geq \max(\sum_{j=1}^{i} |S_j^2|, \sum_{j=1}^{i} |R_j^2|)$$

We can use the lower bound to post the new batch of receive:

$$M_i^1 = \max(\sum_{j=1}^{i} |S_j^2|, \sum_{j=1}^{i} |R_j^2|) - \sum_{j=1}^{i-1} M_j^1$$  (6.1)

**Claim 6.4.3** The above strategy is deadlock-free.

**Proof:** By contradiction. Suppose a deadlock does occur. Then, by the deadlock condition of Claim 6.4.2 there exist $\alpha, \beta$ for which

$$T_{\alpha-1}^1 < \sum_{j=1}^{\beta} |S_j^2| \quad \text{and} \quad T_{\beta-1}^2 < \sum_{j=1}^{\alpha} |S_j^1|$$

which, in this case, is that

$$\sum_{j=1}^{\alpha} M_j^1 < \sum_{j=1}^{\beta} |S_j^2| \quad \text{and} \quad \sum_{j=1}^{\beta} M_j^2 < \sum_{j=1}^{\alpha} |S_j^1|$$

observing that the end of $R_{\alpha-1}^1$ is the same as the beginning of $S_{\beta}^1$. Without loss of generality, assume $\alpha \leq \beta$. By the strategy, we have that

$$\sum_{j=1}^{\beta} M_j^2 \geq \max(\sum_{j=1}^{\beta} |S_j^2|, \sum_{j=1}^{\beta} |R_j^2|) \quad \Rightarrow \quad \sum_{j=1}^{\beta} M_j^2 \geq \sum_{j=1}^{\beta} |S_j^1| \quad \Rightarrow \quad \sum_{j=1}^{\beta} M_j^2 \geq \sum_{j=1}^{\alpha} |S_j^1|$$

which contradicts the deadlock condition. \[ \square \]

Using this strategy for the example of Figure 6.9, with $M_i^j$ as given in equation 6.1, we obtain the following batch sizes. $M_1^3 = 3$, $M_1^4 = 7$, $M_3^1 = 2$, and $M_4^1 = 0$. $M_2^1 = 2$, $M_2^2 = 2$, $M_3^1 = 1$, and $M_4^2 = 2$. It may be verified that this set of batches of receive guarantees deadlock-free execution.

From equation 6.1, the maximum number of pending receives at any point of time in processor $P_1$ is simply $\sum_{j=1}^{i} M_j^1 - \sum_{j=1}^{i-1} |R_j^1|$.

In using this strategy, if there is a significant time lag between the sends in one processor and the receives in the other processor, we would tend towards making the processes more and more synchronous. The sender blocks, waiting until the receiver has guaranteed that a receive has been posted, which happens at the time that the receiver is
ready to actually use the received data. One way to do reduce this apparent synchronization effect is to post extra "priming" receives in the beginning, to act as a time buffer. Let the maximum allowable number of pending receives on a processor be $\mathcal{L}$. $\mathcal{L}$ can be decided based on the memory space available and the number of live message ids permitted at any time. Then, from equation 6.1 again, we have the new batch size satisfying the condition:

$$\sum_{j=1}^{i} M_j^1 - \sum_{j=1}^{i-1} |R_j^1| \leq \mathcal{L} \Rightarrow M_i^1 + \sum_{j=1}^{i-1} M_j^1 - \sum_{j=1}^{i-1} |R_j^1| \leq \mathcal{L}$$

### 6.4.3 Extension to windows of receives

In the previous sub-section we had assumed that the receives are cleared in the order in which the messages are sent from the sending processor i.e. $R_1^1 = S_2^2$ and $R_2^2 = S_1^1$. In this sub-section, we remove this restriction i.e. $R_i^2 \ne S_i^1$ and $R_i^1 \ne S_i^2$.

We assume that all messages arriving at a processor come in the order in which they are sent out from the sending processor. The slackening of the previous restriction then means that the arriving messages need not be used in the receiving processor in the order in which they arrive, which in turn means that the arriving messages may be cleared in an order which is different from that in which they are sent. For instance, let blocks $B_1$, $B_2$ and $B_3$ be sent in that order from $P_1$. In $P_2$, let the blocks participate in the following tasks in that order - $\tau_1 : < B_3 >$, $\tau_2 : < B_1, B_3 >$ and $\tau_3 : < B_2, B_3 >$.

The task $\tau_1$ needs $B_3$, $\tau_2$ needs $B_1$ and $B_3$, and $\tau_3$ needs $B_2$ and $B_3$. They are executed in the order $\tau_1$ followed by $\tau_2$ followed by $\tau_3$. Corresponding to this execution sequence, the blocks are used in the order $B_3, B_1, B_2$, which is different from the order in which they are sent from $P_1$. The important point of note in this setting is to ensure correct retrieval of messages, receives may no longer be posted one at a time. For instance, if the receiver posts a receive for $B_3$ first, and waits to clear this receive upon the arrival of $B_3$, it will wait forever. This is because $P_1$ will not get a confirmatory message token from $P_2$ that a receive for $B_1$, the first message out of $P_1$, has been posted. So $P_1$ blocks, waiting for this token, and there is a deadlock.

It is easy to see that in this new situation, simply posting receives in batches of
the size given in equation 6.1 does not ensure deadlock-free message-passing. The new necessary condition is that at any point of time on a processor, the number of pending receives, \( k \), is explicitly chosen in such a way that the next \( k \) blocks required are in these \( k \) receives. In the example given above, \( P_1 \) cannot have either one or two pending receives. It must post all three receives in one batch and send a confirmatory message token for the entire batch, to \( P_2 \), so that the three messages can be sent by \( P_1 \) in the order \( B_1, B_2 \) and \( B_3 \) without blocking.

We define a set of sends to match a set of receives if and only if, for every message type in the send set, there is a matching message type in the receive set. As an example, consider message types 47 through 54 sent in that order from \( P_1 \) to \( P_2 \). Suppose they are needed in the order 47, 49, 50, 48, 52, 51, 53, 54 in \( P_2 \). Then, if \( P_2 \) posts receives for the first five blocks that it needs in one batch and blocks until it receives all of them, it will get the message types 47 through 51, but blocks on the pending receive for 52. On the other hand, \( P_1 \) blocks on the send of message type 51 since it does not have a confirmation from \( P_2 \) for this message type. This is a deadlock situation. To avoid this blocking, \( P_2 \) needs to post at least six receives to match the next six sends from \( P_1 \). We will refer to such a batch as a window of receives.

In the following definition of a window, the notation \( W_i \) stands for the \( i'th \) window, an ordered set of messages. \( |W_i| \) is the length of a window \( W_i \), or the number of messages in it.

**Definition 6.4.1**: Let the ordered set of cleared messages in \( P_1 \) (analogously in \( P_2 \)) be divided into consecutive, non-overlapping windows \( W^1_1, W^1_2, \ldots, W^1_l \). Let \( l = \sum_{j=1}^{l-1} |W^j_1| \) and \( k = |W^1_i| \). Then, \( |W^1_i| = k \) is the smallest integer for which the messages \( S^2_{i+1}, S^2_{i+2}, \ldots, S^2_{i+k} \) match the receives in \( W^1_i \).

The restriction of Section 6.4.2 is a special case for which every window is of size one.

To illustrate the division of messages into windows, here is another example. Let \( P_1 \) send to \( P_2 \) the messages numbered 1 through 12, in that order. Let \( P_2 \) clear the messages in the order \( < 1, 3, 2, 5, 4, 6, 7, 9, 10, 8, 12, 11 > \). Then, the windows of receives in \( P_2 \) are \( < 1 >, < 3, 2 >, < 5, 4 >, < 6 >, < 7 >, < 9, 10, 8 >, < 12, 11 > \), in which each
tuple is a window, and represents an ordered set of message types.

Windows place an additional constraint on the way receives may be posted. They are the atomic units of receives i.e. when a receive is posted for a certain message type \( m \), receives for all the other message types in the window to which \( m \) belongs must also be posted at the same time.

We extend the deadlock-free strategy presented in Section 6.4.2 to include windows. Let \( P_1 \) be at the point just before the beginning of send run \( S_i^1 \). Let \( k \) be the smallest integer for which

\[
\sum_{j=1}^{k} |W_j^i| \geq \max\left(\sum_{j=1}^{i} |S_j^i|, \sum_{j=1}^{i} |R_j^i|\right)
\]

Let \( l \) be the number of windows for which receives have been posted so far. Then, the number of receives to post at the beginning of send run \( S_i^1 \) is

\[
M_i^1 = \sum_{j=l+1}^{k} |W_j^i|
\]  
(6.2)

It is easy to see that this strategy guarantees deadlock free execution, since it preserves the deadlock-free condition which was satisfied by the strategy of equation 6.1.

### 6.4.4 An approximation to the deadlock-free strategy

An algorithm which implements the strategy given above would need each processor to store the send runs of all processors. To avoid this storage cost and to make the book-keeping task less cumbersome, we propose alternative strategies which reduce the coding and storage complexity by possibly decreasing the reusability of message ids.

We introduce two changes to the strategy. First, we approximate the length of send and receive runs by the maximum length of a send or receive run, respectively. Second, instead of posting a fresh batch of messages at the end of every receive run, we post a fresh batch at any point in each run of a set of selected receive runs, but not necessarily in all runs. For a processor considered as a receiver, the first change does away with storage of send runs of all processors, and of the receive runs of that processor. The second change does away with both the book-keeping and the storage needed to identify the end of receive runs, which is the same as identifying the beginning of send runs. We first assume that all windows are of unit size.
Essentially, the new strategy is to post $M_i$ receives at the very beginning of execution, and subsequently, to post a fresh batch of $M_i$ receives when the number of pending receives dwindles to $\Delta_i$. This could happen at any point in any receive run. The main question is how to choose $M_i$ and $\Delta_i$ to guarantee deadlock-free execution. Before we answer this question, we introduce some more terminology. Let $U^k_i = \max_i (|S^k_i|)$ and $U^k_l = \max_j (|R^k_j|)$. In other words, $U^k_i$ is the length of the longest send run in $P_k$ and $U^k_l$ is the length of the longest receive run in $P_k$.

As a first cut, choose $M^1_i = \max(U^2_i, U^1_i)$, $M^2_i = \max(U^1_i, U^2_i)$, and $\Delta^1_i = \Delta^2_i = 0$. A fresh batch of receives is posted when the previous batch is completely cleared, and batches are of constant size. But this strategy is not deadlock-free, as shown in the following example. Take $|W_i| = 1$ for all $i$ in both $P_1$ and $P_2$. Let $\mathcal{M}^1 = (0, 2, 2, 1, 2, 5)$ and $\mathcal{M}^2 = (3, 1, 4, 2, 1, 1)$. Then $U^1_s = 2$, $U^2_s = 4$, $U^1_r = 5$, $U^2_r = 2$. So $M^1_i = \max(U^2_i, U^1_i) = 5$ and $M^2_i = \max(U^1_i, U^2_i) = 2$. Processor $P_1$ initially posts 5 receives and processor $P_2$ initially posts 2 receives. It can be seen that $P_1$ blocks on the first send in $S^1_3$ and $P_2$ blocks on the third send in $S^2_2$, thus causing a deadlock. At this point, there are 2 pending receives in $P_1$ and 1 pending receive in $P_2$.

Clearly, the problem is in the choice of $\Delta^1_i = \Delta^2_i = 0$. We now show that a choice of $M^1_i = \Delta^1_i = \max(U^2_i, U^1_i)$, and $M^2_i = \Delta^2_i = \max(U^1_i, U^2_i)$ will guarantee deadlock-free execution. A fresh batch of receives is posted when the number of pending receives goes down to the above pre-determined constant, and the batches are of constant size.

**Claim 6.4.4:** The above strategy is deadlock-free.

**Proof:** The proof is trivial. Consider $P_1$ (the argument is symmetrical for $P_2$). It is easy to see that the number of pending receives at the beginning of any send run $S^1_i$ is at least $\max(U^2_i, U^1_i)$. Since, by definition, $\max(U^2_i, U^1_i) > \max(|S^2_i|, |R^1_i|)$, $P_1$ will complete the send and receives runs $S^1_i$ and $R^2_i$, respectively, and enter the next send run $S^1_{i+1}$. Thus, inductively, $P_1$ will execute to completion $\blacksquare$

When the windows are not assumed to be of unit size, the above strategy has to be modified as follows. When a new batch is posted, it has to ensure that the batch size does not "split" a window. In other words, if the batch size is such that for a particular
window, the trailing \( \eta \) messages are not included in the batch, then these remaining \( \eta \) messages are added to the batch. The proof to show that this strategy is deadlock-free is the same as that for Claim 6.4.4.

6.4.5 Optimization on the iPSC/860

We briefly discuss the message passing protocols on the iPSC/860 and show how our communication optimization algorithm is applicable to reduce latency cost by a significant margin.

Message passing calls on the iPSC/860 are classified into asynchronous and synchronous types. These are not to be confused with the asynchronous communication discussed above. All communication in our application is asynchronous, but the communication may be carried out by using the iPSC/860 synchronous or asynchronous message-passing calls. Messages of both types are characterized by a length, a type and an id. A message is uniquely identified by its message type, and is supplied by the user. The id is an identifier used to check for the completion of asynchronous messages, and is generated by the system.

A *synchronous send* (called `csend`) is one in which a process executing the send waits until the send is complete. This is referred as a *blocking* send. Completing the send does not guarantee that the message has been received at the destination processor. It only means that the message has left the sending process. A *synchronous receive* (called `crecv`) is one in which the process executing the receive waits until the message arrives in the specified buffer. This is referred as a *blocking* receive.

An *asynchronous send* (called `isend`) or *asynchronous receive* (called `irecv`) does not block. It returns a unique message id, which is not reused until released. There are two calls to check whether an asynchronous operation has completed - `msgdone` and `msgwait`. A msgdone call, with an id, returns a 1 if the asynchronous operation for that id has completed, and returns a 0 otherwise. A msgwait call, with an id, *blocks* until the the asynchronous operation for that id has completed.

If a message arrives at a processor, and a receive has not been issued for that message by the receiving process, the message is copied into a system buffer. When the receive is
eventually issued, the message is copied from the system buffer into the process buffer. This situation is illustrated in Figure 6.10. In that figure, process $P_1$ sends a message to $P_2$ using csend, at time $t_1$. Since $P_2$ has not yet posted a receive for that message, it is copied into the system buffer. When $P_2$ does issue a crecv for the message at time $t_2$, it is copied from the system buffer to the process buffer. Also note that the shaded areas in $P_1$ and $P_2$ indicate the time period during which the processor is busy executing the csend or crecv, respectively. Had $P_2$ issued a crecv at a time before $t_1$, the message would have been directly copied into $P_2$’s application buffer.

Figure 6.11 illustrates an asynchronous receive. In that figure $t_i < t_j$ for $i < j$.

Figure 6.11: Asynchronous receive.
Process $P_2$, at time $t_1$, issues an irecv for some message type, say $M$, which returns an id of 5. When the irecv is issued, the process $P_2$ has to allocate buffer space to receive the message. $P_2$ then continues with other work. At time $t_2$, process $P_2$ issues a msgdone for id = 5. Since the message corresponding to this id has not yet arrived, msgdone returns a 0 and $P_2$ continues. Process $P_1$, at time $t_3$, sends a message of the type $M$ to $P_2$, using a csend. The message directly goes into the message buffer of process $P_2$ at time $t_4$, bypassing the system buffer. At time $t_5$, $P_2$ issues a msgwait for id = 5, and clears the message id for reuse.

On the iPSC/860, a message of length greater than 100 bytes follows a three-trip communication protocol. The sending processor first sends the length of the message to the receiver. The receiving processor checks if the corresponding receive has already been posted. If not, it allocates space for the message in the system buffer. It then sends an acknowledgement to the sending processor. Upon receiving the acknowledgement, the sending processor transmits the message. This three-trip protocol leads to an increase in the set-up cost at the sender, and the receiver, in turn, experiences this delay. A force type message on the iPSC/860 bypasses normal control flow mechanisms, including book-keeping involving the system message buffer. It assumes that the receiving processor has allocated space and sends the message off in a single trip. If the destination processor has not issued a receive, the message is lost. It has been shown in [10] that the time in microseconds to communicate a message of length $m$ bytes over distance $d$ on the iPSC/860 is $t = 95 + 0.394m + 10.3d$ using force type messages. The unforced messages take the same time if the message length is at most 100 bytes; for larger lengths, the time is $t = 164 + 0.398m + 29.9d$. It can be seen that force type messages result in significant savings in the latency or set-up cost, and hence the overall communication time, as compared to unforced messages.

The direct protocol is implemented as a force type message on the iPSC/860. Applying our communication optimization algorithm would result in using only force type messages. The experimental results in Chapter 7 show that our optimization results in more than 50% savings in communication cost for block sparse Cholesky factorization on a variety of test matrices.
6.5 Conclusions

We have proposed a new, two-stage task and communication scheduling strategy, which is applicable to any unstructured data-driven algorithm. Generating a static schedule using a one-time scheduling pre-pass, and repeatedly using this static schedule with execution-time communication optimization, is a powerful mechanism towards attaining high performance. The tool is made easy to use by additional support in the form of a set of primitives that can be called from the user program. The task of coding is simplified to building a set of program modules for the computational tasks; the user need have no knowledge of the data distribution or the architecture. One drawback of this strategy is that additional space is required on each processor to store is task list. Another drawback is the time for the scheduling pre-pass, which is added to the overall preprocessing time. In the next chapter, we show that for sparse Cholesky factorization, the preprocessing cost can be amortized over two or three iterations only of the numerical factorization.

