# SOFTWARE SUPPORT FOR PARALLEL PROCESSING OF IRREGULAR AND DYNAMIC COMPUTATIONS

BY JIA JIAO

**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 Apostolos Gerasoulis**

**and approved by**

_____

_____

_____

_____

**New Brunswick, New Jersey**

**October, 1996**

## ABSTRACT OF THE DISSERTATION

# Software Support for Parallel Processing of Irregular and Dynamic Computations

by Jia Jiao

Dissertation Director: Professor Apostolos Gerasoulis

Many real world scientific computations are irregular and dynamic, which pose great challenge to the effort of parallelization. In this thesis we study the efficient mapping of a subclass of these problems, namely the "stepwise slowly changing" problems, onto distributed memory multiprocessors using the task graph scheduling approach. There exists a large class of applications which belong to this category. Intuitively, the irregularity requires sophisticated mapping algorithms, and the "slowness" in the changes of the computational structures between steps allows the scheduling cost to be amortized, justifying the approach.

We study three representative and widely-used applications: The N-body simulation in astrophysics, the Vortex-Sheet Roll-Up and the Contour Dynamics Computation from Computational Fluid Dynamics. We start with an initial global compile-time scheduling, and apply new rescheduling algorithms to improve performance when this schedule degenerates over the iterative process. We develop rescheduling algorithms for two important dynamic patterns: task graph weight variation, and dynamic spawning of new subgraphs. These algorithms are tested on random graphs and real applications such as the FMM N-body and Vortex Sheet. Our experiments show that global scheduling using sophisticated methods can be beneficial for these problems, and our

fast rescheduling algorithms can correct run-time imbalance with very low cost.

In summary, we discuss several central issues such as schedule reuse, performance/overhead trade-off and the selection of rescheduling methods. We identify classes of problems where rescheduling algorithms are applicable, and present experimental evidence to justify our approach.

Throughout the thesis, performance results are obtained from particular problems but presented in the general framework of software support systems. In addition, we examine an automatic task graph generation tool that can handle restricted cases of sequential code, and carry out its integration with our scheduling system. The new system is capable of realizing automatic parallelization of simple programs, a step forward to the grand challenge of fully automatic parallelization of regular and irregular code.

# Acknowledgements

First I would like to express my appreciation of all the guidance from Professor Apostolos Gerasoulis, my thesis advisor. I was introduced to the subject of parallel programming in his class 4 years ago and have been working under him for almost 3 years. His advice has been crucial to my research work and also my personal life. Without him I could never have achieved any success. I am very grateful to his efforts.

I would also like to thank all the other members in my Ph.D thesis committee: Professors Uli Kremer, Gerry Richter and Joel Saltz for their time and kindness.

During my research I receive much assistance from a number of former and current students at Rutgers. In particular, Professor Tao Yang, a former student of Professor Gerasoulis, gave me his full support, especially during the initial stage of my research. I should also mention that Ramkrishna Chatterjee offered his help in the PlusPyr/PYRROS integration project, by working on the coding of the parser for the task graph language. On the other hand, during my five and half years in the department, Ms. Valentine Rolfe, the graduate secretary, has been a good caretaker when students need help. I appreciate all the things she has done for me.

Several people from the Mechanical Engineering Department have offered me tremendous help in understanding the two Computational Fluid Dynamics applications studied in this thesis. Dr. Victor Fernandez helped me with the Vortex Sheet Roll-up problem. Dr. Dristchel and Hong-bing Yao provided me with their papers on the Contour Dynamics problem, together with a sequential Fortran program for the fast method and an input distribution generator.

The work described in this thesis was supported by ARPA contract DABT-63-93-C-0064 under "Hypercomputing and Design" project, and by the Office of Naval research under grant N00014-93-1-0944. The CAIP center at Rutgers provided access to their

Ncube2s parallel machine, where the experiments in the thesis were conducted.

# Dedication

This document is dedicated to my parents.

# Table of Contents

# List of Tables

# List of Figures

# List of Abbreviations

| | |
|---|---|
| **CFD** | is for Computational Fluid Dynamics |
| **FMM** | is for Fast Multipole Method |
| **PT** | is for Parallel Time |
| **GJ** | is for Gauss-Jordan algorithm |
| **LU** | is for LU decomposition |

# Chapter 1

# Introduction

## 1.1 Overview

Parallel architectures are becoming increasingly more powerful. Not only do the peak performances of massively parallel platforms continue to climb rapidly, even traditional workstations are employing multi-processor architectures to deliver computing power at a reasonable cost.

Despite all the advancement in the hardware and architecture front, currently parallel programming software lags far behind. Most of the successes, until now, are for problems where parallelism is relatively easy to explore, or "regular problems". These problems possess nice structures which enable them to be partitioned onto multiprocessors so that there is little communication between the components, and the load in each processor is almost balanced. As for problems which still contain high potential of parallelism but with more complex underlying dependence structures, the computing power of parallel machines are vastly underutilized. There are many reasons behind the immature status of parallel software, but two of the major ones are :

- Difficulties in partitioning the problem onto multiprocessors to produce efficient code.

  There are many ways to model parallel computation, such as data parallelism and functional parallelism. However, no matter which model is adopted, in the presence of communication cost the optimal distribution of the computation, or scheduling, for parallel architecture is extremely hard. In fact, many of the problems involved are NP-complete. Facing such obstacles, researchers are generally pursuing the following two approaches, or a combination of the two:

1.) Developing automatic tools to produce good, although not necessarily optimal, schedule and code, or

2.) Attacking problems individually with manually written parallel programs.

"Combination" could be that the user utilizes certain suggestions from the automatic tool to improve the efficiency of the parallel code. Hand-written codes can deliver good performance for certain regular problems, such as dense linear algebra. Such codes generally rely on certain principles such as the "owners compute rule", and use simple mapping methods such as block or wrap mapping. However, for most problems, especially those with irregular structures, manual coding is tedious and requires sophisticated knowledge of message passing mechanisms that a general user may not possess.

Automatic scheduling is a promising approach, and is the one followed by this thesis. There are two basic underlying models for parallel computations: *Data Parallelism* and *Functional Parallelism.*

In the data parallelism model, data elements, mainly arrays, are distributed across processors. This data mapping determines the work mapping the data access pattern. Research efforts concentrate on finding good data partitioning in order to reduce the parallel time. Many compilers are built to handle regular problems, such as PARADIGM [4], Fortran D [1], and SUIF [2]. For irregular code, the CHAOS [34] system uses inspector-executor approach to produce code for input-dependent problems.

In the functional parallelism model, the basic units of the program are functional components. The PYRROS scheduling tool developed here at Rutgers is based upon the Dataflow Task Graph model, a kind of functional parallelism. Here the components are called tasks. Each task reads certain data, carries out its operations, and sends some results to other tasks. In this model the data is not partitioned among processors but rather dynamically "pumped" to the tasks that need them. This is called "macro-dataflow". The mapping of programs in this model amounts to the assignment of tasks to processors and the ordering of tasks

in each processor. A good mapping, or schedule, should minimize the parallel time. The tasks, together with the dataflow dependences, forms a Dataflow Task Graph. We shall discuss this model and the concept of scheduling in more details in chapter 2.

Previous work has demonstrated that PYRROS can achieve good performance on both random graphs and many matrix algebra problems [59]. The input to PYRROS is a user-supplied task graph, which may be obtained by a task graph generator designed for the specific application problem, or from some automatic parser of serial code with task annotations (currently for code with affine loops only.) The output from PYRROS is parallel code executable on a given architecture such as the Ncube2. Details about PYRROS system organization can be found in chapter 2.

In this thesis, PYRROS will be employed in the parallelization of some scientific computing problems that are more complex and irregular, where simple methods such as block or wrap mapping will not yield good performance. Moreover, closely related to the input-dependent property of irregular problems, for many applications that run iteratively, the task graph may exhibit certain changes if the input data is modified. Such dynamic problems add one more dimension of complexity to the parallelization issue, and the combination of the two aspects is the central issue being studied in this thesis. We are able to derive results from several representative irregular/dynamics problems that have implication on this class of applications in general.

- Difficulties in parallelism detection.

  Even if an all-powerful scheduling and code generation tool is available, the widespread use of parallel platform is still hindered by the lack of parallelism detection support. Existing systems, such as PYRROS, rely on user-supplied task graph and task definition code (or node program). This will pose serious obstacle for any user who is not specialized in parallel programming, and limit the usage of parallel machines.

Automatic parallelism detection has been researched in the literature, mainly based upon dependence analysis of serial code, see for example [43]. Much progress has been made to detect parallelism in affine loops. But again the general problem of determining precise dependences is intractable. Moreover, the problems of inter-procedural analysis and analysis of irregular loops with input dependent bounds, both of which are important for parallelism detection for general industrial strength serial code, are not well understood due to the extreme complexity.

If the task graph model is adopted, we need even more than dependence analysis to derive the dataflow pattern. Traditional dependence analysis methods focus on the statement level, whereas in the task graph model, edge dependences are coarser grain. Although there are works using other notions of data dependence analysis, such as Data Access Descriptors [3] and Regular Section Descriptor [32], few systems are available for task level dependence generation.

PlusPyr, [15], an automatic task graph generation tool developed in France, with the aim of providing a front-end user interface for PYRROS, is the first such attempt, and for dense linear algebra problems it works well. PlusPyr incorporates the Omega test [43] as its dependence analysis engine, and creates coarse grain communication rules between tasks by merging fine grain dependences. In this thesis, we will also address issues in building a unified system that attempts to fully automate the whole process of program parallelization, from sequential code with task annotations to executable parallel code.

Although currently we can only achieve this goal for regular problems with affine loops, this research is still related to the major portion of our work on irregular problems. Since without a prototype system that can at least demonstrate the feasibility of automatic parallelization without user intervention on some regular problems, which can be viewed as special cases, it's impossible to imagine such a system for general problems. We believe that the development of such prototypes is the first step toward meeting the grand challenge of building automatic, efficient parallel compilation systems. And it can be extended to handle more complicated

serial code when the dependence analysis technique improves.

This aspect is addressed in chapter 8 of the thesis, where an automatic parallelization tool for regular code formed by integrating PYRROS and PlusPyr, the PYRROS+ system, is discussed.

It should be noted that although for irregular problems automatic parallelism detection is beyond our capability, we can still derive task graph generators for any specific application, with the knowledge of the algorithm used and the initial input. In this way we can still make much progress on the automatic scheduling issue. Chapters 3 through 7 applies scheduling and rescheduling methods to irregular problems with such individually generated task graphs. We analyze the algorithm and data structure used in the applications, then design task graph generators. The development of these generator should be considered as part of the thesis contribution.

## 1.2 Assumption and Scope

We first sketch the basic framework of our scheduling/rescheduling paradigm in figure 1.1.

The major portion of the contribution of this thesis is on irregular and dynamic problems. For these problems the task graph is produced by specially-designed generators which build task graph from the input data, often as a by-product of the input data distribution process. For instance, for the FMM N-body algorithm discussed in chapter 3, the space containing particles must be subdivided into a hierarchy of boxes in order to build the data structures needed by the algorithm, serial and parallel versions alike. During this process we can obtain a task graph based on the partitioning. The same thing holds for some sparse matrix applications. Of course a generator for the FMM N-body algorithm will not work for sparse matrices, so each generator must incorporate knowledge about the functional components of the irregular application. This is the best we can do, given that automatic parallelism detection is extremely hard for such codes.

The task graph will be scheduled by PYRROS for execution on parallel machines. If

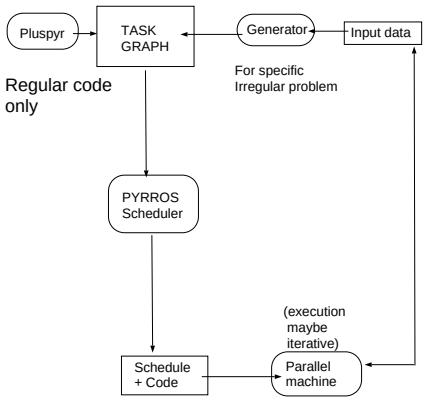



Figure 1.1: The basic framework of our research system

the execution is iterative, the schedule is used for the initial iteration and could possibly be reused. Furthermore, for a dynamic problem where the task graph changes significantly after a certain number of iterations so that the initial schedule performance has degenerated, the task graph can be incrementally updated and we can apply our incremental scheduling methods to re-map the tasks. These new results will be presented in chapter 3 through chapter 7, which form the main portion of this thesis.

On the other hand, if the code is regular, i.e. affine loops, PlusPyr may be used to automatically produce the task graph without requiring assistance from the user. The input to PlusPyr is a piece of serial code contains task annotations, which are similar to the data parallel annotations used in languages such as HPF ([1]). For these problems it's possible to automate the whole parallelization process, so that the integrated system can be viewed as a compiler. We discuss our work on the integration of two existing systems to build PYRROS+ in chapter 8.

### 1.2.1 The static scheduling and code generation framework

The difficulties and challenges discussed above are the issues we will try to address in this thesis. The main theme of the thesis is related to the first challenge. In particular, our work is based on previous effort to tackle the task graph scheduling problem. We continue to use the following basic framework of PYRROS:

- Scheduling is based on static, initial task graph information, instead of dynamic methods which does not rely on such preprocessing.

  The purpose of static scheduling is to exploit parallelism by utilizing more information in the task graph structure. The success of PYRROS has been demonstrated on many regular problems in dense linear algebra ([59]). We will adhere to the basic scheduling algorithms in PYRROS, and move ahead to more complex applications.

  This thesis explores the domain of irregular/dynamic problems. The irregularity demands a good scheduling system since dynamic or naive scheduling will not extract enough parallelism. On the other hand, the dynamic nature poses problems for a static schedule. In our later in-depth discussion, it turns out that the problems suitable for our method are the class of problems where the dynamic change is gradual, or "slowly changing problems", and two low complexity rescheduling methods are developed to handle long-term changes due to the dynamic nature of the computation.

- The code generated is of SPMD nature. Past research has established the correctness of the code and carried out various optimizations. We will continue to use this component of PYRROS.

  For irregular problems, additional information and data structures need to be generated along with the code. This is due to the underlying input-dependent property of irregular problems: with different input, not only will there be different task graphs, but the data access by a generically defined task can be different. For instance, in the adaptive FMM N-body application, a task corresponding to

a box located at a specific position $p$ in space at level $l$ of the subdivision can be defined as $T(l,p)$, but the data accessed by this task varies, depending on the partitioning of other regions in space, which is based on the initial input distribution. This is because the boxes interacting with $T(l,p)$ will be different. Such information is not available until the initial input is presented. Details can be found in chapter 3. On the contrary, for regular problems, a task always access the same data. For instance, a task in the Gauss-Jordan task graph may be defined as $T(k,j)$, which updates column $j$ with column $k$ of the matrix, and it carries out the same operation for all matrices of size $n$. This issue will be clear when we study the N-body problem in chapter 3.

By feeding input-dependent information to the code at execution time in order to carry out correct functions, our approach is essentially the same in spirit, but not in form, to the inspector-executor paradigm proposed in the CHAOS system ([34]). CHAOS, unlike PYRROS, is based on data parallelism model and tries to partition the data in order to reduce parallel time. In this aspect it is not directly comparable to our work. However, CHAOS is also attempting to tackle irregular serial codes, which exhibit input-dependent loop bounds. The inspector-executor approach is essentially the following: For a given problem, a generic code is produced by the inspector component, but the execution is based on specific data related to the input configuration by the executor component, which is the same idea that we use to guide the execution. This should not come as a surprise, since it is a very intuitive way to handle the irregularities in the application.

- Scheduling is done at the sequential front-end of the parallel machine, which serves as the host. The host/node configuration is very common since the host can serve as a user interface running ordinary operating systems such as UNIX, and also handle the preprocessing and post-processing, which often involve accessing files, while leaving the computationally intensive part to the parallel nodes. PYRROS was built with such a configuration in mind and the host carries out scheduling and cost generation before loading the parallel code to the nodes. Therefore the

scheduling is centralized.

There exist other scheduling policies that are distributed, i.e. each node has its own scheduler. Such systems are meant mainly for dynamic load balancing. Since PYRROS uses relatively sophisticated algorithms that are inherently sequential, we believe that a centralized scheduler is necessary.

It should also be mentioned that in order to be consistent, our new rescheduling component, which carries out schedule adjustment, is also running at the central host. It collects information about parallel time per iteration, does rescheduling when needed by the user, in order to improve performance for later iterations. It does not attempt any adjustment in the middle of an iteration. However, the host process can collect results from the nodes when needed and continue running even if the node programs halt temporarily, and redistribute the data when the nodes are reloaded by the host.

### 1.2.2 Investigation of various computational problems: Computer Science vs. Physics

Since the bulk of this thesis is devoted to the investigation of irregular/dynamic problems, we choose some representative real-world scientific computing applications, such as the classic N-body simulation in Astrophysics and the Vortex Sheet Roll-Up problem in Computational Fluid Dynamics. Here we have to clarify the scope of our research on such problems.

Computational simulations are used by scientists and engineers to model complex physical systems that are intractable by theoretical analysis alone. The interests of these computational scientists, therefore, are mainly to achieve deeper understanding of the phenomena by simulating larger examples or for more time steps. On the other hand, when these problems are researched in this thesis, they are put in the perspective of Computer Science. Here our concern is how to build a system that will provide support for the execution of such problems, including the following questions, among many others:

- How can we exploit more parallelism, given a problem and the input data?
- How can we handle dynamic changes in the input, what are the methods to choose from?
- How do different patterns in the input affect our choices of scheduling and rescheduling methods?
- How to lower the cost of mapping and re-mapping, and use limited resources?

On the other hand, we make no attempt to analyze the physical implications of the simulation results, nor do we try to predict the execution outcome. This thesis will incorporate just enough physics knowledge to enable a general understanding of the application and to permit the generation of task graphs, scheduling, and coding.

With our emphasis on the software support methods rather than the numerical part of the problem, we will focus on the complexity of the task graph structure instead of mathematical formulae. For example, in the case of the FMM algorithm for the N-body simulation ([28]), we choose the 2-dimensional adaptive version, since it has a irregular/dynamic task graph structure that poses problem for parallelization, while the non-adaptive version is relatively easy. On the other hand, we did not attempt to parallelize the 3-dimensional algorithm, since it has a similar task graph structure to the 2-D case but with very arcane sphere harmonics computation inside each of the tasks, which means most of the work there will be spent on sequential coding of these mathematical operations, instead of parallelization issues. But we strongly believe our system should work well on the 3-D version since the task graphs for both versions are essentially the same while the 3-D version is more computationally intensive inside each task, which should yield even better speedup results. Of course, a good implementation of the 3-D version is a research problem in its own right, and is of more interest to physicists, so it is still a future research direction, although it is out of the scope of this thesis. Here we focus more on parallelization issues instead.

### 1.2.3 Integration of PYRROS/PlusPyr systems

This is a different research subject concerning automatic generation of task graphs, and the first step toward a fully automatic parallelizing compiler. In this part of the thesis we make the following assumptions:

- We will make use of existing dependence analysis algorithms, instead of developing new algorithms. The current algorithm used in PlusPyr is the Omega test([43]). It can be replaced by other methods, but such a research direction is beyond the scope of this thesis.
- We will only apply the integrated system, PYRROS+, to those problems that can be successfully analyzed by the PlusPyr front-end. Given the current status of research effort in dependence analysis and task graph generation, this obviously will limit our test cases to affine loops.

## 1.3 Thesis Contributions

The contributions of the thesis are many-fold. The major portion, chapter 3 through 7, is devoted to scheduling and rescheduling of irregular/dynamic problems. But there are other works: In chapter 2 a new algorithm is introduced for a component of the PYRROS scheduler, and in chapter 8 we discuss the integration of PlusPyr and PYRROS systems.

- *Physical Mapping*

  We have refined the physical mapping algorithm used in PYRROS. Physical mapping is one of the scheduling stages within PYRROS. In the original version of PYRROS, the heuristic used did not perform well. Although it was proposed that Bokhari's algorithm ([8]) could be used, it turns out that the complexity of that algorithm is prohibitively high for our purpose. Our new heuristic is inspired by Bokhari's algorithm and based on similar idea, but its complexity is acceptable for incorporation into PYRROS, and its performance is competitive.

- *Rescheduling heuristics and system*

  Since we are dealing with dynamic problems where task graph structure can change during iterative execution, we need to provide rescheduling support methods that are fast and yield competitive performance.

  We proposed two of such heuristics, dealing with different types of dynamic changes in the task graph. They both have very low complexity because they are localized and incremental algorithms, and tests on random graphs and application to our representative dynamic problems demonstrated their success. The algorithms are implemented as a new rescheduling component that is independent from the rest of PYRROS, which may be invoked when such a need arises, such as when the performance exhibits degradation and the user decides to do re-mapping.

  We believe that this research is the first investigation of dynamic task graph rescheduling problem. Existing research papers on dynamic problems are essential all within the load balancing framework, without the task graph notion and dependence, and all previous research on task graphs are concerned with initial scheduling.

- *Parallelization of the FMM N-body problem and its implication on scheduling systems*

  The N-body problem, an important irregular/dynamic problem being studied intensively, is one of the applications to be investigated. Parallel N-body simulation is an active area of research in its own right, and is the topic of a number of Ph.D dissertations, such as [46], [48].

  We are the first to study this parallelization problem from the perspective of task graph scheduling. We developed a task graph generator for the 2-dimensional adaptive FMM algorithm, implemented the algorithm on the Ncube2s machine, and achieved good performance. Generation of task graph for this problem requires in-depth knowledge of both the FMM algorithm and the task graph model of parallel computing, and is considered to be a contribution by itself.

We also studied various issues in a general framework, such as the performance vs. scheduling overhead trade-off, and whether scheduling is beneficial. Moreover, the problem is used as one of the testbed for the rescheduling heuristics. We monitored the dynamic behavior of typical input cases, and tested the applicability and performance of several approaches. On a higher level, we established a framework for applying our scheduling/rescheduling paradigm to iterative executions.

As far as we know, these works are very significant since no one has investigated parallelization issues in this model and framework. Our work not only provided a new approach for the N-body simulation, but also gave strong evidence of the potential of our system.

- *Investigation of two CFD problems and the related system support issues*

  We also investigated two other irregular/dynamic problems: The Vortex Sheet Roll-Up Simulation and Planar Contour Dynamics, both of which variations of the classical N-body problem. These applications are within the domain of Computational Fluid Dynamics (CFD).

  The vortex sheet problem is a very ideal testbed for our rescheduling system because of the well-understood physical behavior of the test data. Our research yielded considerable in-depth understanding of the circumstances where rescheduling methods are applicable.

  The contour dynamic problem, on the other hand, has a relatively simply task graph but an elusive dynamic alteration pattern that demands entirely new approaches. We have the first parallel implementation of this problem, and we raised some open problems for further investigation. The research work on the contour dynamics problem indicates that the nature of dynamic problems are diverse. Although the results here are more preliminary, a foundation has been laid for this kind of applications.

- *Automatic parallelization process for regular code*

We have achieved initial success in the long term goal of building automatic parallelizing compilers, by integrating PlusPyr and PYRROS tools. The new PYRROS+ system built is capable of carrying out the whole process of compilation, starting from sequential code with task annotations and ending at executable parallel code, for regular code that can be handled by current dependence analysis techniques. The uniqueness of this system is that it uses functional parallelism approach.

## 1.4 Thesis Organization

This thesis is organized as follows: In chapter 2 we give the fundamentals of task graph model, classifications of task graphs, and the concept of scheduling. We also discuss the PYRROS system and a new physical mapping algorithm. Then we present the task graph structure, scheduling and performance results for the FMM N-body algorithm in chapter 3. The next chapter, i.e. chapter 4, is devoted to the heuristics for schedule readjustment drawn from the need to handle dynamic problems. In chapter 5 we apply the dynamic reschedule system to the FMM algorithm. Then in chapter 6 we investigate rescheduling issues using the Vortex Sheet Roll-Up problem as a testbed. In the next chapter, chapter 7, we discuss parallelization of the Planar Contour Dynamics problem. Finally, in chapter 8 we present our approach in the integration of PlusPyr and PYRROS systems, and we draw conclusion and discuss future directions in chapter 9.

# Chapter 2

# Background and a New Physical Mapping Algorithm

## 2.1 Program Parallelization and the Task Graph Model

This thesis focuses on the central theme of mapping computational problems onto distributed memory multiprocessor architecture, where each processor has its own memory space and uses message passing mechanism for inter-processor communication. Such machines, although potentially very scalable, are more difficult to program than shared-memory machines, where a global memory space is used for all processors.

There are many models proposed for program parallelization. For example, in the data parallelism model, array elements are partitioned among the processors, and the partitioning determines work mapping. Here is a simple example. Consider the loop:

```
For i = 1 to 100
  a(i)=i;
endfor
```

A data parallel implementation on 10 processors may allocate array elements $a[p * 10 + 1]$ to $a[(p+1) * 10]$ to processor $p$ $(p = 0..9)$. This is called a Block mapping of array $a$. After the data mapping, each processor works on its own data set. For a general overview of data parallel programming environment, see [1].

Another basic paradigm in parallel computing is functional parallelism, where functional components, such as loop iterations or task, are the units of partitioning. A functional parallel implementation of the simple loop above may be assigning to each processor 10 iterations of the loop. Here we adhere to the Dataflow Task Graph, a kind of functional parallelism model. A *task* is a basic unit of computation consisting

of statements and/or procedures, in the sense that this group of computation should always be executed sequentially. After the definition of tasks, the inter-task data dependences form *edges*. Tasks, as nodes, and edges together constitute a graph that is weighted, directed and acyclic(DAG). In the task graph, the weight of a node is the time to execute the task on one processor, and the weight of an edge $(n_i, n_j)$ is the shortest time needed for transmitting data if $n_i$ and $n_j$ are in separate processors. If the two end tasks are in the same processor there is no cost associated with the edge. It's clear that these weights are machine-dependent. Exact weight estimation is almost impossible, but in practice, approximation is good enough for our purpose. The execution of the DAG follows the *macro data-flow*: we assume that each task receives all the incoming data from predecessors in parallel, then starts execution to completion and upon finishing, sends data to all the successor tasks, again in parallel. This model, although idealizes the send and receive operations, is widely accepted in the works of many other researchers such as Sarkar [47].

It also should be noted that the execution is asynchronous and the send operations are non-blocking, meaning that the sending processor continues its execution immediately after the send operation without waiting for an acknowledgment of receipt. This model is free of communication deadlock.

The *granularity* of a task graph $G = (V, E)$, $g(G)$, describes the ratio of computation to communication in the task graph. Intuitively, task graphs that are computationally dominant permit greater degree of parallelism. We define the grain of a task $T_x \in V$ as

$$g_x = min(\frac{\min_{T_k \in PRED(T_x)} \tau_k}{\max_{T_k \in PRED(T_x)} c_{k,x}}, \frac{\min_{T_k \in SUCC(T_x)} \tau_k}{\max_{T_k \in SUCC(T_x)} c_{x,k}})$$

where $\tau_k$ is the weight of the $k$th predecessor (or successor), i.e. $T_k$ of $T_x$, and $c_{k,x}$ is the weight of the communication edge from $T_k$ to $T_x$.

And the *granularity* of a DAG $G$ as

$$g(G) = \min_{x=1:v} \{g_x\}.$$

We call a graph "coarse grain" if $g(G) \geq 1$, and "fine grain" otherwise. We sometime call a graph "mixed grain" if its granularity is around 1.

Given a task graph $G(V, E)$ and $P$ identical processors, a *schedule* consists of the following:

1.) A mapping of each $t \in V$ to a processor $p_i$, $0 \leq i \leq P-1$, and

2.) An ordering of all tasks mapped in each processor $p_i$, $i = 0...P-1$.

A schedule has to be *valid* in the sense that all precedence constrains imposed by the macro-data flow dependence are satisfied. The length of the schedule is the *parallel time*, or *makespan* .

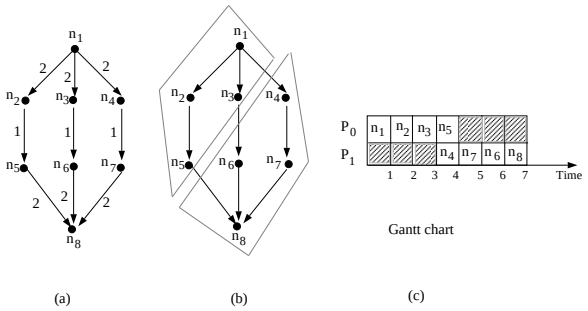



Figure 2.1: (a) A DAG with node weights equal to 1. (b) A processor assignment of nodes. (c) The Gantt chart of a schedule.

Fig. 2.1(a) shows a weighted DAG with all computation weights assumed to be equal to 1. (For this graph only, weights can be arbitrary for general graphs.) Fig. 2.1(b) shows a processor assignment using 2 processors. Fig. 2.1(c) shows a *Gantt chart* of a possible schedule for this DAG. Gantt chart can be used to illustrate both the mapping of tasks to processors, as well as the ordering of tasks in each processor. Note that this is not the only schedule, and it is non-optimal.

A good scheduling algorithm should try to minimize the parallel time. In general such minimization problems are intractable, and many heuristics are proposed for good, albeit non-optimal results. See, for example, work done by Sarkar, Hwang et al. [47, 33], and our own heuristic.

## 2.2 The Multistage Approach and a Sketch of Each Stage

PYRROS is an automatic task graph scheduling tool developed at Rutgers. The tool has undergone extensive evolution since its inception. At present time, it is capable of

taking user defined task graphs and manually specified task definition code, i.e, the code fragments containing the portion of sequential operations within tasks, and producing executable parallel code. The PlusPyr tool [15] was aimed at providing an automatic task graph generation front-end, and our recent work involved PlusPyr/PYRROS integration, but we will postpone the discussion to chapter 8 of the thesis. For now we focus on PYRROS itself.

Briefly, let $v = |V|$, $e = |E|$ for graph G=(V,E) and assuming $p$ processors are available, the following multistage approach to scheduling is used in PYRROS:

1. **Clustering:** Assign tasks to a set of clusters. Tasks in the same cluster will be executed in the same processor. At this stage it is assumed that unbounded number of processors are available, and the stage returns $u$ clusters. The purpose of this stage is to identify tasks that need to be sequentialized regardless of the number of processors.

2. **Cluster Merging:** Merge the $u$ clusters into $p$ completely connected virtual processors if $u > p$. At this stage the fact that only $p$ processors are available is taken into account.

   Some other algorithms, such as ETF ([33]), schedules the graph directly on $p$ processors. They are called one-stage method. It's not clear which class of methods is better. For example, ETF often gives shorter schedule on coarse-grain graphs but the running time is much higher than PYRROS, making the overhead excessively high for real applications. See [21].

3. **Physical Mapping:** Map the $p$ virtual processors to $p$ physical processors. The reason is that virtual processors are assumed to be completely connected while the real architecture may be not. For example, the Ncube machine where our experiments are conducted has a hypercube architecture.

4. **Task Ordering:** Order the execution of tasks within each processor.

The overall complexity of this method is $O((v+e)\log v + p^3)$. We sketch the algorithms employed for each stage below.

**Clustering**

PYRROS uses the Dominant Sequence Algorithm (DSC) to automatically determine the clustering for task graphs, see [61] for a detailed description. It has been shown to be superior to several other clustering algorithms in the literature in the following sense:

- DSC has an "almost linear" complexity, $O((v+e)\log v))$ time complexity and $O(v+e)$ space complexity. Most of the algorithms in the literature have complexity higher than $O(v^2)$ which makes them impractical for large task graphs.
- Experimental results using random graphs and numerical computation DAGs show that DSC is comparable or even better than several higher complexity algorithms from the literature [36, 47, 58] in terms of the makespan of the schedule produced.

**Cluster Merging**

PYRROS uses a variation of *work profiling method* suggested by George et. al.[19] for cluster merging. This method is simple and has been shown to work well in practice, e.g. makespan [45], Ortega [40], Gerasoulis and Nelken [22]. The complexity of this algorithm is $O(u\log u+v)$, which is less than $O(v\log v)$. It is as follows:

1. Compute the arithmetic load $LM_j$ for each cluster.
2. Sort the clusters in an increasing order of their loads.
3. Use a load balancing algorithm so that each processor has approximately the same load.

**Physical Mapping**

At this stage, we have $p$ virtual processors (or clusters) and $p$ physical processors. Since physical processors are not completely connected, we have to take the processor distance into account. Let $TC_{i,j}$ be the total communication which is the summation of costs of all edges between virtual processor $i$ and $j$. And Let $CC=\{TC_{i,j}|TC_{i,j}\neq 0\}$ and $m=|CC|$. In general we expect that $m << e$, because $m$ depends on the number

of the links between the $p$ processors and it is much less than the number of edges in the task graph $G$ if $G$ is not extremely small.

The goal of the physical mapping is to determine the physical processor number $P(V_i)$ for each virtual processor $V_i$ that minimizes the following cost function F(CC,P):

$$F(CC, P) = \sum_{TC_{i,j} \in CC} distance(P(V_i), P(V_j)) \times TC_{i,j}.$$

The reason for using this cost function is that virtual clusters that communicate intensively will be mapped to processors that are close to each other. While this model may overestimate the overall communication volume in a wormhole-routing network, it does capture the fact that the overall communication overhead increases if the distance between the source and destination increases. Even in a wormhole-routing network, the longer the distance between processors the higher the chances for communication contention, so physical mapping still captures the need of communication volume reduction.

The original version of PYRROS lacked a good physical mapping algorithm. The algorithm of Bokhari ([8]) has very high complexity and can not be used without modification. Our contribution here is the development of an algorithm that produces good mapping at an acceptable cost. It is based on Bokhari's algorithm but with modifications to control the running time. We will discuss the algorithm in section 2.3.

**Task Ordering**

After the physical mapping has been fixed, the interprocessor communication delay between tasks can be determined. We still need to order the execution of tasks within each processor such that the total parallel time is minimized.

PYRROS uses the RCP (ready critical path) algorithm with complexity $O(v \log v + e)$. One major difference between RCP and the classical critical path algorithm is that it maintains a *ready* task list instead of a *free* task list. (A free task is a task such that all its predecessors have finished execution, while a ready task is a task such that all the data needed have arrived. A task becomes free first but it may still be waiting for some data to arrive from predecessors.) A detailed analysis of the task ordering problem is given in [62]. It provided some evidence that using ready list has some advantages over

using free list.

**Code Generation**

All of the above stages are components of the scheduler. PYRROS contains one other major component which is the code generator. After the scheduler produces a symbolic schedule, the code generator examines the message passing patterns based on the schedule and task ordering in each processor, and carries out communication and memory optimizations, which are aimed at removing redundant communication, saving space, and reducing idle communication time. In particular, the current system tries to optimize broadcasting and multicasting for the hypercube architecture. One of the future research direction is to extend communication optimization to different architectures such as mesh, which is adopted by the more powerful Cray T3D machine.

After communication optimization, the code generator produces parallel C code by inserting communication primitives, depending on the machine type. Currently the system supports Ncube and Intel code. It should be extended to produce PVM ([50]) and MPI ([52]) message passing primitives, since these are widely used in machines such as IBM SP2 and Cray T3D, as well as network of workstations. This extension should be carried out simultaneously with the above-mentioned communication optimization for architectures other than the hypercube.

Next we focus on the physical mapping part of the system, where improvement can be made.

## 2.3 A Physical Mapping Algorithm Developed for PYRROS

The physical mapping problem is a special case of the Quadratic Assignment Problem [39]. Determining an optimal physical mapping is difficult and we modify the Bokhari's heuristic algorithm [8] to be used for PYRROS.

The Bokhari's algorithm starts from an initial assignment, then performs a series of pairwise interchanges so that the $F(CC, P)$ decreases monotonically. We do not use the original Bokhari's algorithm due to its high complexity which can be exponential in the worst case.

The key observation is that the reduction of $F(CC)$ is large initially but begins to dwindle during the later phases. Thus, it is not necessary to spend vast amount of time in order to achieve very limited additional reduction after the first few stages, unless the minimization of $F(CC)$ is the only goal and the algorithm running time is not a concern. This is true for certain combinatorial optimization problems that motivated Bokhari's original algorithm, but obviously the cost is not affordable for a scheduling system.

Thus the basic idea of our approach is to perform a smaller number of steps and get a significant amount of reduction as fast as possible. Here is a description of our algorithm:

Set the mapping $M$ to be an arbitrary initial mapping.
**For** $i = 1$ to $p$ DO
  $gain = 0; maxnode = -1;$
    **For** $j = 1$ to $p$, $i \neq j$ DO
      Let $M_2$ be the mapping obtained by exchanging $map[i]$ with $map[j]$;
      $improve = F(CC, M) - F(CC, M_2)$
      If $(improve > gain)$ $gain = improve;$ $maxnode = j;$
    **Endfor**
    Modify the current mapping $M$ by exchanging $map[i]$ with $map[maxnode]$;
**Endfor**
Output mapping $M$

The above algorithm goes through a double loop. In the outer loop it picks up a candidate cluster $i$. Then it searches among all the other cluster to find the node called *maxnode* that will lead to maximum reduction of $F(CC, M)$ if we exchange the mapping of $i$ and that of *maxnode*. After this exchange, the algorithm continues to improve the current mapping by examining the next candidate cluster $i + 1$.

We should point out that in the inner loop the cost functions for mappings $M$ and $M_2$ do not need to be computed completely. Only the difference between the cost functions is needed. The reason is that if a communication link is not adjacent to either cluster $i$ or $j$, the cost incurred on this particular link is the same for both mappings $M$

and $M_2$. Therefore only communication links that are adjacent to cluster $i$ or $j$ need to be considered at each step.

This simplification makes the complexity of the algorithm $O(pm)$ where $m$ is the number of communication links between clusters. This is because the outer loop is executed $p$ times, and for the inner loop the total computational cost sums up to $O(m)$ since each link is examined at most twice, once for each end point. Since $m \leq p^2$, the total complexity is at most $O(p^3)$. This complexity is acceptable for most medium to coarse grain parallel machines since the number of processor seldom exceeds 1000.

We now compare the performance of our heuristic with the original Bokhari's algorithm in mapping 6 groups of test graphs. We use the trivial mapping of cluster $i$ to processor $i$ as the initial assignment for both algorithms. Each group group contains 10 graphs from GJ and PDE numerical computing graphs. We vary the number of clusters for each graph in a group. In the first group each graph has $p = 8$ clusters. In the second group $p = 16$ clusters, and so on. The results are listed in Table 2.1. For each group we compute the average of the following two values among the 10 graphs tested:

$$Gain\_ratio = \frac{FCC(trivial) - FCC(bokhari)}{FCC(trivial) - FCC(modified)}.$$

This value indicates how much gain from the trivial mapping we have retained by using the modified algorithm instead of the original Bokhari's algorithm. Notice that both algorithms use the trivial mapping as the initial mapping and then try to improve upon it.

$$Cost\_ratio = \frac{Running\_time(bokhari)}{Running\_time(modified)}.$$

This indicates the speedup in terms of complexity we have achieved using the modified algorithm.

| # Proc | p=8 | p=16 | p=32 | p=64 | p=128 | p=256 |
|---|---|---|---|---|---|---|
| *Gain_ratio* | 96.2% | 85.2% | 87.8% | 83.3% | 74.7% | 90.1% |
| *Cost_ratio* | 16.7 | 18.3 | 22.8 | 34.3 | 46.3 | 49.1 |

Table 2.1: Mapping obtained by different algorithms.

It can be observed that the performance of our algorithm is not much worse than that of the Bokhari's algorithm in terms of the cost function $F(CC)$. This algorithm has retained most of the gain achieved by the original Bokhari's algorithm. Moreover, the running time of our algorithm is reasonable for handling a system with 32 processors or more (a few seconds for 64 processors and less than 1.5 minutes for 256 processors) while the original Bokhari's algorithm can be almost 50 times slower (more than 1 hour for 256 processors). Thus, our algorithm produces a solution with a performance competitive to that of the Bokhari's algorithm by spending a reasonable amount of time.

## 2.4 Classifications of Task Graphs

### 2.4.1 Regular and irregular task graphs

Task graphs can be *regular* or *irregular*. Although these terminologies are commonly encountered in the literature, no precise definitions have been given to them.

We attempt to characterize regular task graphs as those permitting a compact representation using a set of parameters, and irregular task graphs as those allowing only complete description. Consider the example of Gauss-Jordan algorithm. If the matrix is dense, we could derive the task graph by the dimension of matrix, $n$, because we know the task graph has the shape shown in figure 2.2, regardless of the matrix content. For a given $n$ and the machine parameters, all the weights of nodes and edges can also be determined.

On the other hand, if the matrix is sparse, although the same regular task graph can be used, there are many empty tasks due to missing matrix entries, and for large matrix such graphs are obviously undesirable. After deleting useless tasks, the task graph can no longer be described using only the matrix dimension, instead it becomes problem specific, and exhibits irregular pattern. For example, see figure 2.3.