We have also implemented a communication optimization algorithm for blocks based on windows discussed earlier. We show that the gains in communication are significant, and well worth the optimization effort. The communication optimization code is again embedded within the GET_BLOCK and READY_BLOCK primitives. These primitives have not been implemented for block sparse Cholesky factorization. However, since an optimized communication code and a scheduler have been implemented, it is a simple matter to build the interface for the primitives around the implementation.
Chapter 7

Experimental evaluation of distributed block sparse
Cholesky factorization using SHAPE

Our methodology for partitioning and scheduling unstructured asynchronous computations on large-scale multiprocessors has been applied to block sparse Cholesky factorization. In this chapter, we present extensive experimental results to show both the efficiency of the methodology and its effectiveness in delivering high performance.

The performance of a parallel algorithm is affected by several factors that are closely related to each other. To understand the effect of these factors, both individually and as a coupled system, we need to generate and interpret performance figures for an extensive set of tests, with fast turnaround time. In order to conduct tests for sparse Cholesky factorization, we have built a run-time tool called Sparse Hybrid Automatic Parallelization Environment, or SHAPE, for short. It provides the required infrastructure to experiment with a variety of structured and unstructured matrices, with very little effort on the part of the user. This infrastructure provides parametric control over the partitioning and scheduling steps, and an exhaustive output of performance figures to help the user analyze performance.

We first give an overview of SHAPE and describe its functional composition. We then briefly discuss performance issues and describe the performance measures used to quantify our experimental results. The world of distributed memory message-passing architectures is currently populated by an array of machines with wide-ranging performance characteristics, including different generations of machines from the same manufacturer. In such a diverse environment, it is important to present and interpret performance figures on real machines keeping in mind several machine and problem peculiarities. Towards this end, we describe the testing environment in detail, and
highlight the implementation-specific features that affect performance. We present and analyze the uniprocessor and multiprocessor performance for a variety of test matrices, measuring performance in terms of the computation, communication, idling and overhead costs. The preprocessing cost of partitioning and scheduling in SHAPE is measured, and compared to the numerical factorization cost.

Throughout the chapter we compare the block factorization code with a column-based factorization code which is derived from a distributed fan-out algorithm, with columns being wrap-mapped to processors. We refer to this as the column-wrap code. We also compare the block code with a distributed multifrontal code. We conclude the chapter by outlining the limitations of the current implementation and propose ways to improve performance. In order to validate our programs, we have compared the numerical values of the factor $L$ produced by SHAPE against those produced by the column-wrap code, and found them to match.

7.1 Overview of SHAPE

SHAPE, standing for Sparse Hybrid Automatic Parallelization Environment, is a tool that we are developing with the long-term objective of partitioning, scheduling and conducting detailed investigation of the parallel performance of sparse matrix computations. At present, it is possible for us to perform a thorough study of sparse Cholesky factorization on large, structured and unstructured sparse matrices. In the current design, SHAPE functions both as a run-time preprocessor prior to the numerical factorization, and as an execution-time support tool that can be used by the numerical factorization via a simple interface.

The input to SHAPE is the symbolically factored lower triangular matrix $L$. As a partitioner and scheduler, SHAPE the output of SHAPE is a schedule of tasks, in the form of a total ordering on each processor, and a complete inter-processor communication profile, including the ordering of communication within each processor. Using this profile, SHAPE is able to optimize the execution-time communication during numerical factorization. All the functions of SHAPE are accessible to the user program by means
of an interface consisting of three primitives, which also hide the data distribution and architectural details from the user. These primitives are described in Section 6.2.3. Figure 7.1 gives a top-level view of how SHAPE fits into the step-wise process of solving the sparse linear system $Ax = b$. The dotted box enclosing both SHAPE and the

![Diagram of SHAPE process]

**Figure 7.1: Solution of a sparse linear system using SHAPE.**

numerical factorization step indicates that the numerical factorization code interacts with the communication profile set up by SHAPE for efficient message passing.

Functionally, SHAPE operates in four stages: pre-partitioning, partitioning, allocation and scheduling. The interaction among these four functions is shown schematically in Figure 7.2. The role of each of these stages, including the role of the parameters, is described in detail in Chapters 5 and 6. Here we give a brief outline of each to show how the stages are integrated into SHAPE. In the rest of the chapter, we use $L$ to denote the lower triangular symbolic factor. When we make a reference to the numerical factor $L$, as the output of the numerical factorization step, we will make an explicit reference to it.
The input to the pre-partitioner is the lower triangular symbolic factor $L_\alpha$. It also accepts the parameter $W$, the minimum acceptable cluster width, whose function is described in Section 5.3.3. The output of the pre-partitioner is a division of $L$ into clusters of columns, and within each multiple-column cluster, the naturally occurring triangular and rectangular shaped dense blocks. The partitioner uses the grain size parameter $G$ to divide the naturally occurring blocks in each multiple-column cluster into unit dense blocks of non-zeros. The parameter $G$ is described in Section 5.4.5 and Section 5.5. The partitioner produces blocks that are either single sparse columns or dense triangles or dense rectangles, formats the input data in this block form, and outputs these block partitions. It computes the dependencies among these blocks, and creates the BCR, the Block Computation Representation, for the blocks and the dependencies. It also generates the inter-cluster dependency graph, the CDAG. The BCR is described in Section 3.2, and the CDAG is described in Section 6.1.2. The allocator uses the CDAG and the BCR, with the number of processors, $P$, as a parameter, to compute...
an assignment of blocks to processors. The scheduler then executes the BDAG using this block-to-processor assignment and the owner-computes rule. It determines a task schedule for each processor and the overall time and space profile of the inter-processor communication.

To assist the user in experimenting with different partitioning and scheduling strategies, SHAPE generates exhaustive information including number of messages, communication volume and load balance information. Currently, it does not give time estimates of computation and communication due to the difficulty of modeling these for real machines. The computation and communication times vary widely with both the type of block or task involved, and the computational and communication characteristics of the architecture. This is discussed in greater detail in Section 7.7. However, if the computation and communication cost functions for the target architecture can be modeled with reasonable accuracy, then SHAPE can use these cost functions to give an estimate of the total execution time, with a breakdown of the time for each additive component of the total time.

7.2 Performance measures

Order of complexity analysis are of limited use in predicting the behavior of distributed sparse Cholesky factorization algorithms, with the result that experimental results play a crucial role in understanding their performance. Recently, there has been a spate of articles regarding the reporting of experimental results to demonstrate the performance of parallel algorithms [6, 8, 31, 49, 9, 46]. The two primary issues addressed in these articles are: the metrics to be chosen to evaluate parallel performance, and the guidelines to be followed to report experimental results. These are especially important for sparse Cholesky factorization, which is widely regarded as a problem that is hard to parallelize on distributed memory message passing machines, and for which, experiments need to be conducted and results presented in a manner that offers as much insight into the problem-solving process as possible.

The most well-known metric to evaluate parallel performance of an MIMD code
is speedup. There are several definitions of speedup, summarized by Barr and Hickman [8]. However, for sparse Cholesky factorization, reporting performance using a single speedup number is hardly enlightening, and in fact, may be quite misleading. There are two reasons for this.

First, given that the input matrices and therefore the dependencies are highly unstructured, it is extremely hard to come up with a measure that quantifies the parallelism that can be exploited in the problem instance on the given architecture. Therefore, a poor speedup figure for a parallel algorithm does not necessarily imply that the algorithm is inefficient - it may be due to an inherent limitation in the exploitable parallelism in the problem.

Second, as pointed out by Gustafson [31], the conventional speedup figure that measures reduction in execution time with increase in the number of processors penalizes faster absolute speed. For example, consider the iPSC/2 and the iPSC/860 machines. Helin and Berrendorf [35] have studied the computation and communication properties of the iPSC/860 and compared these with the iPSC/2. Their study shows that a single iPSC/860 processor is nearly 50 times as fast as a single processor of the iPSC/2, on the average. Because of this computing speed, the startup overhead or latency is lower on the iPSC/860 than on the iPSC/2. For messages greater than 100 bytes long, the startup time is $175\mu s$ on the iPSC/860, while it is $700\mu s$ on the iPSC/2. The transmission time per byte is the same for both machines. The communication to computation ratio for the iPSC/860 is approximately 10 to 50 times as much as that for that for the iPSC/2, depending on the length of the messages transmitted. Thus, for the same problem instance and the same parallel algorithm running on the iPSC/2 and iPSC/860 machines, one would tend to see much better speedup figures on the iPSC/2, even though, in terms of absolute speed, the algorithm would run much faster on the iPSC/860. A corollary to this is the observation that it becomes harder to parallelize a problem as the grain size, or the communication to computation ratio of the architecture, increases.

For block sparse Cholesky factorization, we evaluate performance on the basis of the time spent on a processor for all the different activities performed in the course of
the factorization. To understand this performance, we use quantitative and qualitative measures of the factors that determine the times for these activities. The total execution time on a processor is made up of four components: the computation time, the communication time, the idle time, and the overhead time. The parallel execution time is the maximum of the execution time over all processors, or, the time taken by the processor that finishes last. The computation time is the time spent by an individual processor in performing floating point operations that lead directly to the computation of the Cholesky factor. The communication time is the time spent in the assembling and disassembling of the messages, time spent in the data movement between memory and message buffers, and any system overheads (e.g., context switching) related to message-passing. The overhead time is the time spent in the bookkeeping activities associated with scheduling computations and communication. Finally, the idle time is the "thumb-twiddling" time on a processor when none of the above activities are being carried out.

On any given processor, the computation time is a function of the total number of useful floating operations assigned to that processor, the processor speed, and the effect of the data layout in utilizing the cache and the floating point unit. For a given machine, the effect of the data layout is in turn determined by the number of each type of block and the number of tasks of each type. Sparse columns use the cache less efficiently than dense triangles and dense rectangles. The fewer the sparse columns, the fewer are the tasks that operate on sparse columns, and the better is the cache utilization. Among the dense blocks, the more blocks there are with long vectors, the better is the utilization of the floating point unit. We quantify the effect of the data layout by relating it to the above factors. The communication time, for a given architecture, is a function of the number of messages sent and received, and the volume of data exchanged in satisfying the inter-processor task dependencies, which in turn are directly affected by the data partitioning strategies used. We measure the number of messages and data volume for all the experiments carried out, and show the relationship between the communication cost and the relative proportion of the number of dense blocks to the number of sparse columns. The idle time results from load imbalance
among processors. The load imbalance may be because of uneven distribution of work and/or because of the dependencies that lead to processor starvation. The idle time is determined by the sparsity structure of the matrix and by the effectiveness of the task scheduling strategies used. The time spent in overhead activities is mainly due to the bookkeeping required to schedule the computations and communication.

Each of these four components is tightly coupled with the other three. Minimizing any one of the components may adversely affect one or more of the remaining three components. In the rest of the chapter, we follow the guidelines outlined by Bailey [6] and Barr and Hickman [8, 9] to report the results from our experiments, starting with the following description of the testing environment.

7.3 Testing environment

We used SHAPE and a block sparse Cholesky numerical factorization code to conduct experiments on a suite of test matrices. In later sections, we compare the performance of the block code with a column-wrap code based on a static task schedule. In this section we highlight some specific features of the implementation of SHAPE, the block code and the column-wrap code that are essential to understand the performance of these implementations. We then describe the suite of test matrices on which we ran our experiments. All the programs are written in C. All experiments reported here were conducted on the Intel iPSC/860 at ICASE, NASA Langley Research Center. Each node of this multiprocessor system is a 40MHz i860 processor and has 8MB of main memory.

7.3.1 SHAPE: implementation

The pre-partitioner is a sequential program that runs on a single processor. Currently, it does not use the parameter \( W \); instead, \( W \) is used by the partitioner. The pre-partitioner identifies all the clusters in the input symbolic factor, regardless of their widths. The code works for any input matrix size by reading the input in batches of columns instead of loading the entire matrix at once. The size of each batch of columns
is determined by the amount of memory available on a processor.

The partitioner is a parallel program which runs on any number of processors, as desired by the user. It accepts the parameters $W$, $T$ and $R$. In the current implementation of the partitioner, $T$ and $R$ are the grain size for the triangle and rectangle, respectively, in number of non-zeros. We do not use the work requirement version of the grain size described in Section 5.4.5. The rectangular blocks are partitioned into horizontally long and flat shapes, instead of square, as described in Section 5.4.5. The partitioner code is flexible enough that this can be changed to square or vertically tall and thin shapes. Due to memory limitations, we had to make one change in the way the rightmost naturally occurring dense triangle in the matrix is partitioned. Typically, this triangle incurs the largest number of updates compared to all other triangles and rectangles. In some of the large matrices in our test suite, the computation of the dependencies for this triangle causes the processor which is responsible for that triangle to run out of memory space. To prevent this, we use the option of splitting the last triangle into several clusters, each of some fixed width which is controlled by a parameter. These clusters are distributed among several processors, so that the number of dependencies computed for the last triangle is distributed, and the processors to which these clusters are assigned do not run out of memory space.

The allocator is a sequential program. In the current implementation, we use the version of the allocator in which the CDAG is not used, see Section 6.1. Again, the problem with using the CDAG version is that there is not enough memory space to keep the CDAG in core when doing the allocation. The scheduler is a parallel program that runs on the same number of processors as the numerical factorization code. The details of the scheduler are given in Section 6.3. We have also implemented a parallel symbolic factorization code to compute the symbolic factor for some of the biggest test matrices.

7.3.2 Distributed block-based factorization: implementation

The parallel numerical factorization code uses the task list and the communication structure produced by the scheduler. The task list is stored in an array. The code
executes by following this ordered task list, and executing each task in turn. Each of
the blocks required in a task is checked for availability on the processor. If a block is
not available, the execution is suspended in a busy wait state, and is continued only
when the block is made available by receiving it from the owning processor. After a
task is executed, if the updated block has been factored, it is immediately sent to all
the processors that need it. The code then goes on to the next task in the task array.
We have not implemented the block primitives as yet.

The blocks are stored, and communicated among processors using the AVR format
described in Section 5.4.7. When a block is to be sent off-processor, the entire block
is sent, even though only a part of it may be required by the receiving processor. In
the current implementation the send.set is not used to speed up the sending out of
a block. Instead, the processors to which a block are sent are computed on the fly.
The factorization code is the same as that given in Figure 6.8 in Section 6.3.2, except
that GET_BLOCK and READY_BLOCK should be treated as simple procedure calls
instead of primitives, since the program-primitive interface is not yet set up. However,
the functionality of the procedures is unchanged. The factorization code may be run
in two modes - the optimized communication mode or the unoptimized communication
mode. In the optimized communication mode, force type message-passing is used to
significantly reduce the communication cost, and the code uses the fetch.set generated
by the scheduler to post asynchronous receives. A preliminary version of the deadlock-
free message-passing strategy described in Section 6.4.4 is implemented, with a single
user-supplied batch size as a parameter, which is used as the value for $M$ and $\Delta$ for all
processors. In the unoptimized communication mode, a receive for a block is posted at
the time that a task requires it for execution, and the fetch.set is not used.

The computational tasks for the numerical factorization and the scheduler when it is
run in the non-symbolic mode, are identical. All tasks use single precision arithmetic.
The code for each of the tasks is given in Section 5.4.6. The tasks $CC$, $CR$ and
$CT$ are sparse column updates, and are implemented using simple index matching as
in the column fan-out $cmod$ tasks. The task $SC$ is essentially a scalar-dense-vector
multiplication. All the other tasks use the Kuck math library routine $s\text{dot}$ for single
precision dot product. However, there is an option to turn off the sdot call and use in-line C code instead. The tasks RRT, RRR, RT and TR use the sdot routine only for vector lengths of at least 30, since, for vector lengths of less than 30, we found that our C code does better than the Kuck library routine, for these tasks.

7.3.3 Column-wrap: implementation

The code follows a static task schedule, just as in the block version. The tasks are cmod and cdv, as described in Section 2.1 and Section 2.2. A parallel column-based scheduler is executed prior to the numerical factorization, to generate a static, completely ordered task list for each processor. The column-wrap code that executes these task lists during numerical factorization is shown in Figure 7.3. The tasks cmod and cdv both use single precision arithmetic.

```
while (task_list is not empty) do
  T ← task at head of task_list; delete T from task_list
  if (T is cdv(k)) then
    col ← pointer to local column k
    cdv(col)
    send_out(k)
  else if (T is cmod(j,k))
    if (k is not available locally) then
      receive k from processor which owns it
    endif
    col₁ ← pointer to column k
    col₂ ← pointer to local column j
    cmod(col₁, col₂)
  endif
endwhile
```

Figure 7.3: Column-wrap numerical factorization code.

The “receive” is a blocking receive, and execution of the program does not resume until the column k is available in the processor. The procedure send_out computes the processors to which a column is to be sent out, on the fly. Different sub-parts of a column are sent to different processors. A destination processor may have several dependents of a column; the largest sub-part of the column required by any of the dependents is sent to that destination, so no processor is sent more than one message.
Sending only the required sub-parts of a column saves on communication volume, but this is traded-off against the additional time spent in packetizing the messages since messages are all of different sizes and consist of different sub-parts of the column.

### 7.3.4 Test matrices

The test matrices were chosen from a variety of applications. A brief description of each test case appears in Table 7.1. The first six cases are from the Harwell-Boeing sparse matrix collection [17]. These matrices are highly unstructured. For brevity, these matrices will also be referred to as B13, B15, B16, B17, B24, and B28, respectively, and collectively, as B* matrices. The 7-point Laplacian discretizations are for structured 3-D grid problems, and the last test case is a structured 2-D grid problem. These cases will be referred to as L16, L20, L25, and L127, respectively, and collectively, as L* matrices. The entry "nonzeros in factor" is the number of nonzeros in the lower triangular factor including the elements on the diagonal.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Description</th>
<th>Order</th>
<th>Nonzeros in factor</th>
<th>Opcount</th>
<th>Block</th>
<th>Column</th>
<th>Block Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCSSTK13</td>
<td>Fluid flow problem</td>
<td>2003</td>
<td>265575</td>
<td>56551652</td>
<td>56459453</td>
<td>47293</td>
<td></td>
</tr>
<tr>
<td>BCSSTK15</td>
<td>Module of an offshore problem</td>
<td>3948</td>
<td>637680</td>
<td>157584821</td>
<td>157421732</td>
<td>59402</td>
<td></td>
</tr>
<tr>
<td>BCSSTK16</td>
<td>Corps of Engineers dam</td>
<td>4884</td>
<td>749027</td>
<td>153737832</td>
<td>153556767</td>
<td>62075</td>
<td></td>
</tr>
<tr>
<td>BCSSTK17</td>
<td>Elevated Pressure Vessel</td>
<td>10974</td>
<td>1020970</td>
<td>148116601</td>
<td>147899872</td>
<td>88329</td>
<td></td>
</tr>
<tr>
<td>BCSSTK24</td>
<td>Calgary Olympics Saddledome Arena</td>
<td>3562</td>
<td>278922</td>
<td>32519351</td>
<td>32454128</td>
<td>14004</td>
<td></td>
</tr>
<tr>
<td>BCSSTK28</td>
<td>Solid element model, linear statics</td>
<td>4410</td>
<td>352261</td>
<td>37667507</td>
<td>37575497</td>
<td>17075</td>
<td></td>
</tr>
<tr>
<td>L7PT16</td>
<td>7-point Laplacian discretization</td>
<td>4096</td>
<td>257288</td>
<td>36260111</td>
<td>36150554</td>
<td>63907</td>
<td></td>
</tr>
<tr>
<td>L7PT20</td>
<td>7-point Laplacian discretization</td>
<td>8000</td>
<td>646839</td>
<td>136190284</td>
<td>135896617</td>
<td>165265</td>
<td></td>
</tr>
<tr>
<td>L7PT25</td>
<td>7-point Laplacian discretization</td>
<td>15625</td>
<td>1719292</td>
<td>556839224</td>
<td>556052830</td>
<td>402495</td>
<td></td>
</tr>
<tr>
<td>L9PT127</td>
<td>9-point Laplacian discretization</td>
<td>16129</td>
<td>563345</td>
<td>42934697</td>
<td>42675065</td>
<td>145164</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1: Test matrices.

The *Opcount* columns list the total number of floating point arithmetic operations
in the numerical factorization. This is obtained by assuming the following: an add and multiply are one flop each, a reciprocal is three flops, a square root and a divide are four flops each [35]. Each entry under “Block” is the number of floating point operations in the block-partitioned method and each entry under “Column” is the number of floating point operations in the column-partitioned method. In the block method, we perform some additional computation due to the TR tasks. To scale a column of the matrix in column-based factorization, the reciprocal of the diagonal is computed once and is used to scale the off-diagonal elements. In our block-based factorization, the scaling of a column is shared by one ST task, and several TR tasks. For each of these tasks, we compute the reciprocal once. So if a column is distributed among k block partitions, we compute the reciprocal k times. The difference in the number of floating point operations between the block and column methods is therefore proportional to the number of rectangles, which in turn depends on the block partitioning parameters W, T and R. For the Opcount entries presented in Table 7.1, we set W to 5 for all the test matrices. For B13, we set T to 800 and R to 1200. For B24 and B28, we set both T and R to 400. For the rest, we set T to 400 and R to 800. In all the cases, except L127, the difference in number of floating point operations between the column and the block methods is between 1% and 2%. For L127, the difference is 6%. Finally, the column BlockTasks gives the total number of tasks in the block factorization for the given partitioning. Note that the number of tasks in the column-wrap factorization is equal to the number of non-zeros in the factor. This is because in every column, every nonzero entry below the diagonal results in a remod task, and the diagonal entry can be thought of as giving rise to a cdv task.

In performing the experiments, two reordering schemes were used. These were: the recursive automatic nested dissection (AND) scheme [28] and Liu’s multiple minimum degree ordering (MMD) scheme [42]. In the following, unless otherwise specified, the B* matrices were reordered using MMD and the L* matrices were reordered using AND. The selection of the reordering schemes was arbitrary. We used SPARSKIT [61] and the Wisconsin Sparse Matrix Manipulation System [2] for generating and converting the test matrices into various formats, and for ordering and symbolically factoring most of
the test matrices. For some of the biggest matrices, we used our own parallel symbolic factorization code running on the iPSC/860.

7.4 Numerical factorization timings

We first present the best results we obtained using the current implementation of SHAPE, and the current block numerical factorization code. The results are divided into two sets, one for unstructured matrices and the other for structured matrices. Table 7.2 gives the factorization timings for some of the unstructured matrices. In this

<table>
<thead>
<tr>
<th>P</th>
<th>Fact</th>
<th>Comp</th>
<th>Comm</th>
<th>Idle</th>
<th>Ovld</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.2: Block factorization timings, in milliseconds, for unstructured matrices.

In this table, we give the total factorization timing (Fact) for up to 32 processors, starting
with the least number of processors on which each test matrix could be factorized. The total execution time on each processor is divided into computation time (Comp), communication time (Comm), idle time (Idle) and overhead time (Ovhd). In the table, for each set of processors, and each time component, we give the mean time ($\lambda$) among all processors, and standard deviation ($S$). The standard deviation in the computation is a measure of the load imbalance among the processors. In Section 7.9.2 we show that to maintain a balanced load with an increasing number of processors, one has to decrease the grain size. In other words, the load imbalance increases with increasing numbers of processors. This is borne out by the increasing standard deviation figures for computation, in Table 7.2.

Figures 7.4(a) and 7.4(b) illustrate the variation of the timings with number of processors, for the B15 and B17 test cases. Time, in seconds, is plotted against 2, 4, 8, 16 and 32 processors for B15, and against 4, 8, 16 and 32 processors for B17. In

![Graph](image)

**Figure 7.4:** Variation of block factorization time with processors: unstructured matrices.

In both these figures, the uppermost curve represents the factorization time. The other curves are the mean times for the components that make up the factorization time. The factorization time curve shows poor speedup in either case. As for the contributions of the time components to the factorization time, communication and overhead both play relatively small roles. The idle time component, however, plays an increasingly
significant role as the number of processors is increased.

Table 7.3 gives the factorization timings for a selected set of the structured matrices. The load imbalance behavior is similar to that seen in the unstructured matrices. Figures 7.5(a) and 7.5(b) illustrate the variation of the timings with number of processors, for the L20 and L127 test cases. Again, we see poor speedup figures, as we did for unstructured matrices. However, we now also notice that the overhead plays a bigger role than it did for the unstructured matrices. This behavior is explained in Section 7.5.2. The idle time again plays an increasingly significant role as the number of processors increases. The idle time is examined in detail in Section 7.7. A final point of note is that the communication time per processor is always less than the computation time per processor for up to 32 processors for both unstructured and structured matrices.

<table>
<thead>
<tr>
<th>P</th>
<th>Fact</th>
<th>Comp</th>
<th>Comm</th>
<th>Idle</th>
<th>Ovhd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>λ</td>
<td>S</td>
<td>λ</td>
<td>S</td>
</tr>
<tr>
<td>L16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>6338</td>
<td>5194.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>4197</td>
<td>2700.50</td>
<td>5.50</td>
<td>382.50</td>
<td>0.50</td>
</tr>
<tr>
<td>4</td>
<td>2831</td>
<td>1511.75</td>
<td>16.02</td>
<td>456.75</td>
<td>13.40</td>
</tr>
<tr>
<td>8</td>
<td>2142</td>
<td>806.38</td>
<td>28.20</td>
<td>444.38</td>
<td>18.71</td>
</tr>
<tr>
<td>16</td>
<td>1770</td>
<td>411.88</td>
<td>53.53</td>
<td>313.85</td>
<td>72.66</td>
</tr>
<tr>
<td>32</td>
<td>1295</td>
<td>199.25</td>
<td>33.36</td>
<td>161.91</td>
<td>62.62</td>
</tr>
<tr>
<td>L20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>7962</td>
<td>4651.50</td>
<td>26.32</td>
<td>1210.25</td>
<td>74.37</td>
</tr>
<tr>
<td>8</td>
<td>6097</td>
<td>2561.62</td>
<td>102.95</td>
<td>1267.88</td>
<td>122.57</td>
</tr>
<tr>
<td>16</td>
<td>4291</td>
<td>1356.00</td>
<td>70.54</td>
<td>1028.12</td>
<td>83.18</td>
</tr>
<tr>
<td>32</td>
<td>3523</td>
<td>666.44</td>
<td>60.82</td>
<td>602.94</td>
<td>117.79</td>
</tr>
<tr>
<td>L127</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8546</td>
<td>4649.50</td>
<td>6.50</td>
<td>1193.50</td>
<td>20.50</td>
</tr>
<tr>
<td>4</td>
<td>6407</td>
<td>2696.25</td>
<td>60.13</td>
<td>1321.25</td>
<td>45.74</td>
</tr>
<tr>
<td>8</td>
<td>4494</td>
<td>1465.12</td>
<td>73.50</td>
<td>1026.62</td>
<td>18.06</td>
</tr>
<tr>
<td>16</td>
<td>3114</td>
<td>728.50</td>
<td>60.90</td>
<td>669.50</td>
<td>76.04</td>
</tr>
<tr>
<td>32</td>
<td>2224</td>
<td>371.75</td>
<td>92.63</td>
<td>366.72</td>
<td>115.62</td>
</tr>
</tbody>
</table>

Table 7.3: Block factorization timings, in milliseconds, for structured matrices.
Figure 7.5: Variation of block factorization time with processors: structured matrices.

7.4.1 Column-wrap: numerical factorization

The results for the column-wrap numerical factorization code are divided into two sets, one for unstructured matrices and the other for structured matrices. Table 7.4 gives the factorization timings for some of the unstructured matrices. We give the total factorization timing (Fact) for upto 32 processors, starting with the least number of processors on which each test matrix could be factorized. The total execution time on each processor is divided into a combined computation and overhead time (Comp + Ovhd), communication time (Comm) and idle time (Idle). In the table, for each set of processors, and each time component, we give the mean time (λ) among all processors, and standard deviation (σ). Table 7.5 gives the factorization timings for some of the structured matrices.

Overall, the block code is three to five times as fast as the column-wrap code for B15, B16 and B17. For the B13 case, it is two to five times as fast. For the structured matrices, the block code is three to four times as fast as the column-wrap code for L127, L16 and L20. The column-wrap code is not communication-optimized, so the comparison is not completely equal. However, we see later that the unoptimized communication times for the block code are still far superior to the column-wrap code,
<table>
<thead>
<tr>
<th>P</th>
<th>Fact</th>
<th>Comp + Ovhd</th>
<th>Comm</th>
<th>Idle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>λ</td>
<td>S</td>
<td>λ</td>
</tr>
<tr>
<td><strong>B13</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>23505</td>
<td>21975.50</td>
<td>52.50</td>
<td>1169.50</td>
</tr>
<tr>
<td>4</td>
<td>13192</td>
<td>11226.50</td>
<td>134.79</td>
<td>1600.00</td>
</tr>
<tr>
<td>8</td>
<td>7987</td>
<td>5856.62</td>
<td>94.77</td>
<td>1783.38</td>
</tr>
<tr>
<td>16</td>
<td>5415</td>
<td>3093.81</td>
<td>55.40</td>
<td>1886.06</td>
</tr>
<tr>
<td>32</td>
<td>4234</td>
<td>1665.44</td>
<td>59.45</td>
<td>1876.66</td>
</tr>
<tr>
<td><strong>B15</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>36523</td>
<td>32128.75</td>
<td>190.13</td>
<td>3546.25</td>
</tr>
<tr>
<td>8</td>
<td>21286</td>
<td>16545.25</td>
<td>181.75</td>
<td>4004.75</td>
</tr>
<tr>
<td>16</td>
<td>13710</td>
<td>8650.88</td>
<td>131.17</td>
<td>4234.88</td>
</tr>
<tr>
<td>32</td>
<td>10148</td>
<td>4618.69</td>
<td>91.14</td>
<td>4282.16</td>
</tr>
<tr>
<td><strong>B16</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>36523</td>
<td>32128.75</td>
<td>391.72</td>
<td>4247.75</td>
</tr>
<tr>
<td>8</td>
<td>21286</td>
<td>15896.62</td>
<td>50.21</td>
<td>4813.25</td>
</tr>
<tr>
<td>16</td>
<td>13710</td>
<td>8361.38</td>
<td>74.11</td>
<td>5226.94</td>
</tr>
<tr>
<td>32</td>
<td>12829</td>
<td>4453.38</td>
<td>51.65</td>
<td>5354.88</td>
</tr>
<tr>
<td><strong>B17</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>25324</td>
<td>15921.12</td>
<td>60.98</td>
<td>7552.25</td>
</tr>
<tr>
<td>16</td>
<td>19068</td>
<td>8500.56</td>
<td>109.44</td>
<td>8003.25</td>
</tr>
<tr>
<td>32</td>
<td>16877</td>
<td>4602.44</td>
<td>62.00</td>
<td>7894.81</td>
</tr>
</tbody>
</table>

Table 7.4: Column-wrap factorization timings, in milliseconds, for unstructured matrices.

and so is the overall performance of the block code.

The load balance in the case of the column-wrap code is better than that of the block code for both structured and unstructured matrices. Considering only the column-wrap code, the load balance is best for the L127 case, and remains fairly steady with increasing number of processors.

Another point worth noting is that the column-wrap code scales better than the block code. The block code is significantly more efficient in both computation and communication. However, as the efficiency improves, parallelization becomes harder. This is because computations need to be increasingly coarser-grained. This difficulty in parallelization is made clear in the way the load imbalance increases with increasing grain size, as we noted earlier. Another manifestation of the difficulty in parallelization
<table>
<thead>
<tr>
<th>( P )</th>
<th>( \text{Fact} )</th>
<th>( \text{Comp + Ovhd} )</th>
<th>( \text{Comm} )</th>
<th>( \text{Idle} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \lambda )</td>
<td>( S )</td>
<td>( \lambda )</td>
<td>( S )</td>
</tr>
<tr>
<td>L127</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>17823</td>
<td>9834.00</td>
<td>133.22</td>
<td>5349.00</td>
</tr>
<tr>
<td>8</td>
<td>13402</td>
<td>5347.25</td>
<td>64.72</td>
<td>5336.38</td>
</tr>
<tr>
<td>16</td>
<td>9900</td>
<td>2906.94</td>
<td>52.24</td>
<td>4970.62</td>
</tr>
<tr>
<td>32</td>
<td>7370</td>
<td>1570.31</td>
<td>33.11</td>
<td>4267.12</td>
</tr>
<tr>
<td>L16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>15907</td>
<td>14287.00</td>
<td>59.00</td>
<td>1334.00</td>
</tr>
<tr>
<td>4</td>
<td>9593</td>
<td>7386.00</td>
<td>58.10</td>
<td>1812.50</td>
</tr>
<tr>
<td>8</td>
<td>6263</td>
<td>3937.88</td>
<td>41.05</td>
<td>1953.88</td>
</tr>
<tr>
<td>16</td>
<td>4509</td>
<td>2102.38</td>
<td>35.21</td>
<td>1970.69</td>
</tr>
<tr>
<td>32</td>
<td>3788</td>
<td>1119.88</td>
<td>31.08</td>
<td>1931.81</td>
</tr>
<tr>
<td>L20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>32255</td>
<td>27250.00</td>
<td>243.19</td>
<td>3884.75</td>
</tr>
<tr>
<td>8</td>
<td>19340</td>
<td>14159.12</td>
<td>146.78</td>
<td>4252.75</td>
</tr>
<tr>
<td>16</td>
<td>12638</td>
<td>7437.12</td>
<td>120.60</td>
<td>4343.00</td>
</tr>
<tr>
<td>32</td>
<td>9637</td>
<td>3924.56</td>
<td>122.50</td>
<td>4369.03</td>
</tr>
</tbody>
</table>

Table 7.5: Column-wrap factorization timings, in milliseconds, for structured matrices.

is the increase in the idle time. Thus, even though the overall performance of the block code is much better than the column-wrap code, the speedups for the block code are worse.

We will analyze the block and column-wrap results by considering the individual time components, their contribution to the overall performance and the factors that influence them. We begin by looking at the computation and overhead times in the context of single-processor performance.

### 7.5 Single-processor performance

On a single processor, the execution time is the sum of the computation time and the overhead time. We give the single-processor performance for all the test matrices in Table 7.6. The second, third and fourth columns are the times associated with the numerical factorization, measured in milliseconds. \( \text{Comp} \) is the computation time on a
<table>
<thead>
<tr>
<th>Matrix</th>
<th>Factorization (in secs.)</th>
<th>Mflops</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ptime</td>
<td>Comp</td>
</tr>
<tr>
<td>B13</td>
<td>8157</td>
<td>7221</td>
</tr>
<tr>
<td>B15</td>
<td>20806</td>
<td>19176</td>
</tr>
<tr>
<td>B16</td>
<td>19288</td>
<td>17612</td>
</tr>
<tr>
<td>B17</td>
<td>27591</td>
<td>24327</td>
</tr>
<tr>
<td>B24</td>
<td>4397</td>
<td>4120</td>
</tr>
<tr>
<td>B28</td>
<td>5282</td>
<td>4951</td>
</tr>
<tr>
<td>L16</td>
<td>6338</td>
<td>5194</td>
</tr>
<tr>
<td>L20</td>
<td>24152</td>
<td>18604</td>
</tr>
<tr>
<td>L25</td>
<td>75088</td>
<td>59456</td>
</tr>
<tr>
<td>L127</td>
<td>13208</td>
<td>9299</td>
</tr>
</tbody>
</table>

Table 7.6: Single-processor performance.