Generally speaking, if a task graph allows compact representation, its structure is well-defined, and manual parallelization is relatively easy, while irregular graphs are not well-structured and difficult to be parallelized efficiently. Note that there is one

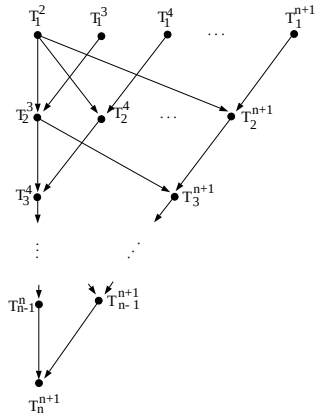



Figure 2.2: The regular task graph structure of dense GJ algorithm

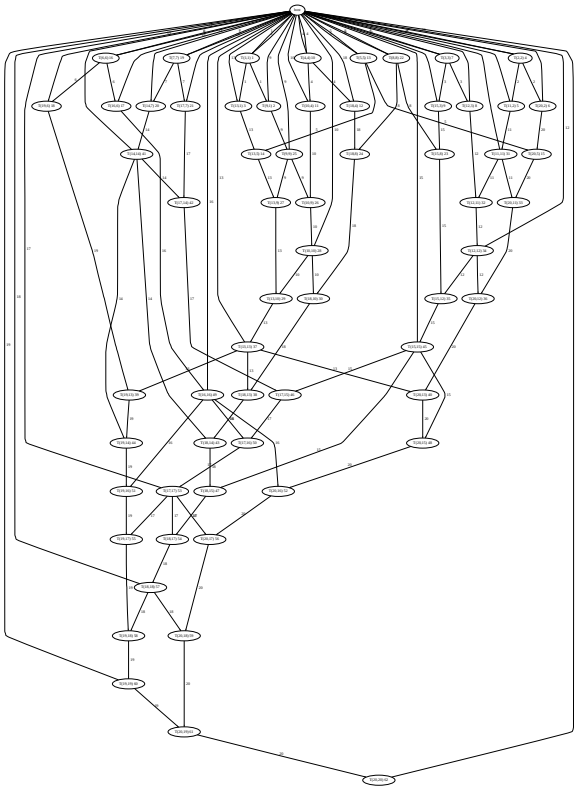


Figure 2.3: The irregular task graph structure of sparse GJ algorithm

kind of task graphs that has a regular structure but the weights are irregular. Such problems are also considered to be irregular, although to a lesser degree than graphs with irregular structures. One example of such cases is the non-adaptive FMM N-body task graph introduced in chapter 3.

Another way of classifying task graph as regular/irregular, which is essentially equivalent to the one above, is to check whether the task graph is input data dependent. A regular task graph for a fix dense matrix algebra operation, such as the Gauss-Jordan algorithm, depends only on the matrix dimension. But an irregular graph like the sparse GJ graph will be altered if a different sparse pattern is introduced. Alternatively, if we focus on the real code instead of algorithm, a regular problem has affine loop bounds which are also input independent, while an irregular problem has indirect, thus maybe input dependent, loop bounds. This issue is discussed in papers concerning the CHAOS system. See e.g. [34].

To summarize, although there are no formal definitions of regular/irregular task graphs, in practice our rather ad-hoc classification is sufficient to distinguish these two kinds of problems.

### 2.4.2 Static and dynamic task graphs

Another classification of task graphs is based on whether the task graph will change during the course of execution. Note that unless the execution is iterative, and the task graph represents the operations of one iteration, this classification does not apply. In a non-iterative problem such as the GJ algorithm, the task graph is used only once. For certain iterative problems, such as Jacobi or Gauss-Seidel methods to solve dense linear equations, the task graphs stay the same throughout the iterative solving process. We can call such task graphs *static*. For other iterative problems, the task graphs could change, either in weights of node and edges, or even in structure, during the course of execution. Such changes are caused by the iterative modification of the input data. We call them *dynamic*.

Based on our discussion in the last subsection, we could say that if an iterative task graph is dynamic, it has to be irregular, since otherwise the task graph is input

insensitive and will not be altered by the underlying data. Therefore, the properties of irregular and dynamic are related. Many problems, including all three iterative application studied in this thesis, are both irregular and dynamic, although there are problems that are irregular but static, such as the iterative sparse triangular solver studied in a paper by Chong et al. See [11].

Scheduling for a static task graph is totally dependent on the initial graph, since the schedule for the first iteration may be reused for all time, although sometimes pipelining technique can be employed to overlap consecutive iterations in order to improve overall performance. For instance [59] uses iteration overlapping for the Laplace iterative problem.

For a dynamic task graph, this is not the case, since the task graph will start to drift from the initial version, causing performance degradation when the weights are altered, or even rendering the schedule invalid when the structure of the task graph changes significantly. Therefore, the recycling of schedule, and rescheduling issues, are the keys to the parallelization of such problems, and they form the central theme of this thesis.

### 2.4.3 Two subclasses of dynamic problems

Here we will mention two subclasses of dynamic problems. For the first subclass, namely the "stepwise slowly changing" dynamic computations, the changes occurs over a number of time steps at a low rate, and behave as weight alterations up to a certain point. Many physical domain phenomena have this property, such as the N-body problem. For others, the dynamic changes can be characterized as rapid and unpredictable. For example, the radiosity calculation, an application in computer graphics, is iterative. The task graph is trivial at the initial stage, but quickly expands during each iteration. The parallelization of this problem is studied in Singh's thesis ([49]), although the task graph model is not employed. The method proposed is that each processor queues its newly expanded workload at run-time and steals workload from other processor's queue when its own queue is exhausted. No compile time method is successful for this problem.

Intuitively, the class of "stepwise slowly changing" problems permits certain degrees of schedule reuse, and reschedule will be feasible due to the acceptable overhead. On the other hand, rapidly dynamic problems are extremely difficult to handle, and currently the only reasonable method is dynamic scheduling. This is exactly what was used in Singh's thesis. Performances for such problems are unsatisfactory, and there is no solution as far as we know.

The main focus of this thesis is on "stepwise slowly changing" problems, where significant performance gains can be achieved. For simplicity, in the later chapters we refer these problems as "slowly changing".

# Chapter 3

# Scheduling For the Fast Multipole N-body Simulation

## 3.1 Introduction

In this chapter, we consider the parallelization of the N-body simulation, which is a very common problem encountered in the domain of scientific computing. Our major contribution is that we apply the task graph scheduling method and study the problem as a representative of irregular and dynamic problems in general. This is an entirely new approach, and we are investigating the benefit and applicability of our scheduling system, namely PYRROS.

As a representative of irregular/dynamic problems, the task graph structure of the 2-D FMM N-body computation will be presented first. Then our compile time scheduling system will be employed to exploit parallelism in the unbalanced task graph. Later, we will also present some results of how our run-time reschedule heuristics are used to deal with possible drastic changes in the task graph, but this will be postponed to chapter 5, after the introduction of these new heuristics in chapter 4. In this chapter we concentrate on the initial compile-time scheduling results.

### 3.1.1 Problem definition

Given a distribution of points (particles) in space, and a rule governing their interactions, e.g. gravitational force, the problem of N-body simulation is to compute the movement of particles. This movement can be discretized in time steps, during each of which the forces exerted on all particles are computed, and the positions of these particles are updated base on the forces. The skeleton of the computation method involved is listed in Figure 3.1.

The N-body simulation is widely-encountered in particle physics and astrophysics. In fact, the astrophysics version, which involves the gravitational interactions between bodies in the universe, is called the classical version. Figure 3.2 illustrates a twin-galaxy configuration found in the classical N-body simulation.

For each time step,

- Compute the gravitational force of each particle.
- Compute the new position of each particle based on the force calculation.

Figure 3.1: The N-body simulation algorithm.

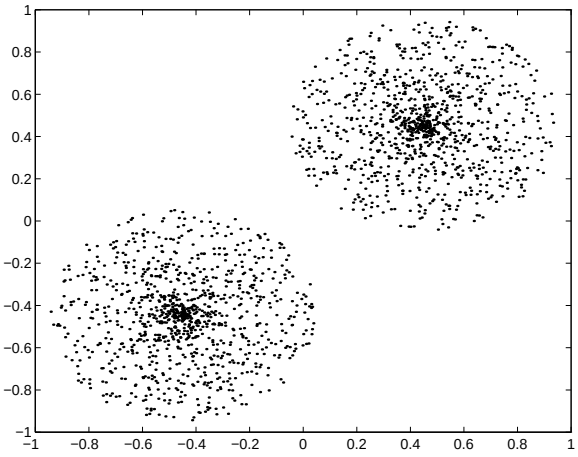



Figure 3.2: A twin-galaxy distribution of particles in space

In two-dimensional space, the force applied on a particle located at position $z_j$, a complex number with the real part as the X coordinate and the imaginary part as the Y coordinate, is:

$$F_j = \sum_{i=1, i \neq j}^{N} \frac{A_i}{z_i - z_j}, \quad j = 1 : N$$

where $A_i$ is the charge of particle $i$ located at $z_i$ and $N$ is the total number of particles. The direct summation for the above expression costs $O(N^2)$. Two fast sequential algorithms are available: the Barnes-Hut Algorithm [5], and The Fast Multipole Method (FMM)[28].

The method we consider here is the FMM algorithm, because of the following reasons:

- Theoretically, the FMM algorithm has lower complexity than the Barnes-Hut method.
- More importantly, the task graph dependence structure of the Barnes- Hut algorithm is relatively simple, consisting of independent trees, one for calculating the force on each particle, rendering it easier to parallelize by using simple load balancing approaches. On the other hand, the FMM task graph is much more complex, making it the kind of problems where automatic scheduling tools could be useful. This is particularly true for the adaptive version of the algorithm.

Our consideration is also limited to the 2D version of the algorithm. There is a 3D version of the same algorithm, but we will not consider it in this thesis, because the task graph structure and parallelization issues are almost identical for both, but for the 3D version, the computation involved in the task nodes are extremely arcane and the amount of work for implementation is very high, which will divert our attention from the parallelization issue of the problem. For the same reason, a number of other works ,such as Singh's parallelization of the FMM algorithm [48], also focused on the 2D instead of the 3D version.

The basic idea of the algorithm is to subdivide the space into boxes, and the evaluation of particle positions between far-away boxes can be approximated by series of expansions. The coefficients of the series are computed once and used for all "far-away" boxes. Depending on the input distribution of particles, the algorithm comes in two flavors:

1. *Non-adaptive:* All boxes are of the same size. This is applied to uniform particle distributions.
2. *Adaptive:* Boxes are subdivided further in the regions of higher particle density. This is used for highly non-uniform distributions.

The algorithm for computing gravitational forces is briefly described as follows. Assume that the 2D space is recursively divided into $4^l$ boxes where $l$ is the level of subdivision. Each box contains $s$ particles on average, as in the case of a uniform particle distribution. We have $4^l s = N$ and the force expression is rewritten as:

$$F_j = \sum_{i=1, i\neq j}^{N} \frac{A_i}{z_i - z_j} = \sum_{box=1}^{4^l} \sum_{z_i \in ibox} \frac{A_i}{z_i - z_j}, \quad j = 1 : N.$$

The inner summation is approximated by a multipole series expansion with $r$ terms. Assuming that $z_i$ belongs to a box called $b$ and the geometric center of this box is at position $z_0$, we consider the evaluation for far-away particles $z_j$, i.e. $|z_i - z_0| \leq c \times |z_j - z_0|$ where $c < 1$ is a separation factor. Then

$$\sum_{z_i \in b} \frac{A_i}{z_i - z_j} = -\sum_{z_i \in b} \frac{A_i}{(z_j - z_0) - (z_i - z_0)} = -\frac{1}{(z_j - z_0)} \sum_{z_i \in b} \frac{A_i}{1 - \frac{z_i - z_0}{z_j - z_0}}$$

$$\approx -\frac{1}{(z_j - z_0)} \sum_{k=0}^{r} \frac{a_k}{(z_j - z_0)^k}$$

where $a_k = \sum_{z_i \in b} A_i (z_i - z_0)^k$ is a *constant* term in the expansion since it only depends on the particles within this box and it can be computed once and used for *all* far-away particles. In this way we reduce the time complexity from $O(s^2)$ to $O(rs)$ for evaluating the interaction between $s$ far-away particles. The value of $r$ depends on the separation factor $c$ and the accuracy of the approximation. The truncation error in the series expansion is $O(c^r)$.

To reduce the total complexity further the computations must be performed hierarchically using the tree structure. At any level $l$ and for all particles in a box, interactions are evaluated at its neighbors, and at these "far away" boxes next to these neighbors. For example Figure 3.3 shows a 3-level non-adaptive subdivision. At level $l = 3$, for box O, we evaluate all particles in its 8 neighbor boxes using the direct summations, while the boxes shown with X are evaluated using the series At the next level $l - 1$, the boxes are 4 times larger than the previous level boxes and the series expansions are used to evaluate "far away" boxes at this level. The number of boxes in the list of any single box at any level is always less or equal to 27 in 2D space. At the root of the tree the entire 2D space particle evaluation has been covered. For more details see Greengard's thesis [28].

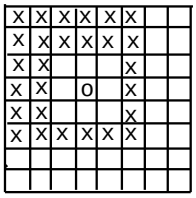


Figure 3.3: A 3-level non-adaptive subdivision. X are the boxes in the interaction list of box O. Any box has most 27 boxes in its interaction list and at most 8 neighbors

For such kind of applications, hand-optimization can be extremely difficult, and automatic scheduling method can be a good approach. On the other hand, since N-body simulation involves the iterative updating of particle positions, we can not afford to use PYRROS to schedule force computation involved at each iteration, since the overhead may be too high. But fortunately, as with a large class of dynamic problems (which we casually called "slowly changing" in chapter 2), the schedule can be derived from the initial input distribution and reused over a considerably large number of iterations so that the compile-time scheduling cost can be amortized. Because particles move *very slowly* in the 2D space, we will only use PYRROS to schedule the first iteration, and use the same schedule for many iterations. Notice that from one iteration to the next, task weights of the graph may change. But the pace is slow and performance degeneration should not be rapid, as we shall discuss later.

For the rest of the chapter, we start with the non-adaptive FMM algorithm in section 3.2. This version of the algorithm has a regular task graph structure but irregular weights. The limited irregularity makes it easier to obtain very high speedup for uniformly distributed test cases, so it's not intended to be our main result. However we describe the task graph in considerable detail because without such explanations it's impossible to move on to the adaptive version of the algorithm which is an extension of the non-adaptive version. In section 3.3, we present our results on the initial scheduling for the adaptive FMM algorithm. Next in section 3.4, we discuss some important issues in the iteration execution of the initial schedule: performance vs. overhead trade-off, and the rate of performance degradation.

## 3.2 Task Graph Scheduling for the Non-adaptive Partitioning:

We start our discussion from the basic non-adaptive version of the FMM algorithm for uniformly distributed particle configurations. The resulting task graphs are regularly structured but have irregular weights.

**Graph Structure:** Figure 3.4 shows a 2-level space subdivision and its corresponding quadtree.

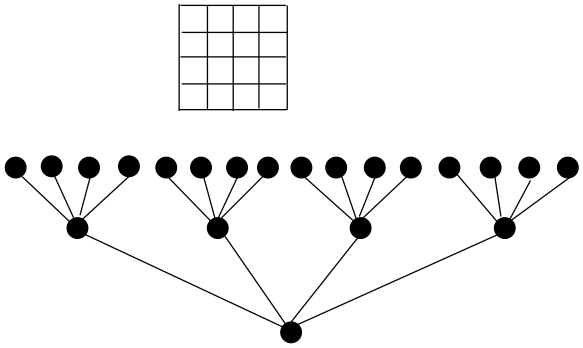


Figure 3.4: A 2-level non-adaptive subdivision.

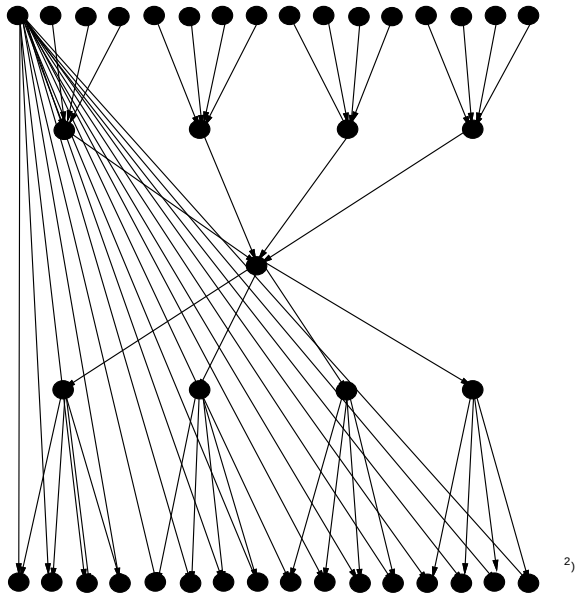


Figure 3.5: Task graph for a 2-level subdivision. The order of task weights are also shown. Only communication from one leave of the upward pass tree to the leaves of the downward pass tree is shown. The others are deleted for simplicity of presentation.

The basic structure of the task graph is a concatenation of this in-tree with an out-tree which is its reverse. Because of the regular subdivisions, each is a complete quad-tree. The leaves of the tree correspond to the boxes at the finest level, and the internal nodes correspond to the boxes at higher levels. Following the thesis of Greengard [28], we call the in-tree "upward pass" and the out-tree "downward pass". There are additional edges between the two parts of the graph, which represent interactions between boxes. Figure 3.5 is an example of the task graph for a level 2 division. For clarity we only draw the communication edges for one leaf node.

Although in this graph, every leaf communicates with all the other leaves, this is not true for divisions of higher levels. A leaf communicates with its neighbors and "interaction list", which is essentially the list of all second-nearest neighbors, see figure 3.3. The number of boxes in each of the lists is bounded by a constant, which is 27 for the 2D case. So the graph has $O(4^l)$ edges. It's not easy to draw these graphs clearly but from the following description it should be easy to understand their structure.

The task graph has a regular structure, but task weights depend on the input distribution and is irregular. We only describe the order of the weights in terms of number of particles in a leaf box, $s$, and number of terms in the series expansion, $r$. In a uniform distribution the number of particle in each box would vary from box to box but not by much. We describe the functions for each node below.

1. A leaf node in the upward pass – Computation weight is $O(rs)$.
   (a) Generates multipole expansions from the particles in the box.
   (b) Sends the multipole expansion coefficients to its parent box in the upward pass. and also to those boxes in the interaction list corresponding to the leaf level of the downward pass.
   (c) Sends the particle positions to those neighboring leaf boxes, including itself, in the downward pass.
2. A non-leaf node in the upward pass –Computation weight is $O(r^2)$.
   (a) Receives the multipole expansions from each child, and shifts the expansions

to its own box center. Then sums them up to obtain the multipole expansion for the whole box.

(b) Sends the multipole expansion to its parent, if it's not the root, and also to the boxes in its interaction list in the downward pass.

3. A non-leaf node in the downward pass – Computation weight is $O(r^2)$.

(a) Receives the multipole expansions from the boxes in its interaction list in the upward pass, then converts these multipole expansions into local expansions. See Greengard thesis [28].

(b) Receives the local expansion from its parent, shifts it to its own center, then adds it to the local expansion computed in (a) above to obtain the total local expansion at this box.

(c) Sends the local expansion to each of its children.

4. A leaf node in the downward pass – Computation weight is $O(s^2 + rs)$.

It does the same as a non-leaf node, except that it does not send local expansions since it has no children, plus:

(a) Receives the particle positions from neighboring boxes in the upward pass, and evaluates nearby interaction using direct pairwise summation.

(b) Computes the far away interaction using the local expansion, by evaluating the expansion at each particle in its box.

(c) Sums up the "nearby" and "far away" interactions, obtaining the total interaction and updates particle positions.

**Performance results:** We use an example of 80000 randomly distribution particles, with a 5-level (32*32) subdivision. This graph has about 2700 nodes and 30000 edges. PYRROS scheduler can handle graph with such size quite rapidly. The speedup is almost linear, shown in table 3.1.

The largest problem we were able to run on the NCUBE-2s machine is a problem with $800,000$ particles, with 128 processors, each having 16MB of memory.

| # proc | Time-run (s) | Speedup-run | Compile-estimation |
|---|---|---|---|
| p=4 | 320 | 3.96 | 3.98 |
| p=8 | 160 | 7.90 | 7.95 |
| p=16 | 81.3 | 15.6 | 15.7 |
| p=32 | 41.6 | 30.4 | 31.0 |
| p=64 | 21.5 | 59.0 | 61.0 |

Table 3.1: PYRROS performance for a uniform distribution with 80000 particles, one iteration on the NCUBE-2s.

## 3.3 Scheduling for the Adaptive FMM Algorithm

Although the non-adaptive algorithm has very good performance for uniform distributions of particles, it cannot handle highly non-uniform distributions encountered in some applications, such as galaxy simulations. For instance, in the twin-galaxy type configuration depicted in figure 3.2, the particles are highly concentrated in two regions of the space, and especially the cores of each galaxy. It's clear that equally sized boxes can not give a reasonable partitioning of the particles. Adaptive subdivision must be used for such distributions.

Figure 3.6 illustrates an example of an adaptive subdivision and the corresponding unbalanced tree for this partition. The real task graph consists of the concatenation of this in-tree with the symmetric out-tree and edges between the two parts.

**Graph structure:** The task graph structure is similar to that of the non-adaptive case in the sense that it still has an upward pass (in-tree) and a downward pass (out-tree). But the tree is unbalanced, and the height is larger for the places with higher particle density. So the tree structure varies, depending on the input distribution. For galaxy distributions, the tree is highly unstructured. The function of the nodes are basically the same as in the non-adaptive algorithm, with additional complications which make the parallelization much more difficult than the non-adaptive version. The complications are two-fold:

- First, due to the uneven tree height, new types of far away interactions that are not present in the non-adaptive case are introduced between boxes at different levels. Such interaction can not be treated in the same manner as the interactions

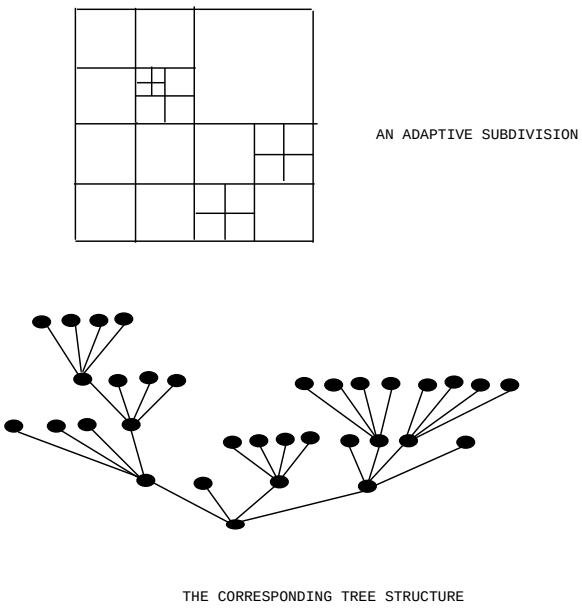



Figure 3.6: An adaptive subdivision for non-uniform distribution of particles.

at the same level, and must be handled separately.

- Second, in the non-adaptive case, the operations relevant to a box can be completely determined, given its level in the tree $l$ and its center position within that level, since the tree is the same for any input. In the adaptive case, however, the function of any box at level $l$ with center position c is input dependent and can not be determined beforehand. This is because with different input the task can be either leaf or non-leaf, and the interacting tasks also vary. So when we schedule the adaptive FMM algorithm onto parallel machines, we must use certain data structure generated from the input distribution to provide such information to the tasks.

The data structures includes different lists of interactions. For the name of list we use the notion introduced by Greengard [28]. In particular, a "colleague" of a box $b$ is a box that is at the same level as $b$ and is neighboring to $b$. (For non-adaptive case, colleague and neighbor are equivalent, but for adaptive case a neighbor may not be a colleague of $b$.)

For a leaf box $b$ at level $l$, we have:

1.) $U$ list: this list contains all direct neighbors of $b$. Note that a neighbor of $b$ may

not necessarily be at level $l$ due to irregular tree height, but all interactions between $b$ and $b' \in U(b)$ are calculated using direct method.

2.) $V$ list: this list is essentially the same as the "interaction list" in the non-adaptive case. It contains all children of colleagues of $parent(b)$. Interaction between $b$ and $b' \in V(b)$ is computed by translation of multipole expansions. Note that all boxes in this list is at the same level $l$ as $b$, this is a condition for translating expansions.

3.) $W$ list: this list contains descendents of $b$'s colleagues that are not themselves in $U(b)$, but their parents are in $U(b)$. These boxes also interact with $b$, but since they are all smaller than $b$, they need to be handled differently from those in $V(b)$. The algorithm will take the multipole expansion due to all $b' \in W(b)$ and evaluating the expansion at each individual particle in $b$.

4.) $X$ list: It's the "mirror image" of $W$ lists: $b' \in X(b)$ iff $b \in W(b')$. These are the boxes interacting with $b$ but with larger size. The algorithm has to take each particle in $b' \in X(b)$, generate a local expansion and move it to the center of $b$.

For a non-leaf box, we still have $V$ and $X$ lists as above, but there are no $U$ and $W$ lists for such boxes.

To illustrate these lists, we adopt a complex irregular partitioning from [28], and show the interaction lists of a particular box $B$ in figure 3.7. In the figure boxes marked $U$ are those in $Ulist(B)$, and so forth.

With these lists in mind, the functionalities of node in the task graph can be briefly summarized as follows. Notice that each node is required to carry out all the functions of its counterpart in the non-adaptive version. But there are extra functionalities for each task, caused by the irregularity of the tree:

1. A leaf node in the upward pass:

   (a) Generates multipole expansions from the particles in the box.

   (b) Sends the multipole expansion coefficients to its parent box in the upward pass, and also to those boxes in $V$ and $X$ lists corresponding to boxes in downward pass. Unlike the non-adaptive case these boxes may or may not be at the leaf level.

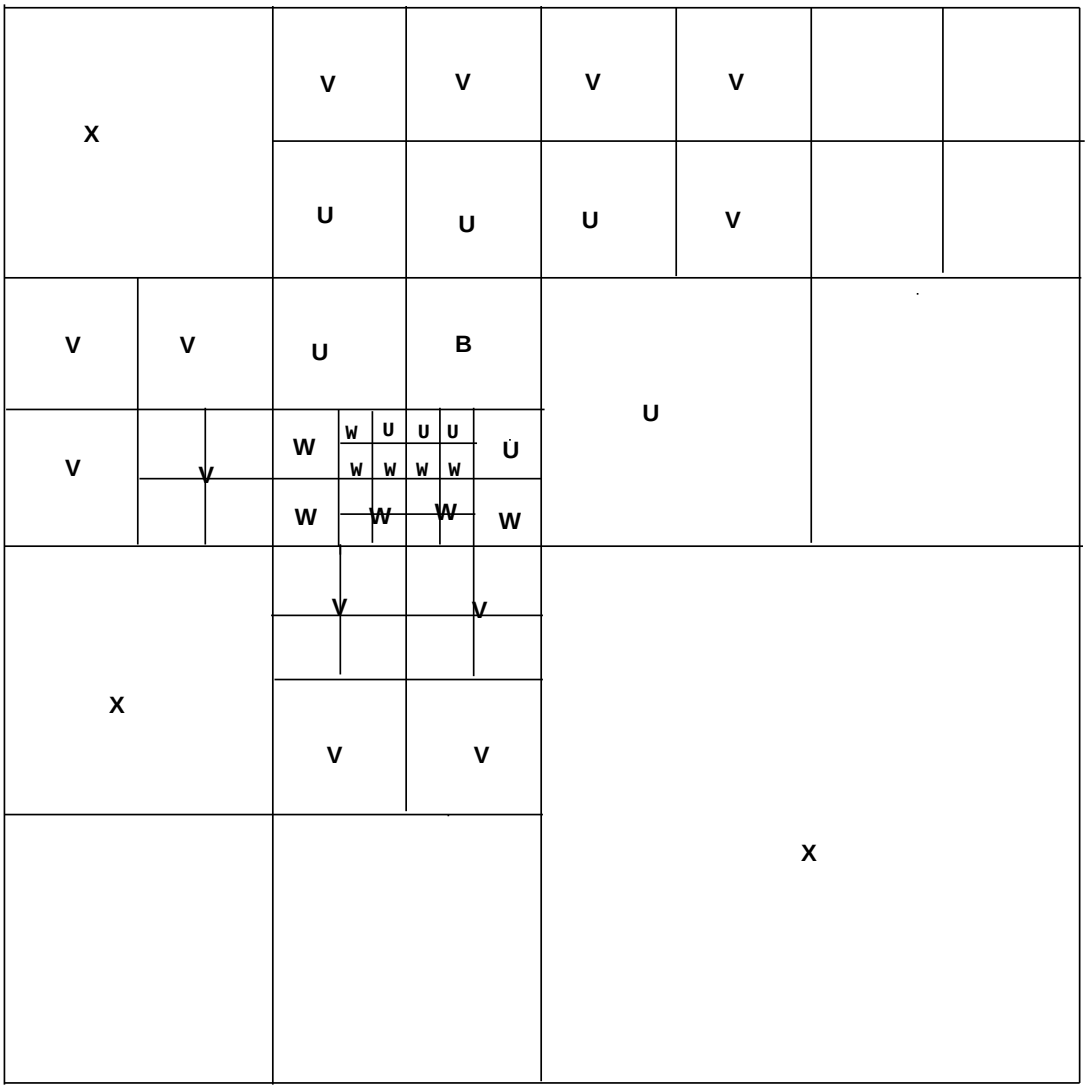



Figure 3.7: The various interaction lists of box B in an irregular subdivision

(c) Sends the particle positions to those neighboring leaf boxes in $U$ list, and this box itself, in the downward pass. it also sends the particle position to the boxes in $W$ list in the downward pass.

2. A non-leaf node in the upward pass:

   (a) Receives the multipole expansions from each child, and shifts the expansions to its own box center. Then sums them up to obtain the multipole expansion for the whole box.

   (b) Sends the multipole expansion to its parent, if it's not the root, and also to the boxes in its $V$ and $X$ lists in the downward pass.

3. A non-leaf node in the downward pass:

   (a) Receives the multipole expansions from the boxes in its $V$ list in the upward pass. Then converts these multipole expansions into local expansions, see Greengard thesis [28].

   (b) Also receives the particle positions from $X$ list boxes in the upward pass, but instead of direct summation, converts the field each particle in the $X$ list box into a local expansion about the center of the receiving box, and adds the local expansion to the total expansion.

   (c) Receives the local expansion from its parent, shifts it to its own center, then adds it to the local expansion computed in (a) above to obtain the total local expansion at this box.

   (d) Sends the local expansion to each of its children.

4. A leaf node in the downward pass:

   It does the same as a non-leaf node, except that it does not send local expansions since it has no children, plus:

   (a) Receives the particle positions from $U$ list boxes in the upward pass, and evaluates nearby interaction using direct pairwise summation.

(b) Computes the far away interaction as follows:

i) Using the local expansion, by evaluating the expansion at each particle in its box.

ii) The above still does not include the contribution from the "far-way" boxes in the $W$ list. So we evaluate such interactions by calculating the multipole expansion due to each box in $W$ list at each particle in the current box, and add the interaction to obtain the total "far-away" potential field value.

(c) Sums up the "nearby" and "far away" interactions, obtaining the total interaction and updates particle positions.

Given an input distribution, these lists can be generated simultaneously with the N-body task graph. The scheduler will then distribute these lists to each node program such that only those tasks executed in that particular node will be included. In a parallel execution, therefore, each task can get this information and complete its function.

It should be mentioned that this is similar to the Inspector-Executor paradigm in the CHAOS system ([34]). The lists are used at run time by the executor component.

**Performance results:** Due to the irregular structure, it's necessary to use an automatic scheduling tool to schedule such task graphs. PYRROS scheduling tool is well-suited for this purpose. Our example is a highly non-uniform one galaxy distribution with 40000 particles. This distribution results in a highly unbalanced tree, with the maximum depth of the leaves being as large as 9, and the minimum depth of the leaves being only 1. Such kind of distribution is very difficult to parallelize. Nevertheless, PYRROS is able to achieve good performance, with speedup of 40 using 64 processors, see Table 3.2.

## 3.4 Performance Issues in the Parallel FMM Algorithm

PYRROS is a static scheduling tool, while the N-body problem is irregular, iterative and dynamic. So there are some major issues that must be addressed. First, PYRROS can exploit parallelism from task graphs, but this comes with the penalty paid in scheduling cost. So is this cost justifiable? Second, the particle movements in the input

| # proc | Time | Speedup-run | Compile-estimation |
|---|---|---|---|
| p=4 | 105.0(s) | 3.94 | 3.97 |
| p=8 | 53.7 | 7.76 | 7.83 |
| p=16 | 28.3 | 14.7 | 15.1 |
| p=32 | 16.6 | 25.0 | 28.0 |
| p=64 | 10.5 | 40.0 | 45.4 |

Table 3.2: The speedup of PYRROS for the adaptive FMM on NCUBE-2s. One galaxy distribution with 40000 particles for one iteration.

distribution will certainly alter the task graph weights, so how does the schedule derived for the initial task graph perform iteratively? We discuss the two issues next.

### 3.4.1 Performance vs. overhead trade-off for the N-body computation

The major overhead in the adaptive fast multipole N-body method is the construction of a task graph and the scheduling of this graph. During task graph construction, space is partitioned into boxes based on the input distribution. The interaction lists of each box and the task graph are then derived. The cost of task graph generation is $O(b)$ where $b$ is the number of boxes in the partitioning. The scheduling cost, on the other hand is still $O((e+v)\log v)$, which is $O(b\log b)$. As for the computation cost, it is theoretically $O(N)$, but varies depending on the distribution, and this computation is repeated over a large number of iterations.

We compare the PYRROS solution with hand-made N-body code. The hand-made code uses two types of mappings: the block level mapping and the cyclic level mapping. These two methods take tasks from each level of the tree structure and map an equal number of tasks onto each processor.

The two regular mapping methods, especially the cyclic mapping, can actually perform well for uniform and even for single galaxy distribution due to certain degree of regularity in the task graph. PYRROS may not be able to offer much improvement in terms of performance for such cases. But for irregular particle distribution such as twin galaxies with different density patterns, PYRROS offers improvements especially for larger number of processors. Our example is a twin galaxy distribution with 6000 particles (5000 and 1000 particles for each galaxy). The parallel time with unit seconds

for one time step is shown in Table 3.3. It can been seen that the block mapping results

| # proc | 6000-particle twin galaxies | | |
|---|---|---|---|
| | PYRROS | Block | Cyclic |
| 4 | 12.2(s) | 13.4 | 12.6 |
| 8 | 6.56 | 7.80 | 7.38 |
| 16 | 3.83 | 4.93 | 4.44 |
| 32 | 2.48 | 3.77 | 2.98 |
| 64 | 1.89 | 3.15 | 2.53 |

Table 3.3: The performance of PYRROS and level scheduling on N-body graph.

in the longest parallel time, while the cyclic mapping improves the performance due to much better load balancing. For the single galaxy distribution, cyclic mapping is doing so well that it is almost as good as PYRROS. But for this irregular twin galaxy case, PYRROS utilizes more information and offers parallel time that is up to 40% less than that of the block level mapping method and 25% less than that of the cyclic mapping. This example demonstrates that automatic scheduling offers a better solution when distribution of particles is irregular.

It should be noted that PYRROS carries an additional scheduling overhead compared to the hand-made code. For instance, when $p = 64$, the code produced by PYRROS takes 1.9 seconds for one time step (cf. 2.5 seconds by cyclic mapping), while the cost of scheduling is about 5 seconds for this problem on a SUN 4/340 front-end workstation of the NCUBE-2 machine. Therefore, for more than 9 iterations, the total gain in performance should exceed the initial cost. In our experiment, the same schedule is reused for 100 iterations with little performance loss. We have found that the scheduling approach will outperform the cyclic mapping by 24% in terms of the total running time and overhead after 100 iterations.

One advantage of PYRROS is the ability to produce good, albeit non-optimal, schedule at reasonable cost. Other scheduling algorithms such as the ETF algorithm [33] has higher complexity ($O(v^2)$). In experimental comparisons we find that ETF could yield better schedule than PYRROS but for moderate to large number of processors the cost of ETF is prohibitive. See [21].

We should mention that similar performance-overhead trade-off was observed by

Chong et al. ([11], [12]) using DSC clustering method on irregular sparse matrix task graphs.

### 3.4.2 Performance of scheduling in the iterative execution of FMM

We use PYRROS to schedule one iteration and then execute the same schedule for many iterations. Since the positions of particles are updated during each time step, they could move from one box to another, resulting in changes in the task graph. In the non-adaptive partitioning, the movement changes the weights. In the adaptive partitioning, the particle movement can lead to changes of the graph structure if the space is re-partitioned. In order to maintain the graph structure and reuse the schedule, we choose to maintain the same partitioning before re-scheduling. Thus the particle movement only affects weights in our approach. We should be aware that the performance of the schedule derived for the initial iteration degrades when particles move.

In general, the movement of particles is small during each time step. Since time step size $\delta t$ has to be chosen small enough to ensure numerical accuracy and stability. This insight is crucial to almost all the previous works relating to the N-body problem, since it allows the partitioning obtained from the initial distribution to be reused. See for example, [49]. Computational scientists had a general rule of allow only very tiny movements per time step, even if the cumulative effect during the whole simulation process is dramatic. This is the only way to follow the fastest motion in the system with reasonable accuracy. In a variation of the classic N-body problem, the protein dynamics simulation, time step size has to be as little as $10^{-15}$ seconds, and it takes $10^6$ to $10^7$ steps to complete a folding process in the order of nanoseconds. See [9].

Furthermore, the PYRROS scheduler does not need precise weight information, we expect that small weight perturbations caused by particle movement should not lead to significant deterioration in scheduling performance, which is intuitive and analytically shown in [20]. Thus a schedule can be re-used for many steps, until its performance deteriorates to the point where rescheduling becomes necessary. The interesting question is how frequently we need to reschedule, since this will determine the overall performance of PYRROS for the N-body problem. We conduct experiments to investigate

| processors | slow-down(%) |
|---|---|
| p=8 | 0.4% |
| p=16 | 5.1% |
| p=32 | 7.4% |
| p=64 | 6.3% |

Table 3.4: The increase in parallel time after random weight perturbation.

| processors | slow-down(%) |
|---|---|
| p=8 | 4.0 |
| p=16 | 4.9 |
| p=32 | 6.1 |
| p=64 | 7.5 |

Table 3.5: The increase in parallel time after 100 iterations.

this issue below.

**Experiment 1.** First we choose a uniform distribution of 10000 particles, schedule the computation and record the parallel time. Then we make random perturbations to the task and communication weights, so the schedule is produced using inaccurate weights. Afterward we rerun the computation with the new schedule and measure the new parallel time. According to our run time analysis in a related paper [20], the performance should not be affected by much, provided the weight perturbation is not excessive.

We bound the weight perturbation to a random amount between -20% and +20%. Then we observe the increase in parallel time shown in Table 3.4. The increase in parallel time is virtually none for small number of processors and very modest for larger number of processors.

**Experiment 2.** Next we look at performance lost in a real dynamic iterative computation setting. Using a uniform distribution of 10000 particles, we choose a time step size of 0.001 which for this case is large enough to get sizable movement of particles at each step, but small enough to ensure convergence to single precision accuracy. our experiment runs iteratively on 8 to 64 processors. The same initial schedule is used for all iterations, and we calculate the following *slow-down ratio* to model the deterioration

of the performance: $\frac{PT_{100}-PT_1}{PT_1}$, where $PT_1$ is the parallel time for the first iteration and $PT_{100}$ is the parallel time for the 100th iteration. The results are shown in Table 3.5. Note that slow-down ratio is 7.5% for 64 processors after 100 iterations. After 130 iterations, the ratio increases to 10%, in this particular case. In other test cases, we observed slow-down ratios are generally within 10% for 100 iterations, and the need of rescheduling arises after that. Details can be found in chapter 5.

### 3.4.3 Some observations about parallel time increase

One more interesting point to mention is that, the parallel time generally stay even for many iterations with little increase, but may start to jump sharply thereafter. See chapter 5 for some figures description the parallel time trend in iterative executions. Intuitively, this is related to the space partitioning used in the FMM algorithm. Initially, particles have not been able to cross boundaries of boxes to cause any significant shift in the weights of the tasks, therefore the load balance of the processors remain undistributed. However, there will be a certain point when particle movements have accumulate to such a point that frequent "border crossings" between boxes starts to occur, and large variations of weights lead to rapid climbing of parallel time. (We are reusing the same partitioning along with the schedule, since a repartition will change the structure of the task graph and make reschedule mandatory.)

We should also mention that, generally, the algorithm we use is not very sensitive to the input distribution patterns, since the quad tree structure is fairly flexible to accommodate different distributions. However, there are some extreme and pathological cases where the performance behavior may be very undesirable. For instance, imagine all the particles are lined up in the straight line $y = \sigma$ with $\sigma$ approaches 0. Then all the particles are infinitely close to the box boundary. When the simulation starts, any tiny amount of movement downward may cause large number of boundary crossings, thus degeneration of parallel time. For such extreme case it is difficult to maintain performance by reuse schedule. However, we can shift the line and move it far enough from y=0, which will significantly delay the crossings and allow initial schedule reuse. Although the physics aspect involved is too complex to permit a detailed study, we can

say that except for such rare cases, schedule reuse and overhead recovery by iterative execution is generally feasible for slowly changing problems like the N-body.

We have demonstrated the applicability of compile-time scheduling to the FMM computation over a considerable number of iterations. Yet our study of the N-body problem must proceed further to the issue of rescheduling support, since sooner or later the slow down in parallel time will become substantial. In next two chapters, we will first develop new rescheduling heuristics for generic task graphs, and apply them to the FMM algorithm.

# Chapter 4

# Two Fast Rescheduling Heuristics for Dynamic Task Graphs

## 4.1 Motivations

As we have discussed earlier, for iterative problems, if irregularity exists in the task graph, it could be the case that dynamic changes occur across iterations. This is due to the inherent input sensitive property of irregular task graphs. Therefore, irregularity and dynamic changes are often related and coexist, as in the FMM N-body algorithm.

As we have seen in the N-body problem, scheduling is a useful tool to extract parallelism from the irregular structure. But the dynamic nature of such problems presents certain difficulties to static scheduling method. Fortunately, the class of "slowly changing" problem permits considerable schedule reuse. But sooner or later, the degradation of performance will accumulate and rescheduling is called for.