This table shows the performance of matrices on a single processor. The time spent in all the task computational kernels for factorization, known as Ovhd, is the overhead time on a single processor, taken by all activities other than computation. These activities include scheduling the tasks, setting appropriate pointers to the participating data blocks, and, in the case of more than one processor, freeing space that may have been allocated for a non-local block. The entry Ptime is the sum of Comp and Ovhd. We ran B13, B24, B28 and L16 on one processor, and the times given here for Comp and Ovhd are true single-processor timings. The other matrices could not be run on one processor due to space limitations. Instead, we ran each of them on the least possible number of processors, as shown in Tables 7.2 and 7.3. We then multiplied the mean Comp and Ovhd times for the least number of processors by the number of processors, to obtain the figures shown in the table. These figures are overestimates because the Comp and Ovhd times do not scale perfectly with increasing number of processors. In general, when the number of processors is doubled, Comp and Ovhd are halved, at best. This tendency is noted in Tables 7.2 and 7.3. The “Mflops” column is the delivered floating point rate, in millions of floating point operations per
second or megaflops. Each entry in this column is obtained by dividing the number of floating point operations under column “Block” in Table 7.1 by the factorization time under column “Ptime”, and scaling the ratio down to units of megaflops.

### 7.5.1 Task computational speed

We first analyze the computation time, and determine the effect of different factors on the computation time. To start with, we examine the computational speed achievable with each type of task. Table 7.7 gives the computational speeds for the four most frequently occurring types of tasks for each test matrix. These are RR, TR, RT and RRT.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>RRR %</th>
<th>μs/flop</th>
<th>TR %</th>
<th>μs/flop</th>
<th>RT %</th>
<th>μs/flop</th>
<th>RRT %</th>
<th>μs/flop</th>
<th>Comp mflops</th>
</tr>
</thead>
<tbody>
<tr>
<td>L25</td>
<td>87.28</td>
<td>0.09</td>
<td>64.0</td>
<td>0.21</td>
<td>2.91</td>
<td>0.12</td>
<td>2.70</td>
<td>0.11</td>
<td>9.36</td>
</tr>
<tr>
<td>B16</td>
<td>78.25</td>
<td>0.09</td>
<td>9.29</td>
<td>0.16</td>
<td>5.72</td>
<td>0.10</td>
<td>3.24</td>
<td>0.06</td>
<td>8.78</td>
</tr>
<tr>
<td>B15</td>
<td>81.92</td>
<td>0.10</td>
<td>8.52</td>
<td>0.17</td>
<td>5.21</td>
<td>0.18</td>
<td>2.12</td>
<td>0.07</td>
<td>8.51</td>
</tr>
<tr>
<td>B24</td>
<td>68.87</td>
<td>0.11</td>
<td>14.65</td>
<td>0.17</td>
<td>10.53</td>
<td>0.13</td>
<td>4.33</td>
<td>0.09</td>
<td>7.89</td>
</tr>
<tr>
<td>B13</td>
<td>74.05</td>
<td>0.10</td>
<td>12.01</td>
<td>0.15</td>
<td>7.18</td>
<td>0.12</td>
<td>2.80</td>
<td>0.10</td>
<td>7.76</td>
</tr>
<tr>
<td>B28</td>
<td>65.34</td>
<td>0.11</td>
<td>16.08</td>
<td>0.17</td>
<td>11.71</td>
<td>0.13</td>
<td>4.89</td>
<td>0.09</td>
<td>7.60</td>
</tr>
<tr>
<td>L20</td>
<td>81.84</td>
<td>0.11</td>
<td>9.08</td>
<td>0.21</td>
<td>4.94</td>
<td>0.13</td>
<td>2.72</td>
<td>0.14</td>
<td>7.29</td>
</tr>
<tr>
<td>L16</td>
<td>73.42</td>
<td>0.11</td>
<td>12.82</td>
<td>0.18</td>
<td>7.22</td>
<td>0.12</td>
<td>3.76</td>
<td>0.13</td>
<td>6.98</td>
</tr>
<tr>
<td>B17</td>
<td>71.87</td>
<td>0.12</td>
<td>12.24</td>
<td>0.18</td>
<td>8.88</td>
<td>0.18</td>
<td>3.03</td>
<td>0.11</td>
<td>6.16</td>
</tr>
<tr>
<td>L127</td>
<td>60.26</td>
<td>0.15</td>
<td>17.74</td>
<td>0.26</td>
<td>10.22</td>
<td>0.16</td>
<td>5.30</td>
<td>0.20</td>
<td>4.10</td>
</tr>
</tbody>
</table>

Table 7.7: Computational rates for tasks.

An entry in the column “%” for a certain task type is the percentage of the total floating point operations contributed by tasks of that type. An entry in the column “μs/flop” for a certain task type is the average time per floating operation executed in the context of tasks of that type, obtained by dividing the total time taken during the computation of all tasks of that type by the total number of floating operations performed by these tasks. The last column, “Comp mflops”, lists the megaflops rates delivered by the computations only. The entry in this column for any test matrix is the ratio of its “Opcount” entry for the block partition in Table 7.1 to its “Comp” entry in Table 7.6, appropriately scaled to units of millions of floating point operations per second. The matrices in the table are sorted in descending order of the computational mflops.
Note that, for each of the matrices, the tasks RRR, RT, TR and RRT together contribute more than 90% of the total floating point operations in the factorization. RRR contributes the largest number of floating point operations, followed by TR, RT and RRT, in that order. Also, RRR is the fastest of the task types RRR, RT and TR. Thus, the greater the greater the proportion of RRR tasks, the better is the computational performance, provided the rectangles are of a certain minimum grain size. We compare the performance of RRR tasks in B15 with that of RRR tasks in L127, to illustrate the effect of grain size, locality and vector lengths on performance. For the RRR tasks in B15, the computational speed is 0.1 \( \mu s/\)flop and for the RRR tasks in L127, it is 0.15 \( \mu s/\)flop.

The grain size effect is analyzed as follows. Since we are dealing with RRR tasks, we take into account rectangles only. The grain size of a rectangle is the number of non-zeros it contains. For all the test matrices we computed the grain sizes of the naturally occurring rectangles that are at least five columns wide, i.e. \( W \) was set to 5. In Figure 7.6, we divided the grain sizes for rectangles in B15 and L127 into six ranges as shown in the z axis of that bar chart. For each range, we computed the ratio of the number of rectangles whose size fall in this range, to the total number of rectangles. This is shown on the y axis, given by the heights of the bars. About 97% of the rectangles in L127 contain no more than 200 non-zeros each. If the grain size requirement is greater 200, these naturally occurring rectangles fall below the grain size requirement. B15 has a larger number of coarser-grained rectangles, leading to improved cache utilization, in general.

We examine this performance difference between B15 and L127 more closely by considering locality in terms of groups of vector operations, and utilization of floating point units. Given a certain number of floating point operations to be done using rectangular blocks, the greater the number of rectangles over which this is distributed, the worse is the performance. A large number of small blocks will result in worse performance than a small number of large blocks, provided the blocks can fit in the cache. Given a certain block size, longer vectors will result in better performance. For the test matrices, we computed the number of naturally occurring rectangles that are at
least five columns wide. Since RRR tasks determine performance, we again concentrate on the rectangles. Each row of a rectangle is a dense vector in the RRR computation, as described in Section 5.4.6. Figure 7.7(a) compares the lengths of these vectors for B15 and L127. The x axis divides the vector lengths into ranges. The y axis gives the percentage of the total number of vectors that falls in a given range of widths. The B15 matrix contains a larger percentage of long vectors than the L127 matrix. Thus, we can expect much better exploitation of floating point units in the B15 case. In Figure 7.7(b), the y-axis is the ratio of number of rectangles to number of vectors that are contained in these rectangles. We will refer to this as the *locality ratio*. The higher this ratio, the larger is the spread of vectors among the rectangles, and the greater is the number of tasks, leading to reduced cache utilization. From Figure 7.7(b), it is evident that in all but vectors longer than 100 elements, the locality ratio in the B15 matrix is less than half that in the L127 matrix, indicating much better utilization of cache due to improved locality. Further discussion on grain size and floating point operations rate
appears in the section on selection of parameters, Section 7.9.

7.5.2 Overheads

As stated earlier in this section, scheduling the tasks by setting appropriate pointers to the participating data blocks, and, in the case of more than one processor, freeing space that may have been allocated for a non-local block, are the major factors contributing to the overhead. We also include calls to subroutines in the overhead time. The explicit task scheduling approach that we have adopted, with irregular block interactions, involves book-keeping activities that are greater than column-based methods.

The ratio of the number of floating point operations performed during factorization, to the total number of tasks, is a measure of the relative contribution of the overhead to the total execution time on a processor. In Figure 7.8(a), we give the percentage contribution of the overhead to the total execution time for five test cases. In Figure 7.8(b), we show the ratio of floating point operations to number of tasks for these test cases. These figures are obtained by dividing the Opcount entry by the BlockTasks entry in Table 7.1. As can be seen, the greater the number of floating point operations performed per task, the smaller is the relative contribution of overhead to the total execution time. For the L127 case, overheads are responsible for significant performance
degradation. How much of the overheads can be reduced by tuning the code, and is there a point beyond which the overheads cannot be reduced? As we saw in Figure 7.6 for the L127 matrix, there is a large number of small rectangles which result in a large number of tasks. Given a grain size, $R$, the rectangles which contain fewer non-zeros than $R$ cannot be further partitioned. This is a property of the matrix which is inherent in the structure, and which immediately places a constraint on the performance.

### 7.5.3 Improvement of performance

From the preceding discussion, it is clear that computational performance is highly dependent on the size and shape of rectangles. We need rectangles that (a) satisfy the grain size requirement, and (b) give long vectors. What if there are a large number of rectangles that do not satisfy the grain size requirement in a given matrix? This would give rise to a very large number of tasks, high overhead, and also poor cache utilization due to the computations not being sufficiently localized. However, the grain size can be increased by “coalescing” rectangles. By coalescing, we mean putting together rectangles that are vertically contiguous, of the same width and in the same cluster, to create sparse rectangles. These sparse rectangles would be accompanied by index vectors to indicate the row numbers of the constituent vectors, just as in the case
of sparse columns. In fact, a sparse column is a degenerate sparse rectangle in which each constituent row vector is of unit size. We believe that in the RRR tasks that use sparse rectangles, the computational overhead introduced by the index vector matching would be significantly less than the additional locality obtained due to the grouping of hitherto scattered vectors.

To get an idea of the amount of difference such a coalescing would make, we performed the following computations on the sparsity structures of the test matrices. Given a grain size $R$, we coalesced rectangles until either the grain size requirement was met or no more coalescing was possible. We computed the number of rectangles before and after coalescing, and the increase in grain size due to the coalescing. Figure 7.9(a) shows the percentage decrease in the number of rectangles after coalescing and Figure 7.9(b) shows the average rectangle grain size before and after coalescing. The grain size, $R$, was set to 800 for coalescing. The reduction in the number of rectangles due to coalescing is substantial, between 50% and 80%. This would, in turn, reduce the number of dependencies and therefore, the $RRR$ tasks, by a large amount, leading to a significant reduction in the scheduling overheads. The reduction in number of rectangles is consistently greater for the unstructured matrices than the structured ones. As for the grain size, we see in Figure 7.9(b) that the unstructured matrices show a
dramatic improvement, as much as a fivefold increase in grain size. The structured matrices show a relatively less significant improvement. Thus, overall, decreased cost of overheads and improved cache utilization would be observed to a far greater extent in the unstructured matrices.

7.5.4 Column-wrap: single processor performance

We obtain the single-processor timings for the column-wrap code in the same manner as that for the block code, described in the beginning of this section. Table 7.8 gives the total computation and overhead time, in seconds, and the mflops delivered for some of the test matrices. To compute the mflops for any test matrix, we divided the column operation count given in Table 7.1 by the time entry in this table, scaled down to units of millions of floating point operations per second.

<table>
<thead>
<tr>
<th></th>
<th>B13</th>
<th>B16</th>
<th>B17</th>
<th>B28</th>
<th>L16</th>
<th>L20</th>
<th>L25</th>
<th>L127</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>44</td>
<td>122</td>
<td>127</td>
<td>30</td>
<td>29</td>
<td>109</td>
<td>469</td>
<td>30</td>
</tr>
<tr>
<td>Mflops</td>
<td>1.3</td>
<td>1.3</td>
<td>1.2</td>
<td>1.3</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.1</td>
</tr>
</tbody>
</table>

Table 7.8: Single-processor performance: column-wrap.

The column-wrap code consistently delivers under 1.3 mflops. The combined computation and overhead in the column-wrap code is between five and seven times as expensive as that in the block code, in all but the L20 and L127 cases.

7.6 Communication performance

The analysis in Chapter 4 showed that both the number of messages and communication volume for 2D block partitioning grow asymptotically slower than column partitioning, for 2D regular grid problems. In this section, we compare the number of messages and volume of data communicated by the block method on 2D and 3D structured grid matrices and unstructured matrices, with those communicated by the column-wrap method. We then present experimental results to show the significant gains achieved by optimizing the communication.
7.6.1 Messages and volume

For the experimental results on communication messages and volume, we consider the additional structured matrices shown in Table 7.9. These matrices were reordered using the Automatic Nested Dissection (AND) scheme.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Description</th>
<th>Order</th>
<th>Nonzeros in factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>L7PT18</td>
<td>7-point Laplacian discretization</td>
<td>5832</td>
<td>413622</td>
</tr>
<tr>
<td>L7PT23</td>
<td>7-point Laplacian discretization</td>
<td>12167</td>
<td>1197626</td>
</tr>
<tr>
<td>L9PT63</td>
<td>9-point Laplacian discretization</td>
<td>3969</td>
<td>108235</td>
</tr>
<tr>
<td>L9PT100</td>
<td>9-point Laplacian discretization</td>
<td>10000</td>
<td>326365</td>
</tr>
<tr>
<td>L9PT175</td>
<td>9-point Laplacian discretization</td>
<td>30625</td>
<td>1181764</td>
</tr>
</tbody>
</table>

Table 7.9: Structured grid matrices for communication performance evaluation.

The communication figures for the structured grid matrices are given in Table 7.10. Under “Block partitioning”, there are four columns of entries. Of these, “Block Messages” are the total number of inter-processor messages that involve non-column blocks, and “Column Messages” are the total number of inter-processor messages that involve only sparse columns. The “Volume” entry in both block and column partitioning is the total inter-processor communication volume in kilobytes. The blank entries for the column-wrap method indicate that we could not run the corresponding tests due to memory space limitations on the processors. For both 2D and 3D structured grid matrices, the number of messages and communication volume for the column-wrap case is seen to grow much faster than those for the block partitioning case. We illustrate this rate of growth of communication for the representative matrices L9PT127 and L7PT20, by plotting the number of messages against number of processors, as shown in Figures 7.10(a) and 7.10(b). In either figure, the messages are plotted for 4, 8, 16 and 32 processors. There are two curves for the block method. The “Block partition” curve is for the total number of messages, while the “Blocks only” curve is for the number of messages that are used to communicate only the triangles and rectangles. In the L20 case, for a smaller number of processors, the number of messages in the block partitioning is actually greater than the column-wrap method. However, as the number of processors increases, the growth rate in the former is the slower than the latter. Note that most
<table>
<thead>
<tr>
<th>Matrix</th>
<th>P</th>
<th>Block partitioning</th>
<th>Column-wrap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Messages</td>
<td>Volume (kbytes)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Block</td>
<td>Column</td>
</tr>
<tr>
<td>L9PT63</td>
<td>2</td>
<td>1193</td>
<td>3250</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2361</td>
<td>7274</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>3155</td>
<td>10526</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>3652</td>
<td>12693</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>4099</td>
<td>14078</td>
</tr>
<tr>
<td>L9PT100</td>
<td>2</td>
<td>3707</td>
<td>8542</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7886</td>
<td>19153</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>11159</td>
<td>27691</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>13178</td>
<td>33429</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>14066</td>
<td>36331</td>
</tr>
<tr>
<td>L9PT127</td>
<td>2</td>
<td>5566</td>
<td>13814</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>12126</td>
<td>31694</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>17807</td>
<td>47152</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>21164</td>
<td>57487</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>22823</td>
<td>63146</td>
</tr>
<tr>
<td>L9PT175</td>
<td>8</td>
<td>38354</td>
<td>49201</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>47466</td>
<td>59994</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>53149</td>
<td>66513</td>
</tr>
<tr>
<td>L7PT16</td>
<td>2</td>
<td>2435</td>
<td>4338</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6192</td>
<td>10678</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>10212</td>
<td>17271</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>13002</td>
<td>21712</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>14524</td>
<td>24537</td>
</tr>
<tr>
<td>L7PT18</td>
<td>2</td>
<td>4046</td>
<td>6735</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10413</td>
<td>16772</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>17661</td>
<td>27565</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>23921</td>
<td>36344</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>26946</td>
<td>41086</td>
</tr>
<tr>
<td>L7PT20</td>
<td>4</td>
<td>15664</td>
<td>24379</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>27635</td>
<td>41487</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>38695</td>
<td>56656</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>45756</td>
<td>65962</td>
</tr>
<tr>
<td>L7PT23</td>
<td>8</td>
<td>42143</td>
<td>20762</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>58312</td>
<td>26818</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>65659</td>
<td>30407</td>
</tr>
</tbody>
</table>

Table 7.10: Communication for structured grid matrices.
of the communication in the block partitioning method is due to the columns. The number of columns in the partition is dictated by the sparsity structure of the matrix, which inherently bounds the communication performance improvement.