A naive approach, of course, is to restart the scheduling tool and produce a new schedule. This however will result in high overhead, because the existing schedule is ignored and the mapping is built from scratch. On the other hand, for the class of gradually dynamic problem, the task graph changes incrementally, one should hope that a new schedule should also be incrementally built upon the old one, with much lower cost than the initial schedule creation.

In this chapter, we identify two important kinds of dynamic changing patterns, and propose one incremental reschedule heuristic for each of them.

## 4.2 Low Complexity Rescheduling for Task Graphs with Dynamic Weights and Fixed Structures

We first consider the case where after iterative execution, there is no change in the structure of the task graph, but the weights of the tasks and edges have drifted from their initial configuration.

This particular kind of dynamic pattern is commonly found in problems where the underlying partitioning is maintained but the data distribution has shifted across the partitioning. In the FMM N-body algorithm, the partitioning is the recursive subdivision of space and the hierarchy of leaf and non-leaf boxes, and the data distribution pattern is the number of particles contained in each of the leaf boxes, which determines the weight of those tasks corresponding to them in the task graph (both upward and downward passes), as well as the relevant communication edges. Particle movements across box boundaries cause shifts in these weights.

For the class of "slowly changing" problems, the method we propose to handle graph weights alteration is dynamic task redistribution among processors to achieve reduction in parallel time. The method is not a simple load re-balancing but rather movement of certain "candidate" tasks to other processors to achieve possible reduction in parallel time. The process carries out local adjustments without traversal of the whole graph, thus having a low complexity.

Dynamic processor load balancing has been extensively studied in the literature, such as [35]. The common principle behind the many algorithms is that the workload of each processor is sampled and works are transferred between processors in the balancing phase. This method is ideally suited for the case where the workload in each processor does not have any dependence, i.e. independent task units with different weights are distributed among processors. On the other hand, to the best of our knowledge, there has never been a study of dynamic load redistribution when the task graph dependence are present. This is due to the fact that the latter problem is much more difficult.

For non-trivial dependence task graphs, conventional load balancing method may not result in a reduction in parallel time. Let's look at the following examples:

In the first example we have 2 processors, and the tasks in each processor form a linear dependence chain. Suppose that the weights of some tasks in processor 0 have large increases so that the load (sum of weights of all tasks) of processor 0 far exceeds that of processor 1. The load balancing approach, without regard to the structure of the task graph, will move some tasks from processor 0 to processor 1. However it's easy to see that this will only result in a longer critical path with extra communication costs. So the parallel time will only worsen. In fact, for any single processor containing tasks with chain dependence, moving any portion of the chain will introduce extra communication. So for such a task graph, the parallel time is irreducible, we should leave the schedule as is. See figure 4.1.

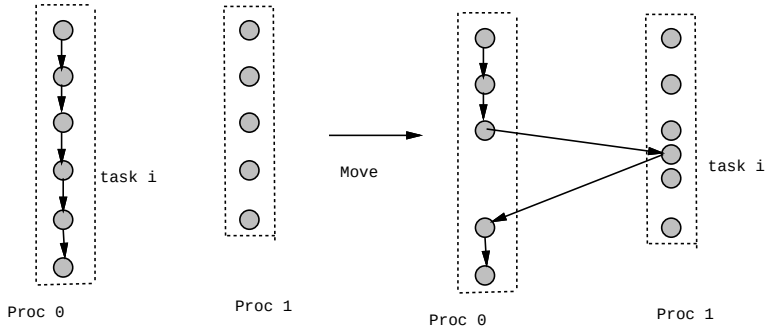



Figure 4.1: The first example where the simple load balancing fails

In the second example, we still have 2 processors, and the load of processor 0 far exceeds that of processor 1, which is the same situation as example 1. Here the last task in processor 0 does not belong to any dependence chain. But a load balancing movement of this task to processor 1 can still increase the parallel time if the communication edge $e_0$ has sufficiently high cost $C_0$ such that $C_0 > W_j$, where $W_j$ is the weight of task $T_j$. It's easy to see that in this case the finishing time of $T_i$ will increase. See figure 4.2.

Therefore, for tasks with dependences, the simple load balancing approach may not yield good results. The reason is that they were not designed with the dependences in

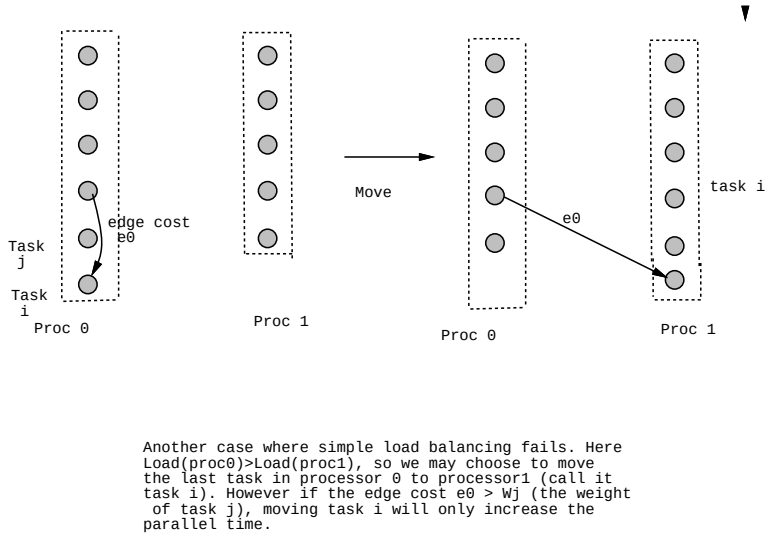



Figure 4.2: The second example where the simple load balancing fails

mind. We must generalize the load balancing approach in a setting where task graph edges are taken into account.

What we need is some new algorithm that has at least the following properties:

- It not only utilizes the information about processor load, but also the structure of the task graph.
- It has very low complexity, since a heuristic with complexity comparable to that of the initial scheduling algorithm is not interesting. We could just reschedule from scratch in this situation.

As far as we know no such algorithm exists up to now, due to the fact that the dynamic load redistribution problem has never been studied in the setting of task graph model. Next we present such a heuristic based on local adjustment. It attempts to carry out load transferring across processors in such a fashion as to avoid the negative effect of parallel time increase. Although there is no performance guarantee theoretically, as is the case for almost all heuristics for intractable problems, such load transferring will hopefully reduce the parallel time, since we are careful to avoid those "bad moves" that would have detrimental effects.

## 4.3 A Localized Schedule Readjustment Heuristic

In a task graph, the changes of weight might not be even across all the tasks and edges. For some tasks, the changes are more drastic. Intuitively, the portions of the graph around such tasks are disturbed more heavily by the changes, thus the adjustment should concentrate on these parts of the graph.

Now that we want a low complexity heuristic, we must not traverse the whole graph to compute the new critical path, since this will cost $O(e)$, which is not much lower than the global algorithms used in PYRROS. Instead we should have a heuristic that identifies the places in the graph that are heavily disturbed, and readjust these parts. We will focus on those tasks with large increases in computation weights.

Before applying our heuristic, we need to carry out the following preprocessing:

**Preprocessing step**

This step samples all the tasks and reports the tasks with large increase in weight. The increase should be in absolute value, not relative ratio of the increase to the original weight, since it is the absolute value that contributes to the parallel time. Assume this step returns a subset of tasks $V_{inc}$, with $|V_{inc}| = n_c$.

At the same time, the total load of each processor is obtained, namely $L_i$, $i = 1...p$.

This preprocessing has cost $O(n)$ where $n$ is the number of tasks. Note that a load sampling procedure is a necessary precondition for any method to work, since we need to know the current weights of all tasks in any case, so we will not consider this as part of our algorithm. However, this preprocessing will limit our attention to those $n_c$ tasks, therefore localize our algorithm, which is described below.

**Algorithm Description**

The algorithm iterates through all the $n_c$ task selected in the preprocessing step. For each $T \in V_{inc}$ the algorithm examines the possibility of a load redistribution movement. So we discuss the execution of the algorithm in one such iteration on a particular $T \in V_{inc}$ next.

Given such a $T$, we know the fact that the weight of $T$ has large increase over its original value, which was used to derive the initial schedule. The result can be that

those tasks ordered after $T$ has been delayed by the weight increase of $T$, although we can not be sure since we can not afford to compute the critical path or the start time of any task. For our purpose, the heuristic will make such an assumption and start trying to compensate for the possible delays in start time for some tasks ordered after $T$, by moving them to some lightly loaded processor if the number of processor is bounded, or to a new free processor if the number is unbounded. In practical situations we are using bounded number of processors.

From our discussion of the failure of simple load balancing, we know the following:

- The tasks moved must not be in the same dependence chain as $T$.
- If any task is moved, all of the tasks within its dependence chain must be moved simultaneously.

Therefore, the adjustments are restricted to moving only task chain(s). To be conservative we will move only one such chain at a time. What the algorithm does next is to identify the head and tail of this candidate chain.

### 4.3.1 Locating a candidate task chain to move

#### A. Identifying the head of a candidate movable task chain

Let $P_i$ be the processor that $T$ is assigned to. For $P_i$, the tasks in that processor are ordered, since we have a previous schedule which specifying an ordering.

The algorithm starts from $T$, proceeds along the task ordering for a constant number of steps $s$, which should be a small integer and can be adjusted, until either it finds a task $T_h$ such that $T_h$ is not a immediate successor of the task just before it in $P_i$, or we have proceeded for $s$ steps. (A third case is that we hit the last task assigned to processor $P_i$, but the real important issue is whether task $T_h$ is found.)

Since $T_h$ is not a successor of the task ordered before it, it is not in the same dependence chain as $T$, and can be chosen as the head of the candidate chain. A more formal description of this procedure is shown below :

Let $next(T)$ be the task that is assigned to the same processor as $T$ and immediately follows $T$ in the task ordering provided by the previous schedule.

$T_i = T$, $T_{i+1} = next(T)$,

step=0

while ( $step < s$ and $T_h$ not yet found)

step=step+1; if there is no edge in $G$ from $T_i$ to $T_{i+1}$

then $T_h = T_{i+1}$, $T_h$ is found

else $T_i = T_{i+1}$, $T_{i+1} = next(T_{i+1})$

If the algorithm fails to find such a task $T_h$ in $s$ steps, it will abandon the attempt of readjustment around task $T$ and move on to the next task in $V_{inc}$. Such a failure indicates that $T$ is in the middle of a relative long task chain. We could attempt to scan the ordering downward even further, but that will increase the complexity of the algorithm, which is a major problem we want to avoid. So we limit the number of steps to a small constant.

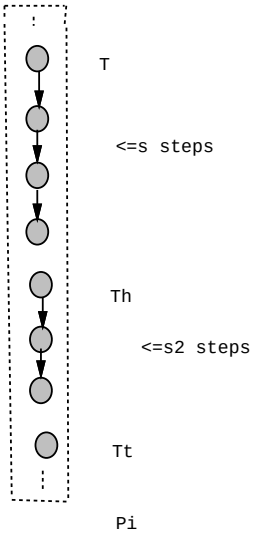
If otherwise we succeeded in finding $T_h$, the algorithm continues to the next stage.

**B. Searching for the tail of the candidate task chain**

Having found the head $T_h$, the algorithm simply continue scanning the task ordering in $P_i$ after $T_h$ in the same manner as the last stage. We will move for only a constant number of steps, again to control the complexity of the algorithm. In the procedure we will try to locate the first task $T_t$ such that $T_t$ is not a successor of $previous(T_t)$. ( By previous we mean the ordering in $P_i$.) It's obvious that the candidate chain ends at $previous(T_t)$. Since this procedure is similar in spirit to the one used to find $T_h$, we will not describe it in more detail.

Failure to locate $T_t$ in $s$ steps implies that the chain starting from $T_h$ is excessively long, and the algorithm will not attempt to move such long chains because we want to localize the execution. On the other hand if $T_t$ is found, we have extracted a candidate chain. Let there be $k+1$ tasks in the chain ($k+1 < s$), namely $T_h$,$T_{h+1}$.... $T_{h+k}$, and $T_{h+i+1} = next(T_{h+i})$, $T_{h+k} = previous(T_t)$. See figure 4.3 for a illustration of the

procedure.




Finding the candidate task to move. Assume that the weight of task T has a large increase. We can not move any task within the dependency chain of T, so we go down the ordering for s steps , trying to find the first task Th that is not in the chain. i.e. not a immediate successor of the task before it. Then we try to find where the Th chain ends, by going dowm for S2 steps till we find the first task Tt that is out of the chain. The candadidate to move is the Th chain. If either of the two chains are too long, we don't attempt to make adjustment for task T.

Figure 4.3: Finding the candidate task chain for moving

### 4.3.2 Conditions for moving the candidate chain

Having identified the $T_h$ chain, we still have to check for certain condition to make sure that the problem in example 2 will not occur, i.e. no excessive communication will be introduced if $T_h$ chain is ejected from this processor to reduce its load. The following two kinds of tests are conducted for this purpose.

#### The predecessor test

One basic criterion is that the ejection will not increase the start time of any task in the $T_h$ chain. Let $T_{h+c}$ be a task in the chain, assuming $T_{h+c}$ has $m$ immediate predecessors in the task graph that is assigned to processor $P_i$ and not in the $T_h$ chain. Those predecessors not in $P_i$ are irrelevant since no new communication cost will be

incurred by the ejection of $T_h$. Same situation for those within the chain. Call these tasks $T_{p1}, ...T_{pm}$. It's obvious that all these tasks are ordered before $T_h$ in the valid schedule. Let edge $e_{pj}$, with cost $C_{pj}$, be the communication from $T_{pj}$ to $T_{h+c}$.

We will initiate another scan of the task ordering, this time upward from $T_h$, again for only a constant number of steps. Let the ordering in $P_i$ be of the form:

$...., T_{k1}, T_{k2}....T_{ks}, T_h, T_{h+1}., T_{h+c}, ..$ (Note that $T$ is among $T_{ki}$ but this is not important.)

Now, for each of the $m$ immediate predecessors $T_{pi}$ of $T_{h+c}$, we have two different cases:

- $\exists j,\ s.t.\ T_{kj} = T_{pi}$.

  Now there is an edge from $T_{kj}$ to $T_{h+c}$ costing $C_{pi}$. If the sum of the task weights in this processor ordered after $T_{kj}$ and before $T_{h+c}$ exceeds $C_{pi}$, the start time of $T_{h+c}$ will not increase due to this particular edge if $T_{h+c}$ is moving out of the processor. i.e. The condition to check is:

  $W_{k,i+1} + ... + W_{k,s} + W_h + W_{h+1} + ...W_{h+c-1} \geq C_{pi}$

- $\forall j, T_{kj} \neq T_{pi}$

  Now $T_{pi}$ is ordered further above $T_{k1}$. We want to avoid higher complexity by not proceeding beyond $T_{k1}$, so we make a conservative estimation: If the sum of the weights of tasks from $T_{k1}$ to the task before $T_{h+c}$ exceeds $C_{pi}$ then the check on edge $e_{pi}$ is passed, since the total weights of tasks between $T_{pi}$ and $T_{h+c}$ in this processor is at least this value. So the condition is:

  $W_{k1} + .... + W_{k,s} + W_h + W_{h+1} + W_{h+c-1} \geq C_{pi}$

This check is conducted for all tasks in the $T_h$ chain. If any incoming edge of any of the tasks fails the test, we are in danger of negatively affecting performance so we abandon the attempt of moving $T_h$ chain. See figure 4.4.

**The successor test**

We also seek to ensure that the successors of the tasks in $T_h$ chain will not see any increase in their start time due to the removal of $T_h$ chain. For the same reason as

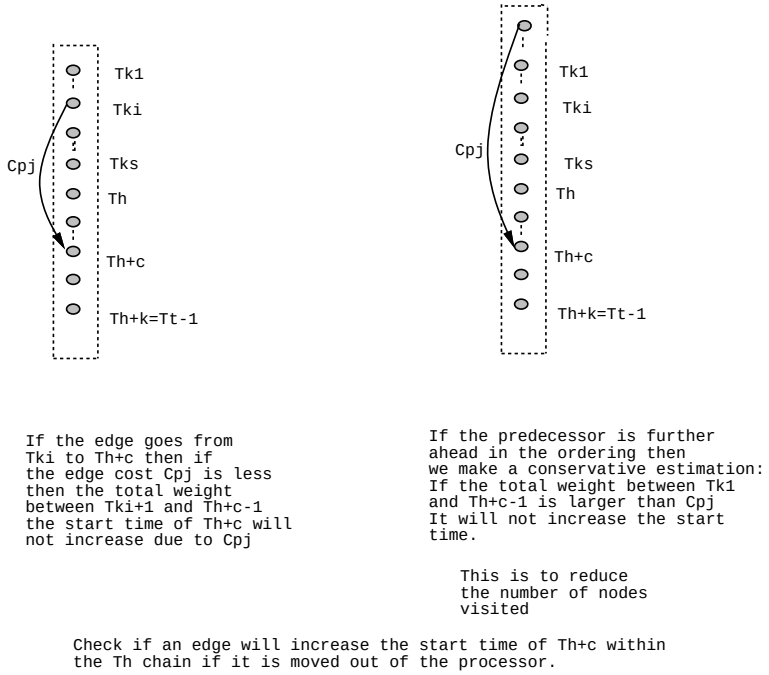



Figure 4.4: Check the predecessors of a task in the $T_h$ chain

the predecessor test, only those successors assigned to this processor, $P_i$, and not in $T_h$ chain need to be checked.

For any task $T_{h+c}$ in $T_h$ chain with $l$ immediate successors assigned to $P_i$ and not in $T_h$ chain, call these successors $T_{s1}....T_{sl}$, and assume the edges have costs $C_{s1}, ....C_{sl}$. We start from the first task in $P_i$ after $T_h$ chain, i.e. $T_t$ and scan downward for a constant number of steps. Assuming the ordering is of the form:

$..T_{h+c}, ...T_{t-1}, T_t, T_{t1}, ...T_{ts}.....,$

Where $s$ is the predefined constant.

Similar to the predecessor test, for an edge from $T_{h+c}$ to $T_{si}$ we have two cases:

- $\exists j \ s.t. \ T_{tj} = T_{si}$.

  Given that the start time of any task in $T_h$ chain will not increase, a condition ensured by the predecessor test, we can see that if the total task weights between $T_{h+c}$ and $T_{tj}$ exceeds the edge cost $C_{si}$, the start time of $T_{si}$ will not increase. So the condition is:

  $W_{h+c+1} + ... + W_{t-1} + W_t + ... + W_{t,j-1} \geq C_{si}$

- $\forall j,\ T_{tj} \neq T_{si}$.

  We again make a conservative estimation since we don't want to scan for more than $s$ steps, by using a lower bound of the total weight between $T_{h+c}$ and $T_{si}$, which is the total weight up to task $T_{ts}$. The condition becomes:

  $W_{h+c+1} + ... + W_{t-1} + W_t + ... + W_{t+s} \geq C_{si}$.

An illustration is in figure 4.5.

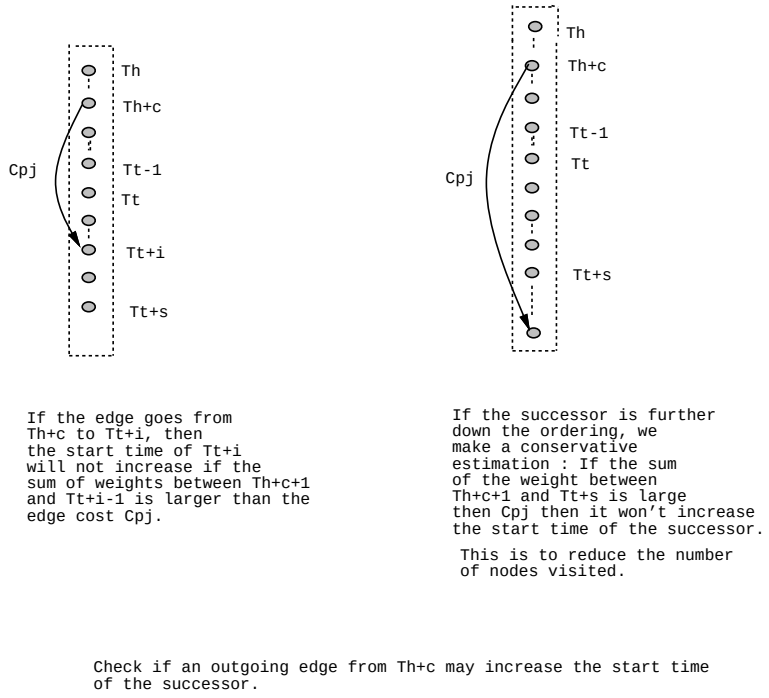



Figure 4.5: Checking the successors of a task in the Th chain

### 4.3.3 Movement of the $T_h$ chain

If the above tests are passed, it can be beneficial to move $T_h$ chain out of the processor $P_i$, as the previous tests have ensured that this load balancing step is "safe", i.e. no adverse effect on the start time of any tasks. This step will hopefully lead to a reduction of parallel time. If we repeat this process to all the "critical" tasks, i.e. those with large weight increase, we will likely to achieve performance gains.

We still have to decide where the tasks in the ejected $T_h$ chain should be assigned. If this is an unbounded number of processors, we simply assign the chain into a new

processor, or cluster. If we compare what we did with the procedure used in the DSC clustering algorithm described in [59], we shall see that we are indeed carrying out cluster linearization: The original DSC scheduling step has created non-linearized processor assignment by putting $T$ and the $T_h$ chain in the same processor. If the weight increase of $T$ is significant, we are attempting to separate them and create a new linear cluster for $T_h$ chain.

On the other hand, for fixed number of processors, as in most realistic cases, we need to merge this new cluster into another processor. There is no clear guideline of the merge process, so we adopt a simple load balancing method by choosing the current processor with the lightest load. This is similar to the cluster merge heuristic employed in the PYRROS system ([60]). We call this processor $P_j$.

Note that in order for $T_h$ to be moved, the following load criterion must also be satisfied:

$L_i - L_j > |(L_i - W_h) - (L_j + W_h)|$

Where $L_i$ and $L_j$ are the loads of $P_i$ and $P_j$ and $W_c$ is the sum of weights of all the tasks in $T_h$ chain. This condition ensures that the imbalance of the load between $P_i$ and $P_j$ will be indeed reduced.

Finally, if all the above conditions are satisfied, we can place the task in $T_h$ chain in $P_j$. Our heuristic is to insert each task into $P_j$ by starting it as early as possible, i.e. order it as early as possible after its predecessors in $P_j$. This is of course not the only way to accomplish the goal, but we have decided to adopt this simple method.

After the adjustment, we see that the load of $P_i$ is reduced and balanced with a light weighted processor $P_j$, just as in the simple load balancing algorithm. On the other hand, we have detected long dependence chains and avoided moving them across processors. We also detect high communication edges and avoid large increase in communication cost. Therefore the adjustment is likely to be a beneficial one. Due to the complexity of the problem, and the constraint that we can not afford critical path calculation, we can not give definite, theoretical assertion about the performance improvement. But our algorithm is certainly superior to that of the naive load balancing without regard to communication edges, and we shall see that this algorithm performs

well for random task graphs.

The complexity of the algorithm, excluding the global sampling phase which has $O(v)$ complexity, is $O(n_c * d)$, where $n_c$ is the number of tasks with large weight increases, and $d$ is the average degree of a task. This is because for each task with large weight increase we will only scan a constant number of tasks in the same processor, and the number of edge examined is at most the total number of incoming and outgoing edges from these tasks. For most task graphs from practical applications, such as the FMM algorithm, the degree of a node is bounded, making the complexity as low as $O(n_c)$.

A nice property of the algorithm is that it's localized. It only focuses on those tasks with large weight increases. For most cases $n_c << n$, rendering the complexity only a fraction of that of the global scheduling. It can also be see that the algorithm reduces to simple load balancing if no dependence are present in the task graph, since all the predecessor and successor tests will be skipped and the adjustment is based only on the workload of each processor.

### 4.3.4 Test results on random task graphs.

We generate 20 random tasks graphs that are either coarse grain or mixed grain (meaning that $g(G)$ is about 1. See chapter 2 for definition of $g(G)$). For each tasks graph we schedule it first on 2 to 64 processors. Then we give large task weights increases to some randomly chosen tasks in the task graph. Next we record the two different parallel times:

1. PT(PYRROS), which is the parallel time obtained by discarding the old schedule and using PYRROS from scratch on the new task graphs.

2. PT(reschedule), which is the scheduling length achieved by using the fast reschedule heuristic that does local adjustments.

We define $DIFF = \frac{PT(reschedule) - PT(PYRROS)}{PT(PYRROS)}$, which shows how far the schedule length obtained by fast reschedule differs from that of applying PYRROS from scratch. Since PYRROS may not find the optimal schedule, this quantity could be negative, which indicates that reschedule is even better than PYRROS from scratch.

We tested these 20 task graphs and recorded the $DIFF$ value for different numbers of processors, and found the minimum, maximum, average and median values of $DIFF$ for different number of $p$. The result are listed in table 4.1:

The running time of fast reschedule approach is always much less than that of PYRROS for all cases, and the ratio of running time is about the ratio of $n_c$ to $n$, as expected. This ratio is set to be about 1/20 to 1/10 for these cases. The constant $s$ in the algorithm is chosen to be 5.

To get stronger evidence, we allow no only one rescheduling step but multiple steps, which means that for multiple times the task graphs are perturbed and the two methods are applied, and we compare the end result after a number of perturbation steps. If the fast reschedule algorithm is inherently inferior to global reschedule, applying the algorithms for multiple steps will enlarge the performance difference between the two. Nevertheless, in the tests we allowed 5 such rescheduling steps and the parallel time at the end is still very close to each other on the average. See table 4.1.

| # Proc | Min | Max | Average | Median |
|---|---|---|---|---|
| 2 | -7.2% | 5.9% | -2.0% | -2.1% |
| 4 | -8.7% | 14.5% | -1.8% | -3.6% |
| 8 | -14.1% | 10.6% | -1.0% | -0.7% |
| 16 | -13.8% | 13.3% | 0.1% | 0.1% |
| 32 | -10.8% | 13.3% | 1.5% | 1.6% |
| 64 | -2.5% | 8.8% | 3.9% | 3.7% |

Table 4.1: The difference between local rescheduling and PYRROS from scratch

This shows that the local readjustment heuristic, with much lower complexity, performs almost as good as the global scheduling method on average.

## 4.4 Incremental Scheduling for Task Graph Spawning

In this section, we discuss another class of dynamic problems where the task graph structure is altered. The changes, however, are not arbitrary but behave as task graph spawning, where new subgraphs are created and attached to the original one. Examples of such class of problems include the FMM N-body algorithm with repartitioning, during

which certain boxes can be split and new subtrees grown.

A naive way to deal with dynamic spawning is to reschedule the entire task graph from scratch. However this can be too expensive since the old schedule derived from the old task graph is not utilized, especially when the new subgraph is small relative to the original graph.

We present here an algorithm that is incremental in nature. The idea is that we first cluster the newly spawned subgraphs, then merge the new clusters into the original schedule in an appropriate fashion. Such approach obviously has lower cost, since it only deals with the new subgraphs, and we will demonstrate that the algorithm derives competitive schedule to that produced by global rescheduling. The result is especially good for coarse grain graphs: For an unbounded number of processors, our algorithm will result in a parallel time no larger than that obtained by rescheduling the whole task graph from scratch. Moreover, if the local clustering algorithm is DSC, then for coarse grain trees that dynamically spawn in any fashion, the incrementally generated clustering will always be optimal at any stage during the tree growing process.

### 4.4.1 Problem definition and clarification

Before discussing our method we have to clarify how a task graph "spawns" new component. Let $G_s$ be a subgraph that consists of all the new node after the "spawning", we first give the following restrictions:

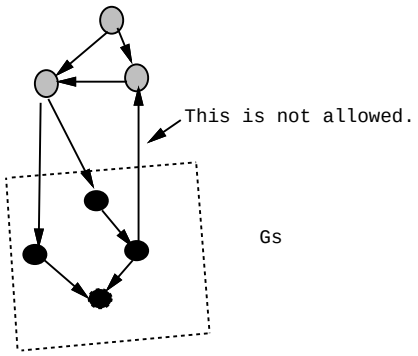
- No node in $G_s$ can be a predecessor of any node in the original graph $G$.

  This is a natural restriction, since if a newly created node is allowed to be a predecessor of any old node, then the graph may have cycles which violates the acyclic requirement. Even if there is no cycle, the critical path of the original graph can be completely skewed, which is undesirable and can lead to global changes.

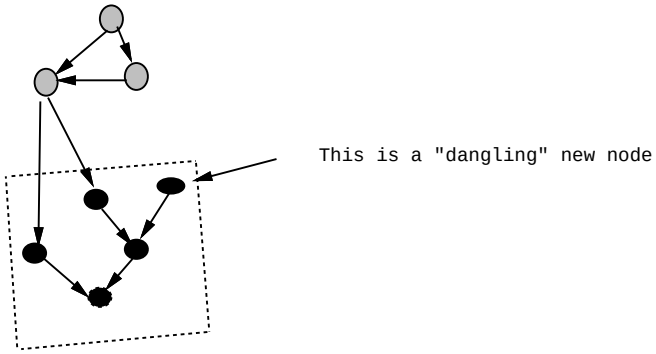
- The second restriction is that there is a special node $n_0$ which is an ancestor to all the nodes in the newly spawned $G_s$. We call this node the "spawning node" since intuitively the subgraph $G_s$ can be viewed as a new part invoked by node $n_0$.

This is also a natural restriction, since without loss of generality, if there exists a "dangling" new node which is not a descendent to $n_0$, we may add an zero-weight edge from $n_0$ to it.

See figure 4.6.




Restriction 1: There should not be edges going backward from the newly spawned graph to the original graph.




Restriction 2: There should be no "dangling" new node, i.e. node in Gs that is not a successor of any node in G.

Figure 4.6: Two restrictions on graph spawning new components

Note that for certain special cases, such as an in-tree spawning new leaves in a reverse order, the situation is completely the mirror image of what we discussed above. In such cases, scheduling can be done by reversing the edges and reducing the graph to an out-tree, which satisfy the above restriction. This will be useful later when we deal with the FMM task graph.

Now we define a subclass of spawning which we can handle using a local incremental scheduling heuristic:

**Tree spawning:** A spawning within a task graph is called tree spawning if the newly spawned subgraph has only edge links to the spawning node $n_0$ in the original graph. Note that the old graph and the newly spawned subgraph need not be trees, as long as only $n_0$ has edges to $G_s$. See figure 4.7.

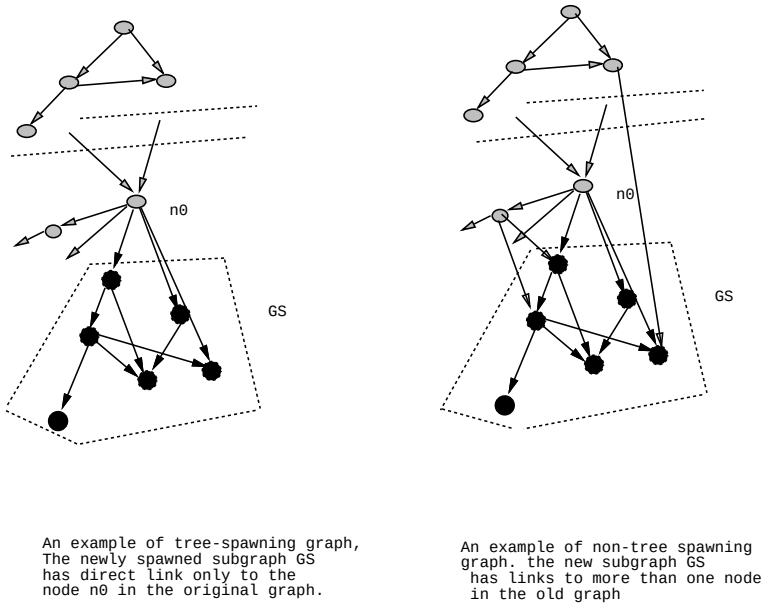



Figure 4.7: examples of a tree spawning graph and a non-tree-spawning graph.

The motivation behind the tree spawning definition is that for such graphs, local clustering of the newly created subgraph can be isolated from the rest of the graph. Since even if a global clustering method is used, the fact that the subgraph is linked to the original graph via only $n_0$ insures that only local clusters are produced on the newly spawned part. In particular, if the global clustering is linear, then the clustering process of the new part is totally isolated, thus a local clustering will result in a schedule as good as scheduling from scratch. This is the case for DSC on coarse grain graphs.

It should be noted that spawning in some problems are not simple tree spawning. The newly created subgraph can have links to multiple old nodes. For instance, in the FMM N-body computation, when a box subdivides, the new smaller boxes not only interact with the divided parent box but other boxes as well, such as those in the

interaction list. Thus there are many links from the old N-body graph to the newly created tasks. For such spawning, it is difficult to prove that the local clustering retains the optimal property. Nevertheless, in such cases the heuristic can still be used. One node may be designated as the spawning root, using certain guidelines. For example, it's easy to see that we will have a nearly tree-spawning graph if the split node in the downward pass, which is the root of the new subtree formed by its subboxes, are chosen as the spawning root. Since all the other edges are coming from the upward pass, and is unlikely, although not impossible, to affect the start time of the new subgraph. The "local critical path", therefore, can be assumed to pass through the spawning root. In general, we should pick those nodes in such a local critical path, and the resulting task graph can be approximated by a tree-spawning graph. Therefore we will focus on tree-spawning graphs for our theoretical analysis.

### 4.4.2 Description of the algorithm

Now we present the local clustering algorithm for tree spawning graphs: Assume $n_0$ is the spawning node, and $G_s$ is the newly spawned subgraph, including $n_0$ as the source node of the graph. Also assume that the original task graph has been clustered and $n_0$ was assigned to cluster $C_i$. Moreover, in each cluster the tasks has been ordered and the start time of each tasks is known, as well as the total parallel time $PT$ of the original schedule. This is a reasonable assumption since all these information is obtainable from the initial scheduling algorithm, so they don't need to be recomputed. For each task node $n_t$, we have:

$Bottomlevel(n_t) = PT - Starttime(n_t)$.

#### Step 1: Generation of local clusters

Using a predetermined clustering algorithm, such as the DSC heuristic, the algorithm first generates local clusters on the subgraph $G_s$.

Let $n_0$ be contained in local cluster $c_1$, we next merge the local and global clusters by accommodating $n_0$ in a proper way. For the local clusters other than $c_1$, they can

be simply added to the global cluster list since they don't contain any old node.

We assume that the informations about bottomlevels of the tasks in $G_s$ are available as a by-product of the clustering and ordering, just as the original global clustering.

**Step 2: Determine the new clustering around node $n_0$**

**Definition:**

*Local Path Length* is defined to be the longest path starting from $n_0$, with the first edge is directed from $n_0$ to a node either in $C_i$ or $c_1$, to the sink node(s) of the task graph (the whole graph including the new component $G_s$).

The motivation behind this definition is as follows: We must decide how to merge clusters $C_i$ and $c_1$, and the merge can affect the bottom level of node $n_0$, and therefore possibly, although not definitely, the global parallel time. More importantly, if the rest of the graph is fixed, minimization of the *Local Path Length* implies the best possible reduction of global parallel time.

So we narrow down the problem to the minimization of the bottom level of $n_0$. It can been seen that if a path from $n_0$ to the sink has its first edge going into $C_i$ or $c_1$, its length can be affected, Otherwise it will not. So this *Local Path Length* is a variable we need to minimize.

There are 3 ways of merging $C_i$ and $c_1$. For each of them we try to find the *Local Path Length*. We will later choose the one that minimizes this value.

- **Method A**:

  Merge $c_1$ and $C_i$ into one bigger cluster, by putting all the tasks in $c_1$ (except $n_0$) behind those in $C_i$.

  For this case, let $n_1$ be the task in $C_i$ that is immediately ordered after $n_0$, and $n_2$ be the task in $c_1$ immediately ordered after $n_0$. After the merge, *Local Path Length* can be seen to be:

  $$LPL = max(Bottomlevel(n_1), Bottomlevel(n_2) + W_0)$$

where $W_0$ is the total task weight of all the tasks in $C_i$ ordered behind $n_0$.

See figure 4.8 for illustration of Method A.

- **Method B**:

  Keep $n_0$ in its old cluster $C_i$, and extract $n_0$ from $c_1$.

  For this case, let there be $m$ edges from $n_0$ to nodes in cluster $c_1$, call them $e_j$, with costs $comm_j$, and the end nodes $n'_j$, $j = 1...m$. Also let $n_1$ be the task ordered immediately after $n_0$ in cluster $C_i$. Now the *Local Path Length* has the following form:

  $$LPL = max(Bottomlevel(n_1), Bottomlevel(n'_1) + comm_1,$$
  $$...Bottomlevel(n'_m) + comm_m)$$

- **Method C**:

  Split cluster $C_i$ into two clusters $C_{i0}$ and $C_{i1}$ at task $n_0$, by putting all tasks ordered after $n_0$ into cluster $C_{i1}$. Then merge cluster $C_{i0}$, which contains $n_0$, with the local cluster $c_1$, by ordering all task in $c_1$ after $n_0$.

  The value of *Local Path Length* may be deduced in a way similar to Method B. let there be $m$ edges $e_1, e_2...e_m$ with costs $comm_1, comm_2....comm_m$ going from $n_0$ to tasks in $C_{i1}$, and edge $comm_j$ is adjacent to node $n'_j$. Also let $n_2$ be the task in $c_1$ ordered immediately after $n_0$. Then we have:

  $$LPL = max(Bottomlevel(n_1), Bottomlevel(n'_1) + comm_1,$$
  $$...Bottomlevel(n'_m) + comm_m)$$

  See figure 4.9 for cases B and C.

We examine all three choices, selecting the one that minimizes the LPL value. This is the new clustering obtained by the algorithm. This minimization will ensure that the global parallel time will be less than or equal to the parallel time obtained by using

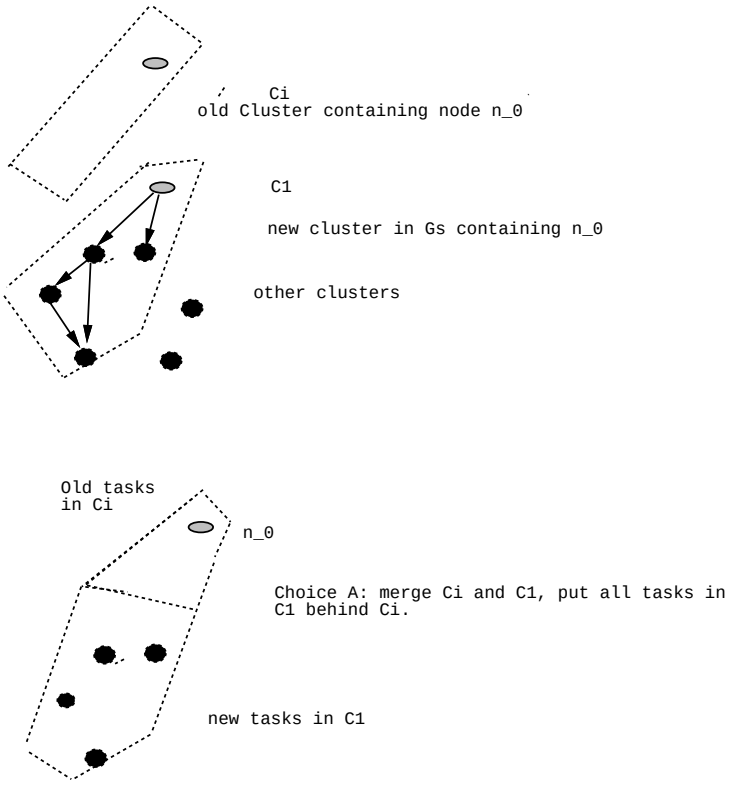
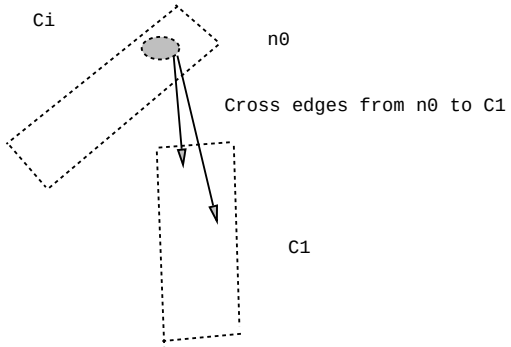
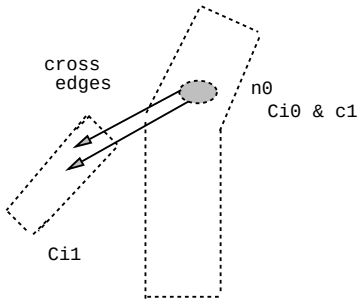



Figure 4.8: case A of handling the new cluster




Case b: Keep n0 in the old cluster.




Case c: split cluster ci at node n0, merge ci0 with new cluster c1.

Figure 4.9: case B and C of handling the new cluster

the other merge methods. Note that the a spawning leads to incremental adjustment of the clusters, not clustering from scratch.

After merging clusters $C_1$ and $C_i$, the bottom level of $n_0$ and nodes in $C_1$ and $C_j$ can be updated if needed, in order to be used for the next spawning of task graph.

**Step 3. Merging clusters into fixed number of processors.**

In case that we have a fixed number of processor, we have to do cluster merge. We may assume that the number of old cluster is equal to the number of processors available, if not we can simply allocate a new processor for a new cluster until there is no processor left. So what we need to do is simply absorbing the newly created clusters into the old clusters.

We use the following simple heuristic: Suppose we have $k$ new clusters, we just merge each new cluster with an old cluster that has the lightest weight, putting the new tasks after the tasks in the old cluster, and update the weight of that cluster. We repeat this merge process until all new clusters has been accommodated. This has the same underlying mechanism as the cluster merge process in PYRROS. See [60].