The growth in volume of communication is similar, as shown for the same test cases in Figures 7.11(a) and 7.11(b). The trend in communication growth rates shown in Figures 7.10 and 7.11 is representative of all the structured grid problems we looked at.

In Table 7.11 we compare the number of messages and volume of communication of the block method against the column-wrap method for unstructured matrices. Again, blank entries in the table indicate that we could not run the corresponding tests due to lack of processor memory space.

The rate of growth of communication for B15 is illustrated in Figures 7.12(a) and 7.12(b). In the first, the total number of messages is plotted against number of processors, and in the second, the total communication volume is plotted against the number of processors. As in Figures 7.10 and 7.11, a separate curve is plotted for the communication involving only triangular and rectangular blocks. The comparative trend in communication growth for block and column-wrap, both in messages and in volume, is the same as that for structured matrices. The absolute communication is greater than
that for the structured matrices since the unstructured matrices have a larger number of nonzeros.

The growth rate of messages and volume for the block partitioning method observed in practice may differ from the analysis of Section 5.1 due to the following reasons. The theoretical analysis is based on a model which does not take into account the granularity of the architecture. In Chapter 5 we saw how strongly the granularity and the shape of the block partitions affect performance. Apart from this, the allocation strategy that we have implemented considers both load balancing and communication cost reduction, whereas for the analysis, load balancing is not a constraint.

To summarize, compared to the column-wrap method, the block partitioning demonstrates much better scalability with respect to growth of communication with increasing numbers of processors. The absolute number of messages and volume of communication can be improved in the column partitioning case by replacing wrap-mapping with better mapping and scheduling strategies, as for instance the subtree-to-subcube assignment method or the fan-in scheduling scheme. However, the asymptotic growth rate of communication with number of processors would be still better in the block partitioning method, as shown by the analysis. The BLOCC analysis is a first step on which our
<table>
<thead>
<tr>
<th>Matrix</th>
<th>P</th>
<th>Block partitioning Messages</th>
<th>Block</th>
<th>Column</th>
<th>Total</th>
<th>Volume (kbytes)</th>
<th>Column-wrap Messages</th>
<th>Volume (kbytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B13</td>
<td>2</td>
<td>1274</td>
<td>721</td>
<td>1995</td>
<td>1147</td>
<td>2002</td>
<td>2121</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3502</td>
<td>2053</td>
<td>5555</td>
<td>3289</td>
<td>5972</td>
<td>6305</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>6572</td>
<td>3829</td>
<td>10401</td>
<td>6262</td>
<td>13717</td>
<td>14429</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>9201</td>
<td>6240</td>
<td>15441</td>
<td>9326</td>
<td>28184</td>
<td>29698</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>10817</td>
<td>8114</td>
<td>18931</td>
<td>11501</td>
<td>53103</td>
<td>57076</td>
<td></td>
</tr>
<tr>
<td>B15</td>
<td>2</td>
<td>1759</td>
<td>1328</td>
<td>3087</td>
<td>2749</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4863</td>
<td>3706</td>
<td>8569</td>
<td>7863</td>
<td>11811</td>
<td>15165</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9372</td>
<td>7044</td>
<td>16416</td>
<td>16196</td>
<td>27365</td>
<td>34845</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>14143</td>
<td>10233</td>
<td>24376</td>
<td>25998</td>
<td>56861</td>
<td>72369</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>18618</td>
<td>12757</td>
<td>31375</td>
<td>35126</td>
<td>109104</td>
<td>140717</td>
<td></td>
</tr>
<tr>
<td>B16</td>
<td>2</td>
<td>1932</td>
<td>1244</td>
<td>3176</td>
<td>3256</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5406</td>
<td>3644</td>
<td>9050</td>
<td>9408</td>
<td>14424</td>
<td>17833</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>10310</td>
<td>7450</td>
<td>17760</td>
<td>19057</td>
<td>33634</td>
<td>40944</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>15002</td>
<td>11398</td>
<td>26400</td>
<td>28898</td>
<td>71546</td>
<td>84939</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>18099</td>
<td>15180</td>
<td>33279</td>
<td>36030</td>
<td>142408</td>
<td>163908</td>
<td></td>
</tr>
<tr>
<td>B17</td>
<td>4</td>
<td>8862</td>
<td>7898</td>
<td>16760</td>
<td>11624</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>14634</td>
<td>14808</td>
<td>29442</td>
<td>21234</td>
<td>72035</td>
<td>54865</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>19260</td>
<td>20791</td>
<td>40051</td>
<td>28910</td>
<td>148898</td>
<td>111109</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>22184</td>
<td>25510</td>
<td>47694</td>
<td>34767</td>
<td>281565</td>
<td>205629</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.11: Communication for unstructured matrices.

The block partitioning method is based, and the consistent reduction in communication compared to the column-wrap method justifies our choice of partitioning.
7.6.2 Communication optimization

Even though the communication performance of the block method is far superior to that of column methods, the cost of communication is relatively high compared with the computation cost. In fact, the high latency cost can easily overwhelm the computation cost on a processor, as shown, for instance, in Figures 7.13(a) and 7.13(b). These figures show the mean per-processor time taken by communication and computation for the block-based factorization of unstructured and structured matrices on 32 processors. For both structured as well as unstructured matrices, communication contributes twice as much or more to the total execution time as computation. Reduction of communication time is definitely needed in order to arrive at a balance between computation and communication.

Currently, we have implemented a preliminary version of the communication optimization strategy described in Section 6.4.4. In this version, we accept a constant batch size $UM$ as a parameter, and post batches of asynchronous receives so as to ensure that at any point of time at least $UM$ receives are always pending. Using this strategy, we get the communication time figures listed in Tables 7.2 and 7.3. The reduction in communication due to the optimization is shown in Figures 7.14(a) and 7.14(b) for several
matrices on 32 processors, and in Figures 7.14(c) and 7.14(d) for B15 and L127.

In Figure 7.15 we show the relative contributions of optimized communication and computation to the total execution time on a processor, for factorization on 32 processors. After communication optimization, the communication time is never more than the computation time, for all the test matrices shown. We therefore achieve the balance we seek between computation and communication time on a processor.

What is the impact of the reduction in communication time on the factorization time? We show this reduction in factorization time in Figures 7.16(a) and 7.16(b) for several matrices on 32 processors, and in Figures 7.16(c) and 7.16(d) for B15 and L127 on a range of processors. Evidently, the reduction in the factorization time due to optimized communication is much less than the reduction in communication time itself. Of course, the reduction in factorization time depends on the extent of the contribution of the communication time to the total execution time. We see that the reduction in factorization time is less than proportionate to the reduction in communication time, indicating that some other time component has not stayed constant. Of the computation, overhead and idle times, the only component that can change markedly is the idle time, since the change in communication protocol does not affect the computation or
task scheduling overhead on a processor. We discuss the role of the idle time, and the ways and means to reduce processor idling, in the following section.

7.7 Idle time

As the computation and communication costs on a processor are reduced, processor idling time plays an increasingly dominant role in the factorization procedure. We show the idle time before and after communication optimization, for all test cases, for 32 processors, in Figures 7.18(a) and 7.18(b) and then for a varying number of processors, for B15 and L127, in Figures 7.18(c) and 7.18(d). The idle times shown in Figure 7.18 increase with optimization of communication, in most cases. For 32 processors, there is a reduction in idle time after communication optimization for some cases. Even in these cases, the decrease in idle time is nominal compared to the decrease in communication time. Overall, the idle time is a limiting factor in achieving better performance.

Going back to Table 7.2 and Table 7.3, we see that for the unstructured matrices, the mean idle time per processor is, in general, greater than the mean communication time per processor, and the difference increases rapidly with increasing numbers of processors. For the structured matrices, the idle time outstrips the communication time for eight or more processors. As for the idle time relative to the computation time, for the unstructured test matrices, the idle time is less than the computation time for up to eight processors. On increasing the number of processors to sixteen and more, the idle time shoots up and overwhelms the computation time. A similar behavior is noticed in the case of structured matrices, in which using more than eight processors results in the idle time dominating the computation time. Obviously, for any given problem size, there is a point beyond which the idle time will always outstrip the computation time as the number of processors is increased. The question is, what are the factors that determine this point, or number of processors at which the idle time overwhelms the computation time. The two major factors are (a) the amount of exploitable parallelism available in the problem, and (b) the extent to which the partitioner and scheduler exploit this parallelism.
The amount of exploitable parallelism available in the problem is a quantity that is hard to measure for unstructured problems such as sparse Cholesky factorization. Besides, for current coarse-grained large-scale multiprocessors, the amount of parallelism that can be exploited is less than the parallelism inherent at the operation level, due to the agglomeration of computations and data in order to increase the granularity of the algorithm. The partitioning algorithm, therefore, plays an important part in determining the extent of the parallelism retained in the problem instance after partitioning. Choosing an appropriate grain size can make all the difference. A bad choice of grain size results in a partitioning that is either too fine-grain or too coarse-grain, and the scheduler can then only do so much to save the situation.

We illustrate these points by giving an example of the B15 test case, which was run using a very large grain size, \( G = 1600 \). The factorization timings are shown in Table 7.12. Compare these timings with those for B15 in Table 7.2. Overall, the factorization time is much worse, especially with a larger number of processors. On 32 processors, for instance, the large grain version is more than twice as slow as the version with \( T = 400 \) and \( R = 800 \). Despite of the grain size being twice as large, the computation time is only marginally better. In Section 7.9.2 we show that that an inordinate increase in grain size does not result in a proportionate decrease in the computation cost, and for a given matrix, there is a saturation low point for the computation time. Moreover, we see a far greater load imbalance in the large grain case, indicated by the high standard deviation figures for the computation time. Evidently, the tasks are so coarse-grained that load balancing becomes an almost impossible task. This is also seen in the idle time - on 32 processors, the mean idle time on a processor is ten times as

<table>
<thead>
<tr>
<th>P</th>
<th>Fact</th>
<th>Comp</th>
<th>Comm</th>
<th>Idle</th>
<th>Ovhd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>( \lambda )</td>
<td>( S )</td>
<td>( \lambda )</td>
<td>( S )</td>
</tr>
<tr>
<td>2</td>
<td>12286</td>
<td>8868.00</td>
<td>203.00</td>
<td>384.50</td>
<td>21.50</td>
</tr>
<tr>
<td>4</td>
<td>9607</td>
<td>4638.25</td>
<td>222.41</td>
<td>413.50</td>
<td>62.50</td>
</tr>
<tr>
<td>8</td>
<td>8425</td>
<td>2329.38</td>
<td>395.20</td>
<td>310.50</td>
<td>111.01</td>
</tr>
<tr>
<td>16</td>
<td>7446</td>
<td>1163.38</td>
<td>575.92</td>
<td>262.50</td>
<td>129.42</td>
</tr>
<tr>
<td>32</td>
<td>7854</td>
<td>585.09</td>
<td>540.60</td>
<td>229.12</td>
<td>249.79</td>
</tr>
</tbody>
</table>

Table 7.12: Factorization timings in milliseconds for B15, large grain size.
much as the mean computation time in the large grain case. The mean communication and overhead times are much less in the large grain version, but the variation is too high for these to contribute in any significant measure toward lowering the overall parallel factorization time. Our partitioner is flexible enough that the grain size parameters can be chosen carefully by using SHAPE to conduct experiments in quick time. Assuming that the grain size has been chosen appropriately so as to retain as much parallelism as possible, while matching the architectural granularity, the allocation and ordering of tasks on the processors determine parallel performance. More investigation needs to be carried out to determine, quantitatively, how these affect the idle time.

We can think of the idle time contributed by the scheduler as a function of the degree of mismatch between the ordering generated by the scheduler and an optimal ordering, for a given partitioning and mapping. This mismatch can be caused by three factors. First, symbolic scheduling, as described in Section 6.3 completely ignores the task computation costs. The ordering generated would create a mismatch with the actual factorization ordering, unless, possibly, all tasks took the same time. In problems such as sparse Cholesky factorization, there is a wide range of task computational requirements due to the unstructured nature of the dependencies, let alone the variation of computational rates due to the shape and size of partitions participating in the tasks. So, in order to reduce the ordering mismatch, we run the scheduler in the non-symbolic mode. In this mode, the numerical computations are performed, thus effectively performing the first iteration of factorization. Second, the data structure manipulation overheads which are fairly high in the scheduling phase are completely absent in the numerical factorization phase. This extra time is currently not accounted for, and gets merged in with the execution of the scheduler. Hence, the arrival of data at a processor in the scheduling phase occurs, in general, at a later time than it would have arrived if the data structures were absent. This would create a mismatch between the generated ordering and an optimal ordering. Third, in the scheduling phase, the inter-block dependencies need to be communicated among processors, as shown in the scheduler code for block sparse Cholesky factorization in Figure 6.7. This increases the communication set-up and transmission times, which affect the data arrival times
in the same way as the data structure manipulation overheads. During the numerical factorization step, these dependencies are not required to be stored or communicated, thereby creating a mismatch between the generated ordering and an optimal one.

It is extremely difficult to isolate the contributions due to these three factors, especially since a mismatch at some point of time tends to get multiplied several times over along the task dependency chains. Currently, we are investigating the idle time effect in different regions of the overall execution trace. The idea is to arrive at a clearer understanding of critical regions where idle time is high, and relate those to the types of tasks that are executed in those regions and the dependencies among them. This would be useful in identifying the critical paths, and assigning higher priorities to tasks in the critical paths. Apart from this there are two possible improvements.

The first possible way to improve the schedule is to generate a total ordering of tasks for every processor by simulating the factorization, given the BCR, the communication cost parameters of the architecture and the average computational rates for the tasks. This would result in a schedule far better than the one generated by the current strategy, provided the communication and computation costs could be accurately modeled. Unfortunately, this provision is not easily met. As we have seen in the task computational rates presented earlier, the geometry and size of the partitioning affects the behavior of the cache and floating point units significantly, and will certainly vary with the architecture of the cpu. Therefore, the task computational rates are hard to determine a priori, given only the number of floating point operations performed. One may have to use a training set of task computational kernels on each target machine to estimate the performance and then use these estimates to run the simulator. The training set method was introduced by Balasundaram et. al. [7] to statically estimate the computation and communication performance and thereby evaluate the relative efficiency of different data partitioning schemes for distributed programs. We can also use the training set method to estimate the communication costs on the target architecture for the various communication primitives used by the program. Assuming that these costs can be estimated, the question is, what strategy do we adopt for the simulation itself and how fast can we execute the parallel simulator? Since the simulation has to be
carried out as part of the run-time preprocessing, we would want the simulator to be not much slower than the numerical factorization itself. This needs further investigation.

The second possible way to improve the schedule is to run the scheduler in two passes. In the first pass, we run the scheduler as before, either in the pure symbolic mode or in the mode where numerical computations are performed, and generate a first ordering. We then use this static ordering as a guiding line to perform the numerical factorization, by following the schedule as much as possible, but not being restricted to it. This second pass therefore avoids both the data structure manipulation, and the storage and communication of dependencies that are the sources of inefficiency and inaccurate timings in the first pass. The ordering obtained by the second pass is then used as the static task ordering for further iterations. We have implemented this two-pass scheduling scheme on the iPSC/860. Preliminary results show that we get about 20% improvement in factorization time for the test cases we tried. A more thorough investigation needs to be conducted to ensure that we see this improvement consistently, and that second pass is more efficient than the first pass.

7.8 Comparison with a distributed multifrontal code

Recently, Eswar et. al. [20] have shown very good timings for the sparse Cholesky factorization of a 127 × 127 grid matrix on a 32-processor iPSC/860, using a new supernodal algorithm. Their results show that the performance of this new algorithm on this matrix is superior to that of a supernodal fan-out and a supernodal fan-in algorithm. They argue that the new algorithm, referred to as a supernodal aggregated source-driven algorithm, can be expected to have good local efficiency, low communication, as well as good load balance. Although the results in this paper are preliminary, they look promising. It remains to be seen as to how the algorithm would perform on a variety of structured and unstructured matrices.

To the best of our knowledge, the best comprehensive performance results for sparse Cholesky factorization for both unstructured and structured matrices on the iPSC/860 are obtained from a distributed multifrontal algorithm developed by Sun [69]. This
algorithm is an improvement on an earlier version designed by Pothen and Sun [57]. We will refer to the latter version as the old multifrontal code and the former as the new multifrontal code. We compare the performance of our block factorization code with both the old and new multifrontal codes for the test cases B13 and B15 to gain a rough idea of where the block code stands at this point.

It is only a rough idea, for several reasons. Only the factorization times are presented by Sun in [69]. The communication, idle and overhead times are not given, so it is hard to get much insight into the general behavior of that algorithm. Although the B13 and B15 matrices in [69] are ordered using minimum degree ordering, the number of non-zeros in the factor is 271671 for B13 and 651222 for B15. In both cases, the number of nonzeros is 2% more than those reported in Table 7.1. The number of floating point operations reported by Sun is 59.72 million to factor B13 and 167.39 million to factor B15. We report 56.55 and 157.58 for the same test cases, respectively. Sun defines a floating point operation as either an add or a multiply, which is the same as our definition. Nothing is said about the square root and reciprocal operations. Both Sun’s code as well as ours are written in C and run on the iPSC/860. However, Sun uses double precision arithmetic, whereas we use single precision arithmetic for the numerical operations performed towards factorization. Both use the Kuck math library routines - the multifrontal code uses daxpy, ddot and dscal, while the block code uses sdot. Heath, Geist and Drake [32] give execution rates for sdot and ddot on the i860. Their assembler version ddot is about 1 Mflops slower than their assembler version of sdot for vector lengths up to 1000. They also give asymptotic execution rates for row-Cholesky, which uses dot product. In this case the assembler version ddot delivers 22.4 Mflops while the assembler version sdot delivers 24.6 Mflops. Thus the double precision arithmetic rate is only slightly lower than the single precision arithmetic rate.