Note that task ordering step in PYRROS can be skipped, since it's clearly we may put those tasks in the new clusters after the old ones, without violating any constraint. (Task ordering in the new clusters has already been established by the clustering step on the subgraph.) Although such an ordering may not be the same as the one obtained by a global reordering step, it has minimal complexity, and keeps the whole algorithm completely localized, which is a desirable property.

### 4.4.3 Analysis of the incremental scheduling heuristic on tree spawning graphs.

The above heuristic has the property of localized cluster formation of the newly spawned tasks, since this is a natural way to reduce the complexity of the incremental step. It's easy to see that in general this method may not yield the same clustering obtained by discarding the old clustering and applying the clustering algorithm on the whole

graph, since global clustering may produce clusters across the old and new parts of the graph. However, it can be argued that it's not clear that the global clustering will yield better result than the incremental localized clustering, except perhaps for very fine grain graphs, where localized clustering may generate too many clusters. (But then a cluster merging step should help reduce the number of clusters.) Even if the global clustering yields better parallel time, the cost of clustering is much higher than the localized heuristic.

In addition, if the clustering algorithm produces only linear clusters for certain classes of graphs, (e.g. The DSC algorithm always creates linear clusters for coarse grain graphs.) then the localized clustering will yield a parallel time no more than the global clustering: Since if the clustering is linear, even if global clustering is used, none of the tasks in the newly spawned $G_s$ could be clustered with any tasks in $G$ except the spawning node $n_0$, due to the fact that $G_s$ is connected to the rest of $G$ via $n_0$ only. So if a cluster in $G_s$ is created by the global clustering algorithm it must also be produced by the local clustering on $G_s$, assuming the same clustering heuristic is used throughout, and the stopping criteria of the heuristic is monotonic decreasing of parallel time, i.e the clustering stops when the parallel time becomes irreducible by the attempted clustering step. This, in effect, causes the global clustering to be completely localized in the subgraph $G_s$. So we have the following result:

**Proposition**

Assuming the number of processor is unbounded, If it's known that only linear clusters are obtained on a class of graphs by a heuristic that always attempts to achieve monotonic decrease of parallel time, then the parallel time obtained by our incremental algorithm will be no more than that obtained by global reclustering.

For details of the stopping criteria of monotonic decrease of parallel time, the reader may refer to Tao Yang's thesis ([59]).

It should be noted that the local clustering may examine more nodes in $G_s$, since the global clustering may stop before all the tasks in $G_s$ are considered. Nevertheless,

the former will never have larger parallel time than the latter.

In particular, the above proposition implies that if DSC algorithm is used, the local clustering heuristic will always yield optimal clustering for coarse grain trees that grows from the root and dynamically spawning new branches, no matter what the order is, at any moment of the tree growth.

For non-tree spawning graph, we will not be able to derive such a definite conclusion, since in these graphs we can not be sure how the global clustering will behave, while in tree-spawning case, we know that the global clustering step will enter the subgraph through the spawning root $n_0$ and does the same as local clustering would do on the same subgraph before terminating. However, if the graph closely approximates the tree spawning case, we can expect similar experimental result.

Even if the graph is not strictly coarse-grain, we shall see that on average, our incremental algorithm is comparable to global clustering in experimental performance.

### 4.4.4 Experimental results on random task graphs

Similar to the previous rescheduling algorithm, we generate 20 random graphs that are coarse or mixed grain. In each test case, a random node are chosen as the spawning root, and a subgraph, which is also non-fine grain and not necessarily a tree, is spawned from the root. To deal with such spawning we can either do global scheduling, or clustering the new subgraph and merge it with the whole graph. We can show that even after repeated spawning (say 10 times), the final result obtained by repeating incremental clustering is still near that of global scheduling.

Again we define $DIFF = \frac{PT(incremental) - PT(PYRROS)}{PT(PYRROS)}$ which shows how far the parallel time obtained by fast reschedule is from that of PYRROS from scratch. Since PYRROS may not find the optimal schedule, This quantity could be negative, which indicates that reschedule is even better than PYRROS from scratch.

We tested the 20 problem and recorded the $DIFF$ value for different numbers of processors. Each test case can have up to 10 times of spawning. We found the min, max, average and median values of $DIFF$ for different number of $p$. The result is listed in the table 4.2:

| # Proc | Min | Max | Average | Median |
|---|---|---|---|---|
| 2 | -5.0% | 1.4% | -0.2% | 0% |
| 4 | -5.3% | 8.4% | 0.5% | 0% |
| 8 | -11.9% | 6.4% | -1.3% | -1.0% |
| 16 | -9.7% | 8.0% | -0.1% | -0.4% |
| 32 | -6.9% | 9.3% | 1.1% | 0.5% |
| 64 | -7.8% | 10.9% | 0.6% | 0% |

Table 4.2: The difference between incremental method and PYRROS from scratch

It can be seen that the local clustering is as good as global clustering for such spawning problems. So for random graphs it could replace the global method.

So far, our discussion about the two dynamic support heuristics focuses only on algorithm description and random graphs. In the next chapter, the real task graph in adaptive FMM algorithm is studied and the schedule readjustment heuristics are applied in a real world setting.

# Chapter 5

# Dynamic Support for Fast Multipole N-body Algorithm

## 5.1 General Framework of Our Dynamic Support Approach

We have demonstrated the applicability of compile-time scheduling to the FMM computation over a considerable number of iterations. Yet our study of the N-body problem must go further to the issue of rescheduling support, since sooner or later the slow down in parallel time will become substantial. Let's first discuss this issue in a generalized framework. From now on when we talk about the FMM algorithm, we are referring to the adaptive version, since the non-adaptive version is of limited practical use and can be considered a special case of the adaptive version.

Consider the status of the execution of the FMM algorithm after iteration $i$. There are 4 choices that can be made regarding the iterative execution.

1. Reuse the same schedule for iteration $i + 1$. This was our strategy for the experiments in Chapter 3. Its advantages are the simplicity and its overhead-free property, and it is the clear choice for the initial iterations to amortize the compile time scheduling cost. But such recycling of schedule will reach its limitations after certain number of iterations, and this number is problem-dependent.

2. Repartition the boxes and schedule the new task graph from scratch. This amounts to disregarding all the previous task graph information. While this approach can produce better parallel time, the overhead is at the same level as that of the initial compile-time scheduling.

3. Retain the the task graph structure, but gather the information of the new task and edge weights, and apply a dynamic balancing heuristic to reduce the parallel

time. This approach will hopefully reduce the parallel time and the overhead is small, if a reasonable heuristic is employed.

4. Let the boxes with excessive number of particles subdivide, resulting in task graph spawning, then apply an incremental scheduling algorithm to handle the new task graph. This approach can take care of particle concentrations and achieve parallel time reduction at low cost. Note that at the same time some empty boxes may be prune from the leaves.

Approaches 2, 3, 4 are all aimed at correcting iterative performance loss. It's obvious that approaches 3, 4 have much smaller overhead than approach 2. The major issue here is whether they offer performance improvements that are competitive to that achieved by scheduling from scratch. Since PYRROS scheduling itself is based on heuristics and not optimized, one can expect that certain local adjustment heuristics may perform comparably well.

In the last section, we presented two fast rescheduling heuristics designed for this purpose, with the local schedule readjusting algorithm for approach 3 and the incremental rescheduling algorithm for approach 4. Next we will discuss the issues in implementing them for the FMM algorithm, and some performance test results.

## 5.2 Issues in Implementing Dynamic Support for the FMM Algorithm

Now we use the FMM algorithm as a testbed for our rescheduling framework. We monitor the iterative execution of some typical examples, and invoke the reschedule system when a need arises. What we want to demonstrate is that rescheduling methods are indeed applicable and the performance result agrees with the test results on random graphs.

First we have to point out some caveats that is inherent for such applications: We know that the parallel time increase is due to particle movements. On the other hand, exactly how the particles move is not within the scope of computer science research, rather it's governed by law of physics and indeed it's a challenging physics problem when the system contains many particles. So we are not trying to "predict" the particle

movements or to provide in-depth physical explanation to the changes in parallel time. We instead focus on the system support part.

When the fast rescheduling heuristics are applied, there are some additional issues that have to be addressed, and also some features in the FMM algorithm that we may take advantage of:

- For the local schedule readjustment algorithm used in approach 3, the nature of the task graph structure is helpful to us, since the weight fluctuations induced by particle drifting only occur at the leaf level, and the weights of non-leaf nodes depend not on the number of particles but the length of the expansion terms. Therefore it is easier to find the candidate tasks to move, since they are mostly among the leaves in the downward pass where the heaviest amount of computation takes place.

  Moreover, at each adjustment step, only one task need to be moved, since there is no long dependence chain among the leaves. Indeed there is no edge between any pair of leaves that are both in the downward pass. Finally, The fact that leaves in the downward pass have no successor saves more effort since the corresponding tests may be skipped.

- For the incremental scheduling algorithm used in approach 4, Some minor modifications are needed:

  1.) Due to the upward-downward concatenation property of the task graph, when a box subdivides, the spawning of the graph occurs not only at the leaf task in the downward pass, but also symmetrically at the leaf node in the upward pass where multipole expansions are generated. The original incremental scheduling algorithm will only handle the spawning of the downward pass leaf nodes since it assumes that the newly expanded nodes are successors of the spawning node. Nevertheless, the upward pass spawning can be easily taken care of by the same algorithm in the reverse, or mirror image, fashion.

  2.) Another issue to be taken into consideration is the cluster merge and task ordering after incremental generation of new clusters. The newly created tasks

at the leaves in the downward pass should be ordered after all the old task in the same cluster, and symmetrically, all the newly spawned leaves in the upward pass should be executed first. So we can simply merge these new cluster into the old $p$ clusters in a way that attempts to balance the total weight of each cluster, and then order the new tasks without affecting the old ordering of existing tasks. The complexity of this procedure is clearly smaller than the usually merge and ordering method since that method involves global task reordering, which is not really necessary in this task graph given the order of the original tasks.

- There is one important point need to be addressed: When we use the task graph spawning and incremental clustering method, i.e. approach 4, we modify the task graph by splitting certain leaf boxes. This will affect the task graph edges adjacent to such boxes and create new edges linked to the new descendent boxes. At the same time, the various interaction lists, i.e. U, V, W, X lists related to the split boxes and the descendent have to be modified. In fact the readjustment of the interaction lists is the basis of edge update. We could repeat the interaction list scanning process used for initial task graph generation, but this is clearly unnecessary. Instead we should update the lists incrementally, with very small cost.

  We present an algorithm to adjust interaction lists in figure 5.1. Without loss of generality, we will only consider the case where a leaf box $b$ is split into 4 children $c_1$ through $c_4$. In some cases the particles in the box may be so highly condensed that a spawning step may create a multi-level subtree, but the list update can be carried out incrementally by growing descendent boxes one level at a time.

  The rules of update are rather involved, but they ensure that we are only dealing with related boxes and not any other boxes in the task graph. The reader should refer to 3.3 for the definition of the lists, and a picture will be helpful for understanding the algorithm.

The complexity of the above algorithm is linear in the size of the interaction lists of $b$, which is really the lowest possible.

**BEGIN**
**For** i=1 to 4 do
Initialize $Ulist(c_i)$ to be its 3 siblings, and other lists of $c_i$ as empty.
**Endfor**
**For** each box $b_u \in Ulist(b)$ do
Remove $b$ from $Ulist(b_u)$.

/* Comment: Due to symmetry $b$ must be in $Ulist(b_u)$, but now $b$ is no longer a leaf so remove it */

**For** each $c_i$, $i = 1..4$ do,
if adjacent($b_u$, $c_i$) add $b_u$ to $Ulist(c_i)$ and add $c_i$ to $Ulist(b_u)$.
else begin
if (size($b_u$)==size($c_i$))
add $b_u$ to $Vlist(c_i)$ and add $c_i$ to $Vlist(b_u)$

/* Comment: now $b_u$ is a neighbor of $b$ but not $c_i$, and they are of the same size, so $b_u$ is interacting with $c_i$ through translation of expansions, i.e. Vlist */

if (size($b_u$)>size($c_i$)) add $b_u$ to $Xlist(c_i)$ and add $c_i$ to $Wlist(b_u)$.

/* Comment: in this case $b_u$ is bigger than $c_i$, and $c_i$'s parent is adjacent to $b_u$, so $c_i$ is in Wlist of $b_u$. Note for the 3rd case where $b_u$ is smaller than $c_i$ they are not in each other's lists, since $b_u$'s parent can't be adjacent to $c_i$. */

**end**
**Endfor**
**Endfor**
Set $Ulist(b)$=empty.
/* Comment: $b$ loses its Ulist since its no longer a leaf. the Vlist and Xlist of $b$ are not related to $c_i$ in anyway, and are not affected by the splitting */
**For** each $b_w \in Wlist(b)$ do
Remove $b$ from $Xlist(b_w)$

/* Comment: since W and X lists are mirror images, $b$ was in $Xlist(b_w)$ but it should be removed since $b$ is no longer a leaf. */

**For** each $c_i$ i=1..4 do
if (size($b_w$)==size($c_i$)) add $b_w$ to $Vlist(c_i)$ and $c_i$ to $Vlist(b_w)$
else if (adjacent(parent($b_w$), $c_i$)) add $b_w$ to $Wlist(c_i)$ and $c_i$ to $Xlist(b_w)$.

/* Comment, here $b_w$ can not be larger than $c_i$ since they are all smaller than $b$. */

**Endfor**
**Endfor**
Set $Wlist(b) = Empty$
**END**

Figure 5.1: The incremental interaction list update algorithm.

| iteration no. | 1 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| parallel time | 1.29 | 1.29 | 1.29 | 1.30 | 1.30 | 1.31 | 1.35 | 1.35 | 1.36 | 1.38 | 1.41 |
| PTincre | 0 | 0 | 0 | 0.8% | 0.8% | 1.6% | 4.7% | 4.7% | 5.4% | 7.0% | 9.3% |

Table 5.1: The first 100 iteration of test case 1

## 5.3 Rescheduling Performance of the Dynamic Support System

We present three typical test cases. Note that we are using the adaptive version of the FMM algorithm. For the rest of this chapter, the time unit in all the tables is second.

### 5.3.1 Test case 1

For the first test case, we use a relative small example of twin galaxies containing 2000 particles , running on 16 processors of the Ncube2s. We choose a rather large number of processors since it's known in previous discussion that the run-time imbalance generally grows with the number of processors, and also in a real production run the user will tend to use a large number of processors as long as the efficiency is maintained at a reasonable level. (In this particular case more than 16 processors will result in poor efficiency.)

We use PYRROS to schedule the initial task graph, and iteratively execute the task graph on 16 processors, We listed the parallel time of the first 100 iterations in table 5.1, in the interval of 10 iterations. In this table, PTincre stands for the increase in parallel time over the first iteration, and the unit of parallel time is second. Here, the timestep size $\delta t$ has to be chosen to be one order lower ($10^{-4}$) than that used in the uniform distribution in the last section, since in such twin galaxy distributions the particle density in some regions (such as galaxy centers) are very high and smaller time step resolution must be adopted to obtain any meaningful simulation result.

It could be observed that the parallel time increase is virtually nonexistent until after 50 iterations, and later the speed of performance loss starts to accelerate, but the overall parallel time increase at the 100th iteration is still not large. This result is consistent with other input distribution described in section 3.4.

| iteration no. | 110 | 120 | 130 | 140 | 150 |
|---|---|---|---|---|---|
| parallel time | 1.42 | 1.45 | 1.48 | 1.57 | 1.60 |
| PTincre | 10.1% | 12.4% | 14.7% | 21.7% | 24.0% |

Table 5.2: The 100th-150th iteration of test case 1

| Method | Approach 2 | Approach 3 | Approach 4 |
|---|---|---|---|
| parallel time | 1.46 | 1.49 | 1.42 |
| Cost | 1.0 | 0.09 | 0.20 |

Table 5.3: The parallel time reduction and cost for different reschedule method on test case 1

We continue the simulation to the 150th iteration, when we find the degeneration over limit, and rescheduling is called for. See table 5.2.

Now we use different methods to carry out the rescheduling, recording the scheduling cost in CPU time and the parallel time in the next iteration for each method applied. We can do one of the following: complete repartition and using PYRROS from scratch (approach 2), fast readjustment without repartition (approach 3), and task graph spawning and fast reschedule (approach 4). We compare the results in table 5.3

In this case all three methods result in similar parallel time reductions, with incremental clustering yields slightly better result than the other two, and the costs of the local re-balance and incremental schedule are both only a fraction of the cost of complete global reschedule. It can be argued that the user may choose between fast scheduling readjustment without repartition, or task graph spawning with fast reschedule, but complete reuse of PYRROS from scratch is obviously unnecessary.

Notice that even with rescheduling support, the parallel time can not be pushed back to the level of the initial iteration. This indicates a reduction of potential parallelism in the input distribution itself, rather than any problem with our scheduling system. Although we can not explain this phenomenon in a strict physical sense, we have observed a concentrating trend of particles into dense clusters, that is, those boxes with higher particle density tend to attract even more new particles. This will result in

an increase in the amount of irregularity of the input distribution. Our observation is consistent with the prevailing view of astrophysics concerning the formation of galaxies in real world particle interaction situations.

We depict the parallel time per iteration in figure 5.2. The final drop in parallel time reflects the result of applying task graph spawning heuristic, i.e. approach 4.

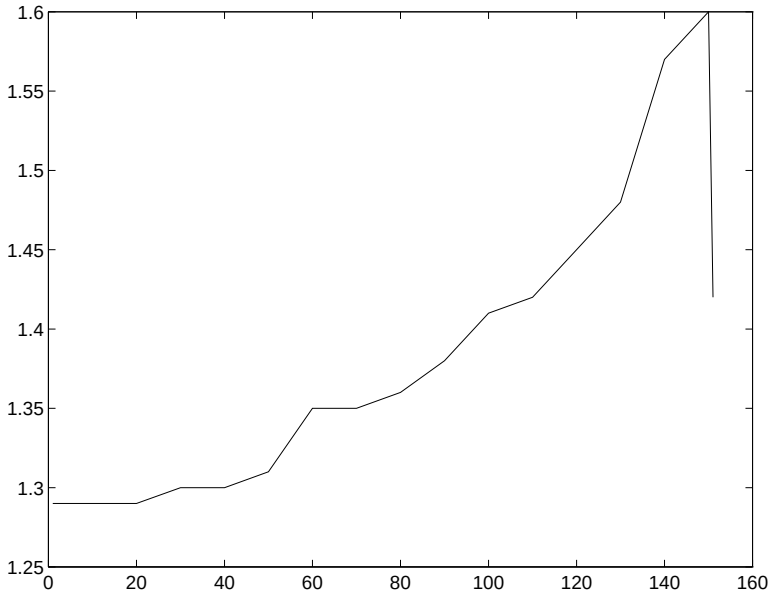



Figure 5.2: Parallel time per iteration for test case 1

### 5.3.2 Test case 2

The second one is a larger twin galaxy example with 10000 particles, with 64 processors used. Time step size is $10^{-4}$.

As in test case 1, we record the parallel time during the first 100 iterations in table 5.4. It's interesting to see that the parallel time is just experiencing some small fluctuations during this period. The reason behind this is that PYRROS schedule itself is not optimized, and the dynamic change in the problem is not large enough to cause any significant imbalance up to this point.

During the next 50 iterations the performance loss becomes much more significant, and when the 150th iteration is reached we see a need to reschedule. See table 5.5

| iteration no. | 1 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| parallel time | 2.51 | 2.51 | 2.52 | 2.50 | 2.55 | 2.52 | 2.50 | 2.51 | 2.56 | 2.54 | 2.59 |
| PTincre | 0 | 0 | 0.4% | -0.4% | 1.6% | 0.4% | -0.4% | 0 | 2.0% | 1.2% | 3.2% |

Table 5.4: The first 100 iteration of test case 2

| iteration no. | 110 | 120 | 130 | 140 | 150 |
|---|---|---|---|---|---|
| parallel time | 2.59 | 2.63 | 2.77 | 2.90 | 3.02 |
| PTincre | 3.2% | 4.8% | 10.4% | 15.5% | 20.3% |

Table 5.5: The 100th-150th iteration of test case 2

Next we use different methods for rescheduling. Recall that approach 2 is the global reuse of PYRROS, approach 3 is fast readjustment without repartition, and approach 4 is task graph spawning and fast reschedule. See table 5.6:

The result is similar to test case 1: Incremental approaches are very competitive and the costs are much lower. They can replace global rescheduling in these cases, because they perform no worse, or even slightly better for the incremental clustering case. Indeed we have proved that incremental cluster is at least as good as global DSC clustering for coarse grain graphs in chapter 4, and even for mixed grain graphs random tests have demonstrated its effectiveness.

In figure 5.3 the growth of parallel time is plotted.