In Figure 7.19 we compare the performance of the block code with both the old and the new multifrontal codes. The curve for the block code is much better than that for the old multifrontal code, especially with increasing numbers of processors. The block code compares very well with the new multifrontal code.
7.9 Selection of partitioning parameters

The selection of the partitioning parameters $W$ and $G$ is crucial for obtaining good performance. In this section we show how these parameters can be selected based on knowing how they affect the different cost components of the parallel time. The results presented here are extracted from experiments we conducted on an earlier version of our numerical factorization code [75]. The differences of note from the current version are (a) the task computational cost in the earlier version was higher for the RRR type tasks, and (b) the communication was not optimized in the earlier version. However, in the current absence of automated selection of parameters, it is important to present the results in the sequence in which the experiments were conducted, since they illustrate our experimentation procedure to decide the parameter values.

7.9.1 Cluster width $W$

The effect of varying $W$ on the total factorization time, the computation time, the communication time, and the idle time are shown in Figures 7.20, 7.21, 7.22, and 7.23, respectively. In all cases, the grain size parameter, $G$, was set to 200. For all the test matrices, the observed trend is that as $W$ is made smaller, the factorization time, and the computation and communication time components decrease. This is most noticeable in the five unstructured B* test matrices. In the case of L16, which is a structured matrix, the effect of $W$ is not so prominent. However, even in that case, reducing $W$ from 15 to 5 resulted in about 5% improvement in the overall performance. Although not shown, in all cases, similar trends are observed in the overhead time.

The effect of $W$ can be explained as follows. As described in Section 5.3.3, the cluster width $W$ is a parameter used to switch between column-based partitioning and block-based partitioning in different regions of the matrix. Only the clusters that are at least $W$ columns wide are accepted for block partitioning. A high value of $W$ results in a partitioning which consists of a relatively large number of single sparse columns and a small number of large dense blocks. With a very large value of $W$, the resulting partitioning is identical to the commonly used column-based partitioning method. On
the other hand, a smaller value of \( W \) results in the clustering of several columns; this
gives rise to a smaller number of single columns and a large number of dense blocks
which are partitioned into many rectangular and triangular unit blocks.

When there are a large number of single column partitions, there are a relatively
large number of CC, CT and CR types of tasks. When there are a large number of block
partitions, there are a relatively large number of tasks which only involve triangles or
rectangles. For the six test matrices, we show in Figures 7.24 and 7.25, the distribution
of the number of CC and RRR tasks, respectively. In those figures, for each test case,
a pair of bars is shown; the left one corresponds to \( W = 5 \) and the one on the right
corresponds to \( W = 15 \). Clearly, when \( W \) is decreased from 15 to 5, there is a significant
reduction in the number of CC tasks. On the other hand, with a smaller \( W \), there are
more RRR tasks. We observed similar trends in the relative distribution of other types
of tasks involving columns, and other tasks involving only dense blocks.

Now consider the computational speeds achievable with each type of task. In Fig-
ure 7.26, we show the single floating point operation cost, in microseconds, for six
different tasks. Figure 7.26(a) gives the cost of floating point operations for factoring
B15 on eight processors. Comparing these with the computational task rates for B15
presented in Table 7.7, the old RRR tasks were twice as slow as the latest version.
Figure 7.26(b) gives the average single floating point operation cost, in microseconds,
for the six different tasks; for each type of task, the averages were taken over all test
matrices. Clearly, the computational speed in the tasks involving only triangular or
rectangular blocks is about four to five times greater than that in the tasks involving
columns. In the new version of the task computational codes, this difference is even
greater. With a lower value of \( W \), a larger number of computationally efficient tasks
are obtained. There is a larger number of dense blocks as compared to sparse columns,
leading to a larger number of tasks involving dense blocks as compared to tasks involv-
ing sparse columns. These experiments led us to choose \( W = 5 \) as the cluster width for
the test results presented in this chapter.
7.9.2 Grain size $G$

For the results presented here, $W$ was set to 5. We first present a result showing the effect of the grain size on uniprocessor performance. Figure 7.27 for the case of B28 shows that the total factorization time, computation time and overhead time all decrease with increase in grain size. The rate of decrease gradually tapers off as the grain size becomes larger. Since the locality of computation improves with increase in grain size, we see a decrease in computation time. Similarly, the bigger the grain size, the smaller is the number of blocks, resulting in a decrease in task scheduling overhead. For any given application, there is a point beyond which increasing the grain size results in a negligibly small number of additional blocks being created. At this point the number of blocks reaches saturation point and the effect of increasing the grain size is not seen any more.

Now consider the effect of $G$ when the number of processors is increased. We present the results for B15. Figure 7.28 shows the total factorization time, and Figures 7.29, and 7.30 show the results for the average per-processor communication and idle times, respectively. Observe from Figure 7.28 that for any given number of processors, the total factorization time first decreases with increasing grain size, but at a certain point it "bottoms out" and then starts increasing with increasing grain size. This behavior is explained by considering the trade-off between communication time and idle time. As $G$ increases, the communication time per processor decreases since there are now fewer messages because of increased locality. This tendency is shown in Figure 7.29. But at the same time, since there are fewer and larger blocks, the distribution of work becomes uneven in time and each processor shows increased "starvation" leading to increased idle times. This is borne out by Figure 7.30. At a certain point the increase in idle time outstrips the decrease in communication time. If the computation time has also saturated in the meanwhile, as described in the earlier discussion on uniprocessor performance, a further increase in $G$ would only serve to increase the factorization time.

Another observation we make here is that the larger the number of processors, the faster is the increase in factorization time with increasing grain size. This is because
as the tasks become fewer and larger, balancing the work load is more difficult with
a larger number of processors, resulting in increased idle times and hence worsening
factorization times. To summarize, the experiments that we conducted show that for
every given number of processors used, there is, in general, a different grain size at which
the communication time is balanced against the idle time. This grain size is usually
smaller for a larger number of processors, and gradually increases as the number of
processors is decreased. Apart from this, we discovered that it was beneficial to choose
a grain size $T$ for the triangles which was smaller than the grain size $R$ for the rectangles,
and for most of the test cases we dealt with, setting $T$ to 400 and $R$ to 800 resulted in
better performance than all other grain sizes we experimented with.

7.10 SHAPE: Preprocessing cost

The overall preprocessing time is the sum of the individual times for pre-partitioning,
partitioning, allocation and scheduling. The partitioner and scheduler are both parallel
programs. The pre-partitioner and allocator are sequential programs that run on a
single processor of the iPSC/860. All programs are written in C. All the times are in
seconds and do not include the time spent in input and output.

The test matrices were all partitioned with the cluster width parameter $W$ set to 5,
the triangle grain size parameter $T$ set to 400 and the rectangle grain size parameter
$R$ set to 800. Table 7.13 shows the timings for the parallel partitioner and parallel
scheduler. In each case, the timing is for the processor which finished last. In each
case, the scheduler was run with numerical computations i.e. in the non-symbolic mode.
The timing shown for the scheduler has been obtained by subtracting the computation
time from the total time for the processor which finished last. Due to lack of memory
space, the scheduler could not be run on a single processor for any of the test matrices
considered. Timings for the pre-partitioner and allocator are not shown here. Together,
they contribute less than 10% of the single processor time required for partitioning and
scheduling.

A blank entry means that the corresponding test could not be run due to lack of
memory space on the processor. As described in section 5.5.3, the partitioner has been parallelized mainly to distribute data so that the partitioner scales with the problem size. The emphasis is not on efficient computational parallelization. The time required by the partitioner is a function of the number of dependencies reported. In parallelizing the partitioner, columns and clusters are allocated in an equitable manner to processors. However, this does not imply an equitable distribution of the computational work involved in determining the inter-block dependencies. Table 7.13, therefore, does not show good speed-up figures for the partitioner.

We also see poor speed-up figures for the scheduler. This is not unexpected since the timings represent purely symbolic computation. The entire computational effort is directed towards symbolic manipulation of data structures. Communication is heavily comprised of simply passing symbolic dependency information. Non-numerical computations such as these typically perform poorly on message-passing machines. Just as in the case of the partitioner, the main intent in parallelizing the scheduler is to have it scale with the problem size in terms of per-processor memory utilization.

Table 7.14 shows the size of each test case in terms of the number of clusters, number of columns, number of dense triangles, number of dense rectangles, total number of blocks (sum of columns and dense blocks), and number of dependencies.

From Table 7.14 and Table 7.13, it is clear that the partitioning time is proportional to the number of dependencies in the BDAG. Also, the number of dependencies is an order of magnitude greater than the number of blocks, for all the test matrices. Thus the construction of dependencies is the major computational work, compared to which

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Partitioner</th>
<th>Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>B15</td>
<td>9.2</td>
<td>5.2</td>
</tr>
<tr>
<td>B16</td>
<td>8.5</td>
<td>5.6</td>
</tr>
<tr>
<td>B17</td>
<td>12.5</td>
<td>9.4</td>
</tr>
<tr>
<td>L20</td>
<td>25.1</td>
<td>14.9</td>
</tr>
<tr>
<td>L25</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>L127</td>
<td>17.7</td>
<td>10.6</td>
</tr>
</tbody>
</table>

Table 7.13: Partitioner and scheduler timings.
the amount of work in partitioning into unit blocks is insignificant. Our decision to have each processor compute the entire block partitioning structure, instead of having them exchanging locally computed structures, is consistent with the above observation.

Figure 7.31 shows the ratio of preprocessing to factorization times for the unstructured and structured matrices, on 32 processors. The preprocessing time is the sum of the partitioning and scheduling times obtained from the respective 32-processor columns in Table 7.13, and the factorization times are obtained from Tables 7.2 and 7.3. Except for L20, the preprocessing time is between two and three times as much as the factorization time. This implies that the preprocessing cost can be recovered by performing two to three iterations of the factorization. There are several applications including linear programming in which the factorization needs to be performed dozens of times with the same sparsity structure but different data values. In these applications, the preprocessing cost is a small price to pay for the improvement in performance. For the L20 case, the preprocessing time is about four times as much as the factorization time.

With further improvement in the factorization time, the ratio of preprocessing to factorization would increase unless the preprocessing code itself is further tuned. However, the preprocessing step is symbolic in nature, and typically symbolic algorithms scale very poorly on large-scale multiprocessors. It might be more beneficial to try duplicate computations on the processors, instead of excessive communication of symbolic information. Also, currently, the number of dependencies is excessively disproportionate to the amount of computational work done for the L20 and L127 cases, and for structured matrices in general. This is because of the large number of rectangles which

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Clusters</th>
<th>Col</th>
<th>Tri</th>
<th>Rect</th>
<th>Total</th>
<th>Deps</th>
</tr>
</thead>
<tbody>
<tr>
<td>B15</td>
<td>112</td>
<td>1337</td>
<td>257</td>
<td>1600</td>
<td>3194</td>
<td>80160</td>
</tr>
<tr>
<td>B16</td>
<td>156</td>
<td>1319</td>
<td>473</td>
<td>1667</td>
<td>3459</td>
<td>80354</td>
</tr>
<tr>
<td>B17</td>
<td>538</td>
<td>3413</td>
<td>778</td>
<td>3701</td>
<td>7892</td>
<td>103918</td>
</tr>
<tr>
<td>L20</td>
<td>291</td>
<td>3917</td>
<td>576</td>
<td>6460</td>
<td>10953</td>
<td>256475</td>
</tr>
<tr>
<td>L25</td>
<td>562</td>
<td>7410</td>
<td>852</td>
<td>13914</td>
<td>22176</td>
<td>646545</td>
</tr>
<tr>
<td>L127</td>
<td>678</td>
<td>8580</td>
<td>777</td>
<td>6648</td>
<td>16005</td>
<td>182167</td>
</tr>
</tbody>
</table>

Table 7.14: Size of partitioning.
are smaller than the required grain size, as explained in Section 7.5.1. Since the amount of work involved in partitioning and scheduling is mainly determined by the number of dependencies, preprocessing for structured matrices consumes more time than preprocessing for unstructured matrices. However, with improvement in blocking to achieve a better grain size using the technique proposed in Section 7.5.3, we hope to see better preprocessing performance.

7.11 Conclusions

We have shown in this chapter that 2D block-based partitioning and scheduling for sparse Cholesky factorization can deliver high performance, with an appropriate selection of the partitioning parameters, provided the matrix is amenable to blocking. This last proviso is illustrated in our experiments, where we observed that the sparsity pattern of unstructured test matrices is in general more conducive to blocking, compared to structured grid matrices. We have also shown that the our partitioning and scheduling method is practical. The preprocessing cost is recovered within very few iterations of the numerical factorization. Currently, the partitioning parameters are manually chosen based on experiments. In future work, we plan to automate the process of parameter selection.

The strengths of our methodology are: (1) it is extremely flexible in partitioning the matrix and offers extensive parametric control so that the partitioning is sensitive both to the matrix sparsity structure and to the architectural granularity; (2) it is modular, so that separate investigation into different aspects of the problem such as mapping and scheduling may be pursued independently and the solution easily integrated with the other modules by using BCR; (3) it offers a set of tools that alleviate the coding effort and hide the details of the architecture from the user; (4) SHAPE provides a solid base for the investigation of the problem, with fast turnaround time. Though we have only investigated a specific 2D block partitioning method, column partitionings and other types of 2D block partitionings can also be investigated using SHAPE.

The flexibility in our approach gives rise to some limitations which do not arise in
implementations that are specifically designed for sparse Cholesky factorization. One such limitation is the additional memory space required to store explicit dependencies. We proposed coalescing of rectangles as one way to reduce the number of dependencies while increasing the grain size, and thereby reduce the memory space requirements. Apart from this, with growing memory space available on the nodes of recent distributed memory multiprocessors such as the IBM SP1, which has up to 128 megabytes of memory on each processing node, we expect the memory constraint to be less of a problem. Another limitation is the preprocessing time required for the partitioning and scheduling. Our approach is applicable to problems in which the Cholesky factorization needs to be done several times with the same matrix structure but different data values, so that the preprocessing cost can be recovered in the first few iterations. We also propose to tune the preprocessing further to reduce its cost.

Currently, the idle time is the primary performance bottleneck in the factorization. We plan to reduce the idle time by pursuing, in greater depth, the allocation and scheduling for the block method. We outlined some of our plans in our discussion of the idle time. We plan to port our investigation to other machines to further understand the relative contributions to performance and the limitations of both the target machines and the sparsity structure of the problem instances. The goal is to determine a set of pairs of problem sizes and machine architectures at which 50% efficiency is achievable, and to deduce whether these pairs are realistic. In other words, we want to compare the scalability of different large-scale multiprocessor architectures for sparse Cholesky factorization.

One of the issues that comes up in testing a new sparse Cholesky factorization algorithm is: what test matrices to use? The unstructured nature of the problem demands extensive testing on a variety of matrices, but there are no standard test matrices that are used to demonstrate performance. As a consequence, it is hard to compare results with other existing methods on an extensive set of common test matrices. We have chosen to test our approach on a host of matrices, the choice being restricted by availability of the matrices, and the availability of memory space on the processors of the 8MB of memory per node iPSC/860.
Finally, schemes to reorder the unknowns and equations of the sparse linear system have a role to play in the generation of parallelism for Cholesky factorization. The idea is that the ordering should result in an elimination tree, (or a dependency graph, in general), that is short and wide rather than tall and skinny, thus providing more parallelism. Two well-known ordering schemes are the multiple minimum degree (MMD) [42] and automatic nested dissection (AND) [28], which were aimed at reducing the fill-in during the factorization, so as to ultimately reduce the computational work. In recent years, the aggressive effort in parallelizing sparse Cholesky factorization has resulted in renewed investigation for designing new ordering schemes which would generate more parallelism for factorization. However, this parallelism has to be traded off against a possible increase in the fill-in, and subsequently in the computational work. A good ordering scheme for parallel factorization would therefore be one that would create a significantly larger amount of added parallelism for factorization, without giving away too much additional computational work. Recent ordering algorithms include spectral nested dissection (SND) [56] and a cartesian nested dissection (CND) [34]. The SND algorithm has been shown to incur fill-in that is quite close to MMD and much smaller than AND. It outperforms both MMD and AND with respect to parallelism offered by the elimination tree for factoring. The CND algorithm has been shown to incur fill-in that is somewhat worse than both MMD and AND. It also produces shorter elimination trees than MMD and AND. The performance of both SND and CND have been shown on very few problems. Besides, it is not clear as to how the performance of parallel factorization itself is improved. Last, but not the least, it is not known as to how the new ordering schemes would affect dense block computations in factorization.
Figure 7.14: Optimized communication cost.
Figure 7.15: Optimized communication relative to computation.

Figure 7.16: Factorization time before and after communication optimization, 32 processors.
Figure 7.17: Factorization time before and after communication optimization, B15 and L127.
Figure 7.18: Idle time before and after communication optimization.
Figure 7.19: Comparison of block with multifrontal.

Figure 7.20: Effect of W on total factorization time.

Figure 7.21: Effect of W on computation time.
Figure 7.22: Effect of $W$ on communication time.

Figure 7.23: Effect of $W$ on idle time.

Figure 7.24: Number of CC tasks.

Figure 7.25: Number of RRR tasks.
Figure 7.26: Cost per floating point operation.

Figure 7.27: Effect of $G$ on uniprocessor performance.

Figure 7.28: Effect of $G$ on factorization time.
Figure 7.29: Effect of $G$ on communication time.

Figure 7.30: Effect of $G$ on idle time.

Figure 7.31: Preprocessing and factorization times.
Chapter 8
Sparse triangular systems and sparse matrix-vector multiplication

In the previous chapters, we have seen how our parallelization methodology of partitioning and scheduling can be applied to block sparse Cholesky factorization. In this chapter, we discuss the application of the methodology to the solution of sparse triangular systems following Cholesky factorization, and to sparse matrix-vector multiplication.

At the end of the Cholesky factorization, the factor is available in the lower triangular matrix $L$. If the factorization is performed in order to solve a linear system $Ax = b$, then the lower triangular system $Ly = b$ and the upper triangular system $L^Tx = y$ have to be solved after the factorization. We present new algorithms for solving the lower and upper triangular systems when $L$ is sparse. In the literature these are alternatively referred to as the forward and back solves, respectively. We assume that the block partitions of $L$, and the mapping of these blocks to processors are available. We describe the algorithms in terms of the BCR. Given the BCR, and the mapping of blocks to processors, the parallel algorithms can be implemented using our tools for scheduling and communication optimization. The primary aim is to illustrate the application of the methodology, so the algorithms we propose are not necessarily competitive with other existing algorithms for the solution of sparse triangular systems [81, 24, 69, 41].

The notations used in this chapter for the block partitions of $L$ are defined in Chapter 5. We assume also that the triangular systems need to be solved for several right hand side vectors $b$, so that the run-time preprocessing cost is amortized over the iterations of the solves.

For sparse matrix-vector multiplication, we sketch how the methodology can be applied to parallelize an algorithm recently proposed by Agarwal, Gustavson and Zubair [1].
Their algorithm is based on preprocessing the sparse matrix to extract a block-based structure, and using the extracted structure to perform several iterations of the multiplication.

8.1 Block sparse triangular systems

The partitioning of the matrix $L$ implicitly defines a partitioning of the matrix $L^T$, and the block partitions are already available after factorization. Also, the mapping of the blocks of $L$ to processors is available, and is used in both the lower and upper triangular system solutions. The vectors $b$, $x$, and $y$ are all partitioned in an identical manner as follows. We use $D$ to denote a generic vector standing for $b$, $x$ or $y$. A single-element $D_k$ is a partition in $D$ if and only if column $k$ is a sparse column partition in $L$. The elements $D_m$ through $D_n$, denoted by $D_{m,n}$, form a partition in $D$ if and only if $T_{m,n}$ is a dense triangular partition in $L$. We assume that the mapping of these block partitions is done separately.

In the rest of this section, we use the term diagonal $D_k$ to mean the single-element partition $D_k$ of any of the vectors $b$, $x$ and $y$, and the term diagonal $D_{m,n}$ to mean the partition of $b$, $x$ or $y$ which consists of the elements $D_m$ through $D_n$.

8.1.1 Dependencies

The element $y_i$ of the solution $y$ in the dense forward solve, and the element $x_i$ of the solution $x$ in the dense backward solve are respectively computed as follows:

$$y_i = \left( b_i - \sum_{j=1}^{i-1} L_{i,j} \cdot y_j \right) / L_{i,i}, \quad x_i = \left( y_i - \sum_{j=i+1}^{n} L_{i,j}^T \cdot x_j \right) / L_{i,i}^T.$$

For the sparse solutions, the above equations are modified so that the computations in which $L_{i,j}$ are zero are skipped over. The element level dependencies follow directly from the above equations. In the forward solve, the element $y_i$ depends on $b_i$, $L_{i,i}$ and on each pair $(y_j, L_{i,j})$, where $j < i$ and $L_{i,j} \neq 0$. In the back solve, the element $x_i$ depends on $y_i$, $L_{i,i}^T$ and on each pair $(x_j, L_{i,j}^T)$, where $j > i$ and $L_{i,j}^T \neq 0$. We now describe the dependencies and the computational tasks at the block level.
Lower triangular system

First, consider the system $Ly = b$. The values of the elements of $y$ are initialized to the values of the corresponding elements of $b$. There are three types of dependencies or updates that can be defined on the blocks. In the following, the diagonal $D$ stands for the vector $y$. The dependencies are illustrated in Figure 8.1.

![Figure 8.1: Dependencies in block forward solve.](image)

- **Diagonal and column update diagonal, $<\text{Diag, Update, Diag, Col}>$**
  Let block $A$ be column $k$, block $B$ be diagonal $D_k$ and block $C$ be diagonal $D_{p,q}$. For $A, B \in C$ to be true, $\exists (P_1 \in A)$ such that $\mathcal{H}(P_1) \cap D_{p,q} \neq \emptyset$. This is satisfied for all $P_1 = L_{i,k}$, $p \leq i \leq q$. The corresponding computational task, $DCD$, involves subtracting the product of $D_k$ and $L_{i,k}$ from the corresponding element $D_i$, for $p \leq i \leq q$ and $L_{i,k} \neq 0$. This update is illustrated in Figure 8.1(a). The dark circle is the element $D_k$. The thick line is the diagonal $D_{p,q}$. Three non-zero elements of the column $k$, in rows $i1, i2$ and $i3$, update the corresponding elements of $D_{p,q}$.

- **Diagonal and rectangle update diagonal, $<\text{Diag, Update, Diag, Rect}>$**
  Let block $A$ be diagonal $D_{m,n}$, block $B$ be rectangle $R_{r,r',c,c'}$ and block $C$ be diagonal $D_{p,q}$. For $A, B \in C$ to be true, $[m, n] \cap [c, c'] = [u, u'] \neq \emptyset$ and $[r, r'] \cap [p, q] = [t, t'] \neq \emptyset$. The corresponding computational task, $DRD$, involves subtracting from element $D_i$, the dot product of $D_{u,u'}$ and vector $R_{i,u,u'}$, for $t \leq i \leq t'$. In Figure 8.1(b), the shaded portion of the rectangle combines with
the portion of the diagonal $D_{m,n}$ between the vertical dotted lines to update the portion of $D_{p,q}$ between the horizontal dotted lines.

- **Triangle updates diagonal, $<\text{Diag}, \text{Update}, \text{Tri}>$**

  Let block $A$ be triangle $T_{m,n}$ and block $B$ be diagonal $D_{p,q}$. For $A \delta B$ to be true, $[m, n] \cap [p, q] = [t, t'] \neq \emptyset$. By definition of the partitioning, however, $[m, n] = [p, q]$, so $[t, t'] = [m, n]$. The corresponding computational task $TD$ involves subtracting from each element $D_i$, the dot product of $D_{m, i-1}$ and the row $i$ of $T_{m,n}$ save the diagonal element, followed immediately by setting $D_i$ to $D_i / L_{i,i}$, for $m \leq i \leq n$, in ascending order of $i$ values. In Figure 8.1(c), the entire shaded triangle updates the diagonal $D_{m,n}$. $TD$ is executed only after all other tasks which update $D_{m,n}$ have finished execution.

**Upper triangular system**

Now consider the system $Ux = y$. The values of the elements of $x$ are initialized to the values of the corresponding elements of $y$ which are available at the end of the solution of the lower triangular system. There are three types of dependencies or updates. These are shown Figure 8.2. These dependencies are obtained by considering the transpose of each of the blocks in the lower triangular system, and tracing the computations from bottom to top in the matrix. Note that the transpose of a block is imagined just for illustrative purposes; no new block is actually created. In the following we use the notation $B^T$ to represent the transpose of a block $B$ of the factor, and diagonal $D$ to
represent the vector \( z \).

- **Diagonal and row update diagonal, \( <Diag, Update, Diag, Col^T> \)**
  Let block \( A \) be row \( k \), block \( B \) be diagonal \( D_{p,q} \) and block \( C \) be diagonal \( D_k \). For \( A, B \in C \) to be true, \( \exists (P_1 \in A^T) \) such that \( HP(P_1) \cap D_{p,q} \neq \emptyset \). This is satisfied for all \( P_1 = L_{k,i}^T, p \leq i \leq q \) and \( L_{k,i}^T \neq 0 \). The corresponding computational task, \( D C^T D \), involves subtracting from \( D_k \), the dot product of the vector \( L_{k,i}^T, p \leq i \leq q \), \( L_{k,i}^T \neq 0 \), and the vector composed of the corresponding elements \( D_i \) of \( D_{p,q} \). This update is illustrated in Figure 8.2(a). The dark circle is the element \( D_k \). The thick line is the diagonal \( D_{p,q} \). Three non-zero elements of the row \( k \), in columns \( i_1, i_2 \) and \( i_3 \), combine with the corresponding non-zero elements of \( D_{p,q} \) to update \( D_k \).

- **Diagonal and rectangle update diagonal, \( <Diag, Update, Diag, Rect^T> \)**
  Let block \( A \) be diagonal \( D_{p,q} \), block \( B \) be rectangle \( R_{r,r',c,c'} \) and block \( C \) be diagonal \( D_{m,n} \). For \( A, B \in C \) to be true, \( [m, n] \cap [c, c'] = [u, u'] \neq \emptyset \) and \( [r, r'] \cap [p, q] = [t, t'] \neq \emptyset \). The corresponding computational task \( DR^T D \) involves subtracting from the vector \( D_{u,u'} \), the product of element \( D_i \) with the vector \( R_{i,i,u,u'}^T \), for \( t \leq i \leq t' \). In Figure 8.2(b), the shaded portion of the rectangle combines with the portion of the diagonal \( D_{p,q} \) between the vertical dotted lines to update the portion of \( D_{m,n} \) between the horizontal dotted lines.

- **Triangle updates diagonal, \( <Diag, Update, Tri^T> \)**
  Let block \( A \) be triangle \( T_{m,n}^T \) and block \( B \) be diagonal \( D_{p,q} \). For \( A \in B \) to be true, \( [m, n] \cap [p, q] = [t, t'] \neq \emptyset \). By definition of the partitioning, however, \( [m, n] = [p, q] \), so \( [t, t'] = [m, n] \). The corresponding computational task \( T^T D \) involves, going in descending order of \( i \) values for \( m \leq i \leq n \), setting \( D_i \) to \( D_i / L_{i,i}^T \), followed immediately by subtracting from the vector \( D_{m,i-1} \), the product of element \( D_i \) with the column \( i \) of \( T_{m,n}^T \), save the diagonal element. In Figure 8.2(c), the entire shaded triangle updates the diagonal \( D_{m,n} \). \( T^T D \) is executed only after all other tasks which update \( D_{m,n} \) have finished execution.
In the rest of this section, we use the term partial product or simply product to mean any of the products defined in the computational tasks described above.

8.1.2 A hybrid fan-out and fan-in algorithm

A fan-out type algorithm based on the dependencies cited above for either the lower or the upper triangular system has the following communication requirement in the worst case. Each updating block contributes one message, sent from the owner of the updating block to the owner of the updated block. A triangle-updates-diagonal type task requires one message; the other two types of tasks require two messages each. We now look at the volume of communication and weigh it against the space required for the result of the product operations, in a bid to reduce the communication volume by trading it off against additional space. The aim is to distribute among the processors, if profitable, the computation of the products required to update a given partition of the diagonal and then fan the products into the owner of that partition, where they can be incorporated into the partition. This approach is similar in spirit to the fan-in column method for distributed sparse Cholesky factorization, and is described in the following.

In a $DCD$ or $DC^T D$ type task, the size of a product is linear in the number of non-zeros used to compute the product. In a $DRD$ type task in the forward solve, however, the size of the product is equal to the height of the rectangle $R$, and in a $DR^T D$ type task in the back solve, the size of the product is equal to the height of the rectangle $R^T$, which is the same as the width of the rectangle $R$. In either case, therefore, the dimension of the product is one less than the dimension of the rectangle used to compute the product. The task $DRD$ (analogously $DR^T D$) may be divided into two tasks. In the first task, $DR$ (analogously $DR^T$), the product of the diagonal and rectangle is computed and stored along with the rectangle. In the second task, $RD$ (analogously $R^T D$), the stored product is subtracted from the appropriate diagonal partition. Additional space is required to store these products. In the forward solve, for each rectangle $R$, the additional space required is equal to the height or row extent of $R$. In the upper triangular solve, the additional space required for each rectangle $R^T$
is equal to the height of $R^T$, which in turn is equal to the width of the rectangle $R$ in the factor. There may be several $D R D$ (or $D R^T D$) type tasks which use rectangle $R$ (or $R^T$). In this case, the products of the individual $D R$ (or $D R^T$) tasks are accumulated in the additional storage space, and when all these tasks are completed, the final product is subtracted from the appropriate diagonal partition.

To weigh the tradeoff of the expected savings in worst-case communication cost against the expected increase in space requirement, we computed several size-related measures for the rectangular blocks in a variety of matrices chosen from those listed in Table 7.1. These quantities are tabulated in Table 8.1. For all the test cases, $W$,

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Rect Nonzeros</th>
<th>Rect Heights Nonzeros</th>
<th>Rect Widths Nonzeros</th>
<th>$\lambda_{comm}$</th>
<th>$\lambda_{comm}$</th>
<th>Space %age</th>
</tr>
</thead>
<tbody>
<tr>
<td>B13</td>
<td>151490</td>
<td>6500</td>
<td>37246</td>
<td>24.59</td>
<td>16.47</td>
<td></td>
</tr>
<tr>
<td>B15</td>
<td>435202</td>
<td>15198</td>
<td>68684</td>
<td>15.78</td>
<td>13.15</td>
<td></td>
</tr>
<tr>
<td>B16</td>
<td>515192</td>
<td>18235</td>
<td>70966</td>
<td>13.77</td>
<td>11.91</td>
<td></td>
</tr>
<tr>
<td>B17</td>
<td>716532</td>
<td>38579</td>
<td>85895</td>
<td>11.99</td>
<td>12.19</td>
<td></td>
</tr>
<tr>
<td>L16</td>
<td>164740</td>
<td>8806</td>
<td>38854</td>
<td>23.59</td>
<td>18.52</td>
<td></td>
</tr>
<tr>
<td>L20</td>
<td>442148</td>
<td>21188</td>
<td>106225</td>
<td>24.02</td>
<td>19.70</td>
<td></td>
</tr>
<tr>
<td>L25</td>
<td>1216278</td>
<td>45853</td>
<td>318600</td>
<td>26.19</td>
<td>21.20</td>
<td></td>
</tr>
<tr>
<td>L127</td>
<td>350254</td>
<td>23579</td>
<td>93509</td>
<td>26.70</td>
<td>20.78</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.1: Communication-space tradeoffs for block triangular solves.

the minimum acceptable cluster width, was set to 5 for the block partitioning. The rectangle grain size parameter, $R$, was set to 800.

"Rect Nonzeros" is the total number of non-zeros contained in all the rectangular blocks. This is the volume of communication in the worst case for a fan-out approach. The "Rect Heights" entries pertain to the forward solve and the "Rect Widths" entries pertain to the back solve. Under "Rectangle Heights", the entry "Nonzeros" is the sum, over all rectangular blocks, of the number of non-zero elements along the vertical side of each rectangular block. The entry $\lambda_{comm}$ is the new communication volume incurred in transferring only the products, as a percentage of the volume incurred in transferring entire rectangles; this is obtained by taking the ratio of the rectangle height non-zeros to the rectangle non-zeros and multiplying the ratio by 100. For instance, in the case of B15, the new volume of communication is barely 3.5% of the volume that would
be communicated were the fan-out approach used and the entire rectangles transferred. Analogously, for the back solve, under "Rect Widths", the entry "Nonzeros" is the sum, over all rectangular blocks, of the number of non-zero elements along the horizontal side of each rectangular block. The entry $\lambda_{comm}$ is the new communication volume incurred in transferring only the products, as a percentage of the volume incurred in transferring entire rectangles; this is obtained by taking the ratio of the rectangle width non-zeros to the rectangle non-zeros and multiplying the ratio by 100. For instance, in the case of B17, the new volume of communication is only 12% of the volume that would be communicated were the fan-out approach used and the entire rectangles transferred. Finally, the entry "Space %age" is the amount of additional space required for the triangular solves as a percentage of the memory space required for the original matrix data. The space required for the original matrix data is equal to the number of non-zeros in the factor, ignoring the row indexing for sparse columns. The percentage space requirement is computed by taking the ratio of the sum of the rectangle width and height non-zeros from the third and fifth columns of the table, to the factor non-zeros listed in Table 7.1, and multiplying the ratio by 100.

From the table, it is evident that for solving the lower triangular system, significant savings in communication volume are obtained by dividing the $DRD$ tasks into $DR$ and $RD$ tasks, and communicating only the products. For the upper triangular system, the communication savings are relatively less impressive, but are still significantly high. Finally, the additional space requirement is very low, especially for the unstructured matrices. We conclude that, in general, it is profitable to incur the cost of additional space to gain the advantage of reduced communication.

As in the tasks involving the product of a diagonal and a rectangle, the task $TD$ also computes a product that is a dimension less than the participating triangle. However, we cannot divide the $TD$ task into two tasks in such a manner that the product is computed in the first, and the update to the diagonal is performed in the second. The reason for this is as follows. $TD$ is the last of the tasks to update a diagonal; in each step, it computes one more element of the solution by performing a division, and then uses that element to update the triangle. The division and
update steps are interleaved with each other. Thus, we cannot break \( TD \) into two
tasks, one of which begins after the other is completely done. We retain the \( TD \) type
task as is, and send the triangle to the processor that owns the diagonal partition.
This would incur relatively little communication overhead compared to communicating
rectangles, since, for all the test cases examined, the number of non-zeros contained in
the triangular blocks is insignificant compared to the number of non-zeros contained in
the rectangular blocks. We expect this to hold for most matrices in practice.

To summarize, our algorithm for lower and upper triangular solves use the original
matrix partitioning constructed for the factorization, with additional storage space as-
signed for the rectangles during preprocessing. A hybrid fan-in, fan-out type method is
implemented with consequent savings in communication and affordable use of additional
memory space. The fan-in type computations are performed for the diagonal-updates-
rectangle and rectangle-update-diagonal tasks, while the fan-out type computations are
performed for all the other tasks. The BCR is presented in the following section.

### 8.1.3 BCR

The basic computational tasks or dependencies have already been defined in Sec-
tion 8.1.1. For the hybrid method, we divide the \( DRD \) and \( DR^TD \) tasks into two
parts each. Before we define these new tasks, we describe the AVR for blocks.

**AVR for blocks**

The lower and upper triangular systems use the same set of blocks. There are four types
of blocks: columns, rectangles, triangles and diagonals. Of these, the AVR for columns
and triangles are unchanged from the factorization, as described in Section 5.4.7. A
column is treated as a row by the upper triangular system, with the same CSC storage
scheme. The storage for a triangle remains unchanged; the data values are in row major
order, and for the upper triangular system, the storage is effectively in column-major
order since it works with the transpose of the triangle. The attributes for a diagonal
partition \( D_{m,n} \) are \((m, n)\); for a single-element partition \( D_k \), \( m = n = k \). The data is
stored in one contiguous block.
A rectangle is composed of two parts - one part is the original block used in the factorization, and the other part is the extra space added on. This is illustrated in Figure 8.3(a). In that figure, the shaded area attached to the right end of the rectangle

\[ \begin{array}{c}
(a) \\
(b) \\
(c)
\end{array} \]

Figure 8.3: Rectangle for triangular systems.

is the extra storage vector of size equal to the height of the rectangle. This space is used by the forward solve as shown in Figure 8.3(b), where several $DR$ type tasks accumulate their respective products into the vector. Similarly, the shaded area attached to the top is the extra storage vector of size equal to the width of the rectangle. This space is used by the back solve as shown in Figure 8.3(c), where the transpose of the rectangle is used in several $DR^T$ type tasks, and the respective products accumulated in the vector.

Consider a rectangular block $R_{r1,r2,c1,c2}$. The attributes for the new rectangular block are the same as that of $R_{r1,r2,c1,c2}$, i.e. $(r1, r2, c1, c2)$. The data for the new rectangular block is split into three groups. The first group is the same as the data for $R_{r1,r2,c1,c2}$, with the data stored in row-major order. The second group is the extra storage space of size $(r2 - r1 + 1)$ for the forward solve, and the third group is the extra storage space of size $(c2 - c1 + 1)$ for the back solve. The data elements in each group are stored in a contiguous fashion in memory, but the different groups are not necessarily adjacent to each other. Note that at the end of the factorization step, the first group of data contains the factor values for the rectangle.
Computational tasks

The basic definitions have been provided in Section 8.1.1 and Section 8.1.1. We provide additional details of the computations here, starting with the definition of the tasks into which the $DRD$ and $DR^TD$ tasks are divided.

We assume that the dependencies for the $DRD$ and $DR^TD$ tasks defined earlier have already been computed. We use the following new notation to describe the "extended" rectangular blocks. A rectangle enclosed within rows $m$ and $n$ and columns $p$ and $q$ is represented by $\overline{R}_{m,n,p,q}$; the original part is represented by $R_{m,n,p,q}$, the additional part for the forward solve is represented by $\overline{R}_{m,n\bar{p}\bar{q}}$ and the additional part for the back solve is represented by $\overline{R}_{m,\bar{m}\bar{p}\bar{q}}$.

For the forward solve, consider the task $<\text{Dia}_g,\text{Update},\text{Dia}_g,\text{Rect}>$, or the dependency $A,B \not\theta C$, where $A$ is diagonal $D_{a,b}$, $B$ is rectangle $R_{m,s,p,q}$ and $C$ is diagonal $D_{r,s}$. This is split into the following two dependencies and tasks, in which $[a,b] \cap [p,q] = [u,u'] \neq \emptyset$ and $[m,n] \cap [r,s] = [t,t'] \neq \emptyset$.

- **Diagonal updates rectangle, $<\text{Rect},\text{Update},\text{Dia}_g>$**
  
  Here, $A$ is diagonal $D_{a,b}$, $B$ is rectangle $\overline{R}_{m,n,p,q}$, and $A \not\theta B$. The corresponding computational task, $DR$, is the dot product of $D_{u,u'}$ and vector $R_{i,i,u,u'}$, for $t \leq i \leq t'$, accumulated in $R_{i,i\bar{p}\bar{q}}$.

- **Rectangle updates diagonal, $<\text{Dia}_g,\text{Update},\text{Rect}>$**
  
  The corresponding computational task, $RD$, involves subtracting vector $D_{t,t'}$, the vector $R_{i,i\bar{p}\bar{q}}$.

For the back solve, consider the task $<\text{Dia}_g,\text{Update},\text{Dia}_g,\text{Rect}^T>$ or the dependency $A,B \not\theta C$, where $A$ is diagonal $D_{r,s}$, $B$ is rectangle $R_{m,n,p,q}$, $C$ is diagonal $D_{a,b}$. This is split into the following two dependencies and tasks. In these tasks, $[r,s] \cap [m,n] = [t,t'] \neq \emptyset$ and $[p,q] \cap [a,b] = [u,u'] \neq \emptyset$.

- **Diagonal updates rectangle, $<\text{Rect}^T,\text{Update},\text{Dia}_g>$**
  
  Here, $A$ is diagonal $D_{r,s}$, $B$ is rectangle $\overline{R}_{m,n,p,q}$, and $A \not\theta B$. The corresponding computational task, $DR^T$, is the product of element $D_i$ with the vector $R_{i,i,u,u'}$. 


for $t \leq i \leq t'$, accumulated in $R^T_{m,m,i,u}$.

- **Rectangle updates diagonal, $<\text{Diag}, \text{Update}, \text{Rect}^T>$**

  The corresponding computational task, $R^T D$ involves subtracting from vector $D_{u,u'}$, the vector $R^T_{m,m,i,u'}$.

  Finally, note that all access to data in the triangle and rectangles is in row-major order in the computational tasks for the forward solve, and in column-major order in the computational tasks for the back solve. Since the back solve uses the transpose of the blocks used by the forward solve, no reorganization of data is required.

**BDAG**

We consider the precedence relationships of each type of block in turn, without going into the syntactic details of the construction of the BDAG. The construction can be done using the algorithm of Figure 3.3 in Section 3.2.3.

A triangular block or a column are not updated in either the forward solve or the back solve. A rectangular block in the forward solve may have several updates to it due to the $DR$ type tasks; in the back solve, a rectangular block may have several updates to it due to the $DR^T$ type tasks. These updates can occur in any relative order, so long as no two updates take place at the same time. A diagonal partition in the forward and back solves may have several updates to it. The only precedence constraint is that the task $TD$ is the last update in the forward solve, and the task $TTD$ is the last update in the back solve. The other updates may be performed in any relative order.

### 8.2 Sparse matrix-vector multiplication

The sparse matrix-vector multiplication, $y = Ax$, is an important component in a variety of engineering and scientific applications, and typically arises in iterative solution methods. One of the most recent algorithms for sparse matrix-vector multiplication is the one proposed by Agarwal, Gustavson and Zubair [1]. The main idea of their algorithm is to exploit any regular block structures or features in the sparse matrix $A$, which could lead to better exploitation of cache and vector processing or floating
point units, and therefore result in high performance. The algorithm is called Feature Based Extraction Algorithm, or FEBA for short. The FEBA consists of two phases: the preprocessing phase and the computation phase. In the preprocessing phase, the regular structures are extracted and stored. In the computation phase, these structures are used to compute the solution \( y \). The preprocessing cost is recovered if several iterations of the multiplication need to be performed with the same matrix structure.

FEBA is a sequential algorithm. In this section, we briefly describe how our methodology could be applied to parallelize FEBA-type algorithms for sparse matrix-vector multiply. Again, we describe the BCR for this algorithm. We assume that the mapping of blocks to processors is done separately. Given the BCR, and a mapping of blocks to processors, we can apply the scheduling and communication optimization algorithms of Chapter 6 to complete the parallelization. In the following, we sketch the different types of partitions of the matrix that are constructed by FEBA as described in [1], and relate them to the AVR for blocks.

There are four types of blocks in the partitioning of matrix \( A \). A generic block of each type is defined below, with a brief description of the data structures used. Each of the following types of blocks is obtained after an extraction step, and the blocks themselves are removed from the matrix and stored separately for processing during the computational phase.

1. A **dense rectangular block** of size \( m \times n \), i.e. a block with \( m \) rows and \( n \) columns. Such a block may have some zero elements in it. The density of a block, i.e. the percentage of nonzero elements in it, is controlled by two parameters whose values depend on the characteristics of the target machine. The block is identified by a pair of index sets \( \{ R, C \} \), where \( |R| = m \) and \( |C| = n \). \( R \) gives the row numbers, and \( C \) the column numbers of \( A \), from which the block has been extracted. The AVR for this type of block is as follows. The attributes are the index sets \( R \) and \( C \). The data values for the block are stored in one contiguous block in column-major order, to allow a saxpy-based implementation of the product of the block with \( \hat{x} \), where \( \hat{x} \) is the vector of elements \( x_i \in z \), \( i \in C \).
2. A **dense diagonal** of size $md$. Each diagonal is of density at least $\gamma$, i.e. the diagonal has at least $md \ast \gamma$ nonzero elements. The diagonal is identified by the row and column number of its first element. Note that if these numbers are $m$ and $n$, the elements of the diagonal are the entries $A_{m+i,n+i}$, $0 \leq i < md$. The attributes of this diagonal in its AVR is the pair $(m,n)$, and the data values are the elements of the diagonal stored in a contiguous area of memory.

3. A **sparse row** of size $nz$, i.e. a row with $nz$ non-zeros, where $nz > nzmin$. The row is stored in the compressed row storage or CSR format, in a manner analogous to the CSC storage of columns used for columns in sparse Cholesky factorization. The AVR for this storage format is described in Section 3.2.1.

4. A **submatrix** of size $ml \times n$ in a **ladder** scheme. The matrix $A$ is sorted by rows, in order of decreasing number of non-zero elements in each row. The rows of the matrix are then partitioned in such a way that each partition consists of $ml$ rows, where $ml$ is a tunable parameter. Each such $ml \times n$ submatrix is a block. The storage for the block is in the ITPACK format, which is summarized in [1]. Essentially, the format consists of two matrices: an $ml \times n'$ matrix $AC$ which holds the data values and a parallel $ml \times n'$ matrix $KA$ which holds column index numbers, where $n' \leq n$. The AVR for the block is straightforward: the attributes are in the $KA$ matrix, and the values are in the $AC$ matrix.

In the computational phase, each extracted block is multiplied with the appropriate portion of $x$ and the partial product is accumulated into the appropriate portion of $y$. These multiply-accumulates can be done in any order so long as no two of them are done at the same time. We will not go into the different ways of partitioning $x$ and $y$. The partitioning of these vectors depend, in general, on the matrix partitioning described above. As in the triangular solves, a simple fan-out method of performing the computations would be as follows. Each multiply-and-accumulate is treated as a single task, and is computed in the processor which owns the portion of $y$ updated in that task. The task may be performed any time after the blocks which participate in the multiplication have arrived at that processor. In a manner analogous to that described
in the triangular solves, we can estimate the worst case communication requirements for such a method, given a partitioning of the matrix $A$. An alternative fan-in method may be developed, in which each multiply-and-accumulate task can be divided into two tasks: a multiply and an accumulate. The multiply can be performed in the processor which owns either of the participating blocks, with additional storage created and attached to some of the blocks to hold the partial products, as for the rectangles in the triangular solves.

8.3 Conclusions

From our study of sparse Cholesky factorization, sparse triangular system solutions and sparse matrix-vector multiplication, it is evident that the partitioning of a problem depends strongly on the problem structure. In order to obtain a good partitioning that leads to high performance, especially for unstructured problems, the partitioner needs to be closely tailored to the problem. Our modular methodology is extremely useful in the parallelization of such problems, since it allows for partitioning to be done separately, and integrated into the rest of the methodology. This was demonstrated in our discussion of sparse matrix-vector multiplication. We also demonstrated the flexibility of the BCR in expressing block-based data-driven unstructured computations. The triangular systems that we covered in this chapter are those that arise after sparse Cholesky factorization. However, the techniques we applied to these triangular systems may also be used on systems arising from other factorization or decomposition methods for solving linear systems. Finally, our methodology may also be applied to dense problems in which one might want to experiment with non-uniform partitioning.
Chapter 9

Conclusions

Extracting high performance from unstructured sparse matrix computations on large-scale multiprocessors has met with little success in the past. The difficulty in parallelization is typified by sparse Cholesky factorization, a problem which has refused to yield good performance on large-scale multiprocessors for several years. The data-driven nature of these computations, arising from the irregularity of dependencies and communication patterns, inhibits the use of analytical methods for partitioning and scheduling. As a result, in order to design efficient partitioners and schedulers, experimental methods need to be employed. A thorough experimental study of problems in the class of unstructured sparse computations in order to understand the contribution of different algorithmic and architectural factors to performance requires a run-time parallelization system with low turnaround time, which may eventually be embedded in a production environment. The system must be composed of efficient and portable software tools that can be easily used by the applications programmer.

In this thesis, we proposed a run-time methodology for parallelizing unstructured sparse matrix computations, and a set of tools required to implement this methodology on scalable architectures. We applied this methodology to sparse Cholesky factorization and built a system called SHAPE, for Sparse Hybrid Automatic Parallelization Environment. The choice of a hybrid matrix partitioning scheme used in SHAPE evolved from our analysis of two-dimensional block partitioning for structured matrices. The analysis shows that the two-dimensional block partitioning scheme is more scalable than column partitioning with respect to communication. We then showed by experiments that the partitioning geometry is an important determinant of the utilization of the cache and vector processors or pipelined floating point units, and demonstrated
how the partitioning parameters could be chosen to maximize this utilization. The partitioner in SHAPE decomposes the symbolic Cholesky factor into a hybrid mixture of sparse columns, dense triangular blocks and dense rectangular blocks. The number and sizes of these blocks can be controlled by tunable parameters whose values are determined by the sparsity structure of the matrix, and the computation and communication cost parameters of the architecture. We showed that the single processor performance using this hybrid partitioning scheme is a significant improvement over column partitioning. We conclude that the partitioner must be tailored to match the structure of the problem in order to deliver high performance.

We proposed a general scheduler for unstructured data-driven computations. The main idea of the scheduling strategy is to create a static task and communication schedule and use this schedule repeatedly for the same problem structure but different data values. We also proposed a deadlock-free parallel communication optimization algorithm that uses the communication schedule generated by the scheduler to reduce the latency of message passing. We implemented the scheduler and an approximation of the communication optimization algorithm for block sparse Cholesky factorization in SHAPE, and demonstrated that the communication optimization results in significant savings in communication cost. To provide easy access to the scheduler and the optimized communication support software, we designed three primitives to serve as interface between the user program and these tools.

Using SHAPE, we conducted an extensive experimental study of block sparse Cholesky factorization on the iPSC/860. The experimental results show that with a judicious choice of partitioning parameters, our block-based partitioning and scheduling method outperforms a well-known column-based method in delivering high performance on a variety of structured and unstructured matrices. We sketched how our methodology could be applied to parallelize sparse triangular solutions and sparse matrix-vector multiplication. We expect that SHAPE will eventually provide the infrastructure needed to develop general principles of partitioning and scheduling for unstructured sparse computations.

Several areas of investigation can be pursued in the future. The modular design of
our methodology facilitates independent investigation of different parts of the methodology. One future area is the design and development of improved scheduling strategies to reduce the idle time on processors. Apart from this, experimentation with different allocation strategies for sparse Cholesky factorization can be conducted using SHAPE. The single-processor performance itself can be improved by refining the partitioning strategy. SHAPE can be expanded to include other problems like sparse QR factorization, sparse LU factorization, sparse triangular solves, and iterative methods for solving sparse linear systems. Experimentation with the constantly expanding SHAPE would lead to a better understanding of partitioning and scheduling for a host of unstructured sparse matrix computations.

An useful area of investigation is the design and development of tools to estimate the parallel performance of an algorithm given the cost parameters of the target architecture, without executing the algorithm. We made some headway into this problem by having SHAPE report the communication volume and the number of messages, and the load imbalance for a given partitioning and allocation. However, execution time estimation is a more difficult issue, as we observed in the chapter on experimental evaluation of block sparse Cholesky factorization. Further investigation needs to be done in this area, with applicability to a broader class of problems. Another useful set of tools would be graphical tools that depict the execution of the algorithm on the target machine. This would help enormously in detecting regions in time during execution when processors are idle, or where they consume high bandwidth in communication, so that experimentation on different partitioning and scheduling strategies can be conducted with low turnaround time. Finally, linking these partitioning and scheduling tools to compilers is an inevitable step in parallelizing large computational science applications on scalable architectures.
References


Vita

Seshadri Venugopal

1963 Born in Madras, India, on December 20.
1981 Graduated from Ramnarain Ruia Junior College, Bombay, India.
1985 B.Tech. in Computer Science and Engineering, Indian Institute of Technology, Bombay, India.
1988 M.S. in Computer Science, Rutgers, The State University of New Jersey, New Brunswick, New Jersey.
1993 Ph.D. in Computer Science, Rutgers, The State University of New Jersey, New Brunswick, New Jersey.
1986- Lecturer and Teaching Assistant, Dept. of Computer Science, Rutgers, The State University of New Jersey, New Brunswick, New Jersey.
1985-86 Systems Analyst Trainee, TCS, Bombay, India.
1988 Summer Intern, AT&T Communications, Piscataway, New Jersey.
1990 Student Engineer, Concurrent Computer Corporation, Tinton Falls, New Jersey.
1991 Graduate Fellow, Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, Virginia.


1993  S. Venugopal, V. K. Naik. An Environment for Partitioning and Run-time Scheduling of Parallel Block Sparse Cholesky Factorization. Submitted to *Parallel Computing*.