**Summary of test cases 1 and 2**

We see that in test case 1 and 2, the local reschedule methods achieved impressive success. There are similarities between these two cases, mainly due to the fact that

| Method | Approach 2 | Approach 3 | Approach 4 |
|---|---|---|---|
| parallel time | 2.74 | 2.78 | 2.70 |
| Cost | 8.0 | 0.35 | 0.7 |

Table 5.6: The parallel time reduction and cost for different reschedule method on test case 2

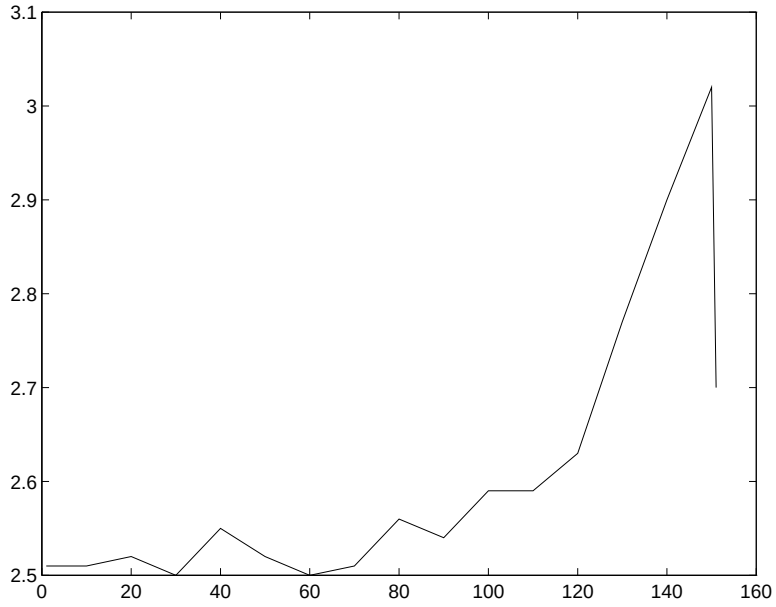



Figure 5.3: Parallel time per iteration for test case 2

both of them are highly clustered twin galaxy distribution, although having different numbers of particles.

We should note the underlying mechanism of applying approaches 3 and 4 to reduce the parallel time is different.

For approach 3, the goal of performance improvement is achieved by load readjustment between processors. So this approach is useful when there is significant load transfer among tasks, caused by global movement of the dynamic input distribution.

For approach 4, the same objective is achieved by breaking up tasks with excessive computation into subgraphs, and through incremental scheduling, mapping these subgraphs in a proper way to minimization the makespan. This is useful when there is concentration of weight in certain regions in the input, and moreover the breakup of these regions has to affect the critical path of the task graph.

It can be argued that the run-time variation in the previous two task graphs are two-fold. There are global movement of particles and redistribution of loads, while at the same time the dense cores of the galaxies are attracting even more particles and the density has increased considerably (above 25% for both cases). Therefore, both

approaches 3 and 4 work, although through different mechanisms. In fact we can see that the splitting of the dense core tasks gives slightly better results than load balancing without splitting, although the former has higher but still tolerable cost.

One can therefore, imagine cases where one approach applies and the other does not. For the classical N-body problem that we are dealing with, there are usually global shift of load, so approach 3 should generally work, albeit with various degree of improvement for different cases. But the concentration of particles may not occur in all cases. Even if such concentration does exist, it might not happen in the regions that could affect parallel time. So for such cases, approach 4 may not be applicable.

The same situation will be encountered in our later work in a variation of the N-body problem: the Vortex Sheet Computation. There we can also find cases where approach 3 works but not approach 4, and more interestingly, cases where approach 4 succeeds but not approach 3. The latter is due to very localized changes in the input distribution with little global effect.

Next we present our third case which is a very large input configuration with a total of $10^5$ particles, which illustrates the above point.

### 5.3.3 Test case 3

In this test case, we have an input distribution of 100000 particles, which are distributed in two regions of the space same as the previous twin galaxies distributions, but we have "tuned-down" the density concentration of the galaxies so they are rendered more uniform. (According to astrophysics this is more like the primitive galaxies where the dense clusters have not yet formed.) We do this in part because of our will to test a rather large example, and due to the small memory capability of the NCube2s machine, such adjustment has to be made to realize a continuous run. On the other hand, testing a different pattern of distribution should hopefully provide us with more insight to the reschedule problem as we discussed in the summary of the previous two cases. Time step size is again $10^{-4}$ in this case.

The first 100 iterations are listed in table 5.7. The increase in parallel time is 8.4%. And after another 40 iterations the parallel time jumps more drastically, with a total

| iteration no. | 1 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| parallel time | 34.7 | 34.7 | 34.5 | 34.6 | 34.7 | 35.0 | 35.5 | 36.2 | 36.8 | 37.0 | 37.6 |
| PTincre | 0 | 0 | -0.6% | -0.3% | 0% | 0.9% | 2.3% | 4.3% | 6.1% | 6.6% | 8.4% |

Table 5.7: The first 100 iteration of test case 3

| iteration no. | 110 | 120 | 130 | 140 |
|---|---|---|---|---|
| parallel time | 37.7 | 37.8 | 40.1 | 42.6 |
| PTincre | 8.6% | 8.9% | 15.6% | 22.8% |

Table 5.8: The 100th-140th iteration of test case 3

increase of 22.6% over the initial iteration. See table 5.8.

For this test case we found that approach 4 is not applicable. There is only an increase of about 8% in the highest density in all boxes, so the movement does not behave in the same manner as the previous test cases, and the new distribution will not trigger any splitting of boxes. On the other hand, approach 3 still works for this case. See table 5.9. Here we see that local readjustment can also achieve parallel time reduction that is close to that achieved by complete reschedule, and the cost is one order less. Note that this task graph is not 10 times as large as the one in test case 2, although there are 10 times as many particles. Instead it's only about 5 times as large due to different distribution patterns.

The parallel time changes are shown in figure 5.4. Note that the final drop is due to the application of fast schedule readjustment algorithm.

We have seen the usefulness of dynamic support on the N-body problem. On the other hand, the iterative performance behavior is not easy to predict and explain. To gain more insight of dynamic problems and the applicability of our dynamic support system, we turn to a problem that is a variation of the classical N-body problem: The

| Method | Approach 2 | Approach 3 |
|---|---|---|
| parallel time | 38.9 | 39.8 |
| Cost | 31.0 | 1.5 |

Table 5.9: The parallel time reduction and cost for approaches 2 and 3 on test case 3

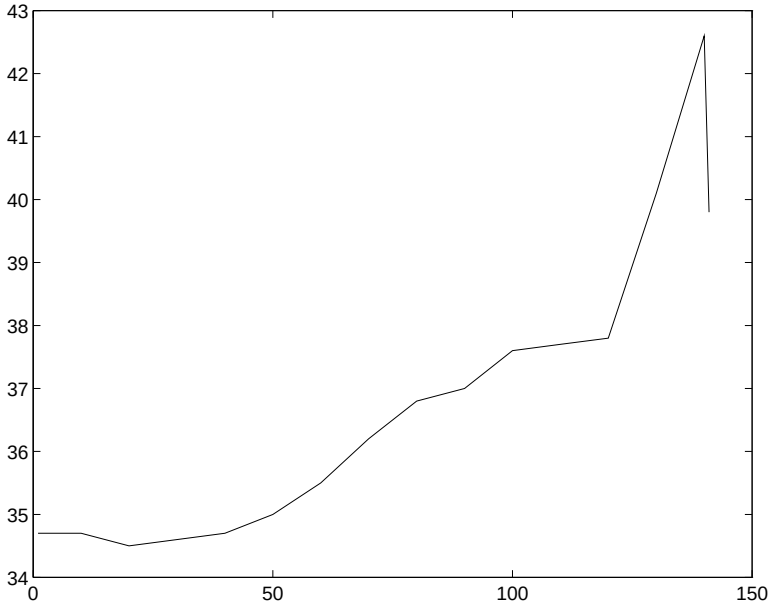



Figure 5.4: Parallel time per iteration for test case 3

vortex sheet computation in Computation Fluid Dynamic. The problem is studied in the next chapter, which focuses on the dynamic support system, using test examples that are more predictable in their iterative behavior.

# Chapter 6

# Testing Dynamic Support System on the Vortex Sheet Computation

## 6.1 Introduction and Problem Definition

The computation of vortex sheet roll-up, arising from aerodynamics studies to determine the fluid structure in an aircraft's wake, is a classic problem in Computational Fluid Dynamics(CFD). This problem has been extensively investigated by Robert Krasny [37], using simulations by sequential code, and the behavior of typical vortex sheet configurations is well-known. The analysis of the physical implications can be found in Krasny's original paper [37]. Recently Draghicescu ([17]) used a fast algorithm for sequential simulation. The algorithm is somewhat similar to the FMM algorithm.

We choose this particular problem to test our dynamic support system, not because there are strong interest in the fluid mechanics model involved, but based on the well-studied iterative behavior of the vortex sheet. Our research does not attempt to discover any new simulation result in fluid dynamics, which is out of the scope of our work. The subject of our research is, instead, how the vortex sheet roll-up process will affect the initial compile-time schedule produced by PYRROS, and how the run-time support heuristics will be able to handle such effects. Our system has been tested against the classical astrophysics N-body problem and yielded interesting results. This new problem, closely related to the classical N-body simulation in the sense of underlying physics laws, could provide us with deeper insight into the dynamic support problem due to the predictable roll-up patterns.

The following definitions are adopted from [37].

**Definition 6.1**

A vortex sheet is a curve in two dimensional space:

$$z(\Gamma, t) = x(\Gamma, t) + iy(\Gamma, t)$$

where $\Gamma$ measures the circulation at a point on the sheet and $t$ is time.

**Definition 6.2**

We define $\sigma = -\frac{d\Gamma}{ds}$, where $s$ is the arclength, as vortex sheet strength. For convenience we use polar coordinates: $\Gamma' = \Gamma(\alpha)$,$(0 \leq \alpha \leq \pi)$ and impose the condition of $\Gamma(0) = \Gamma(\pi) = 0$ on the tips.

Now the evolution of the vortex sheet through time t is governed by the following equation:

$$\frac{\partial \bar{z}}{\partial t} = \int_0^\pi K(z - \tilde{z})\Gamma'(\tilde{\alpha})d\tilde{\alpha}$$

where $z = z(\Gamma(\alpha), t)$, $\tilde{z} = z(\Gamma(\tilde{\alpha}), t)$, and $K(z) = -1/2\pi iz$ is the kernel of the integration. The bar over the time derivation denotes the complex conjugate of $z$.

The above equation, although physically different from the gravitational force equation in the classic N-body computation, nevertheless is computationally similar. Because if the integration is discretized into points along the sheet, the movement of any point is dependent upon all the other points, just as the astrophysics simulation, although with a totally different kernel. So fast algorithms, such as the Fast Multipole Method, can be applied to this problem with certain modifications.

For initial condition, we give $z(\Gamma(\alpha), 0) = c * -cos\alpha, 0 < c \leq 1$, which makes the sheet a straight line segment from [-c,0] to [c,0]. (The paper [37] makes c=1 but we give more freedom to the initial condition, which makes no fundamental difference. Also the y coordinate does not have to be zero. ) We still have to specify the circulation function $\Gamma$. There are two typical circulation distributions:

1. **Elliptic loading:**

   Here we set:

$$\Gamma(\alpha) = sin(\alpha)$$

   This simple distribution will result in a roll-up of the two tips of the sheet, as we shall see from the simulation.

2. **Fuselage-flap configuration:**

   This is more complicated than the elliptic loading case. Here the $\Gamma$ function is defined on different intervals of $x$, with $-1 \leq x \leq 1$. Here we assume that $c = 1$, if $c < 1$ simply scale the x coordinate.

   - For $0.7 \leq |x| \leq 1$, the distribution is the same as elliptic loading.
   - For each interval $0 \leq |x| \leq 0.3$ and $0.3 \leq |x| \leq 0.7$, $\Gamma$ is defined as a cubic polynomial in $|x|$, whose coefficients are chosen such that $\Gamma$ and $\sigma$ are continuous at $|x| = 0, 0.3$ and $0.7$.
   - Finally at $|x| = 0.3$, $\Gamma$ attains its maximum value 2, and at $x = 0$ there is a local minimum value 1.4.

   This distribution is designed to simulate aircraft dynamics, and the roll-up will occur in the middle of sheet, resulting in complicated shape later.

One issue concerning the vortex sheet simulation is that the singular kernel makes computational result highly unstable. So an engineering approach of introducing smoothing factor is employed. let $\delta > 0$ be an predetermined smoothing factor, we can replace the original kernel $K(z)$ with $K_\delta(z)$ such that:

$$k_\delta(z) = K(z)\frac{|z|^2}{|z|^2 + \delta^2}$$

This approximation will yield more stable numerical result if $\delta$ is chosen properly. Such a smoothing approach is not only employed in [37] but many other CFD simulations, such as in [6].

Next, we need to discretize the equation to enable numerical solution. The discretization is quite straightforward. We pick $2N+1$ points $z_j(t) = x_j(t) + iy_j(t)$ to approximate the exact positions of $z(\Gamma(\alpha_j), t)$ at equidistant parameters $\alpha_j = \pi\frac{j-1}{2N}$, $j = 1....2N+1$. Then, using trapezoidal rule to approximate (4), we get:

$$\frac{d\bar{z_j}}{dt} = \sum_{k=1}^{2N+1} K_\delta(z_j - z_k)w_k$$

where $w_k = \Gamma'(\alpha_k)\pi/2N$, $k = 1, ...., 2N+1$. Initially the points are located on the straight line: $z_j(0) = -cos\alpha_j$.

For an illustration of the sheet movements in the two kinds of initial configurations, the reader may refer to the last section of this chapter (section 6.5), where test cases 1 and 2 are for the Elliptic Loading and the other 2 cases for the Fuselage-flap configuration. These test cases will be discussed later.

## 6.2 Applying Fast Multipole Method to the Vortex Sheet Roll-Up problem

The discretized equation of the vortex sheet problem is remarkably similar to that of the classic N-body problem. Here the velocity of a vortex blob $z_j$ depends on all the other vortex blobs, and the faraway interaction can be approximated by grouping blobs into boxes. One difference is that in the classical N-body problem we are computing the total force induced at any particle by all other particles, while here the vortex-induced velocity is calculated. With accommodations to this difference, fast algorithms designed for the N-body problem may be applied.

The Fast Multipole Method (FMM) is one of these algorithms. Since we have discuss the algorithm and its task graph structure in the chapter 3. those topics will not be repeated here. It is clear that the adaptive version of the algorithm must be used, because the vortex blobs are lining up on a curve, which is very far from the uniform distribution. Moreover, the "smoothed" kernel $k_\delta$ itself is not harmonic, which makes it impossible to expand that kernel into a multipole expansion. Therefore we use the original kernel $K$ to obtain the multipole expansion for far-away interactions, and the smoothed kernel is reserved for near interactions. This is a natural way to handle

the non-harmonic kernels, because for faraway interactions the effect of introducing the smoothing factor is rather small, and for nearby interactions the smoothing factor helps to make the solution stable. This practice has been adopted by other researchers as well for vortex simulations. See [6].

There are other issues as well, most notably, the need of dynamic point insertion. The number of points required to discretize the vortex sheet is related to the smoothing factor $\delta$. The smaller the $\delta$, the larger the number of points $N$, because smaller $\delta$ enables finer resolution on the curve. It should be noted that $\delta$ is chosen artificially to avoid excessive interactions when two blobs become too close. So it can not be excessively small. In practice, the number of points initially needed is limited, on the order of hundreds. Later during the roll-up, when the vortex sheet stretches into a curve with many turns, the distance between successive vortex blob can become too far to permit a reasonable interpolation to the vortex sheet curve. The curve could even intersect itself, which is physically impossible. This requires dynamic insertion of new points (vortex blobs) in order to preserve the smoothness and resolution of the curve. We follow the insertion scheme used by Krasny in [37]:

Let $\epsilon$ be a parameter chosen to control the maximum distance allowed between successive points. The condition $|z_{j+1} - z_j| \leq \epsilon$ is enforced at every iteration. For every interval $[z_i, z_{i+1}]$ which the condition is violated, a new point $z_{new}$ is inserted between $z_i$ and $z_{i+1}$, the position of $z_{new}$ is determined as follows:

- The curve parameter value corresponding to the new point $\alpha_{new} = \frac{\alpha_i + \alpha_{i+1}}{2}$.
- The $x$ and $y$ coordinate of $z_{new}$ is determined by using the points $z_{i-1}, z_i, z_{i+1}, z_{i+2}$ to calculate two cubic polynomial interpolations with respect to $\alpha$, one for each of the coordinates $x$ and $y$. In this way, the curve plotted will actually be piecewise cubic polynomials. This is generally sufficient for our purpose.

At each time step, after examining all the intervals and carrying out the points insertions, we need to rename the points from one tip of the sheet to the other. More importantly, the total vortex sheet circulation should be maintained as a constant, so the

weights of $w_k$ in the discretized equation (5) should be updated as $\Gamma'(\alpha_k) * \pi/(N_n - 1)$, where $N_n$ is the new total number of points.

The insertion parameter $\epsilon$ should be chosen appropriately. An $\epsilon$ that is too large could not control the shape of the curve, while an excessively small value will cause unnecessary insertion of points, which leads to needless computation cost.

Dynamic insertions can substantially reduce the overall amount of computation effort in a simulation process since new vortex blobs are introduced on demand. The cost is significantly lower than allocating many points on the initial straight line and fix them throughout the iterations. On the other hand, this leads to more degree of complexity in the parallelization.

Recall that in the classic N-body problem, the particle movements across the boundaries of space subdivision imposed by the FMM algorithm will slowly cause deterioration of the performance. This is because the original schedule obtained from the initial task graph can become imbalanced and need adjustment. We have developed a dynamic support system based on two heuristic algorithms to handle this problem in chapter 4. Here we briefly summarize the results :

- A fast readjustment algorithm aimed at tasks graphs that do not change in structure but with weights alterations. The algorithm tries to move tasks across processors for load balancing purpose. But unlike usual load-balancing methods, it takes the dependences into consideration to avoid movements that may cause increases in parallel time. The algorithm, with complexity $O(n_c * d)$, where $n_c$ is the number of tasks with large weight increases (a threshold was set for this measurement), and $d$ is the average degree of the task graph, is meant to be localized in the sense that it focuses on those task with large weight changes. For most task graph for real applications, including the FMM algorithm, $d$ is a constant, rendering the complexity $O(n_c)$.

  This algorithm can be applied to the N-body problem if the user decide to do fast schedule adjustment without having to repartition the boxes.

- An incremental rescheduling algorithm to handle task graphs that spawn new

subgraphs from certain nodes. This algorithm is based on the DSC clustering algorithm, but it applies local clustering on the new subgraphs and merges them with existing clustering, instead of global clustering. It can be shown that this approach gives results that are no worse than global clustering if the graph is coarse grain. The complexity of this algorithm is based on the size of local subgraphs and is obviously much less than global DSC.

This algorithm can be employed if the user choose to repartition . Now the boxes with excessive particle density will subdivide into finer levels, and the newly spawned subgraph can be incrementally scheduled.

For the vortex sheet simulation, the above approaches still apply since particles (vortex blobs) can also move across boxes. The extra complication has to do with the point insertion scheme mentioned above. The insertion will increase the total amount of computation, but this does not necessarily imply a need of rescheduling. For example, assuming there are $p$ processors, after scheduling each processor is assigned some tasks corresponding to certain leaf boxes. If the insertion is roughly balanced across all processors, we may not need to reschedule, unless all boxes have too many vortex blobs and need subdivision. With the presence of insertion, the increase in parallel time does not necessarily indicate a need for reschedule. Instead, the current number of vortex blobs must be taken into account.

We use the ratio of parallel time to the number of points $R = PT/N_n$, to measure the performance of current schedule. This is a reasonable measure because the fast multipole method theoretically has complexity $O(N)$, although the real running time can be input dependent. If this ratio increases beyond a tolerable range after certain number of iterations, there is strong indication of the need of either fast adjustment (the first algorithm), or subdivision and incremental reschedule (the second algorithm).

Because of the predictable changes in the shape of the curve, we hope that this problem becomes a useful testbed for the two dynamic support algorithms. The next section describes some of our results regarding the application of the dynamic support system.

| iteration no. | 1 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| parallel time | 628 | 629 | 627 | 627 | 628 | 628 | 627 | 633 | 709 | 839 | 928 |
| number of blobs | 401 | 401 | 401 | 401 | 401 | 401 | 401 | 401 | 421 | 451 | 473 |
| PT/No of blobs | 1.57 | 1.57 | 1.57 | 1.57 | 1.57 | 1.57 | 1.57 | 1.58 | 1.68 | 1.86 | 1.96 |

Table 6.1: The first 100 iteration of test case 1

## 6.3 Applicability of Dynamic Support Algorithms on Typical Test Cases of the Vortex Sheet Roll-Up Problem

We have tested our system against 4 typical input cases. Here we describe these test cases in considerable detail.

### 6.3.1 Test case 1

**The input configuration:**

We start with a basic elliptic loading case. The following parameters are chosen:

Time step size $\Delta t = 0.002$, smoothing factor $\sigma = 0.02$, maximal distance allowed between successive vortex blob $\epsilon = 0.009$. Initial number of points is 401. 8 processors are assigned.

We record the parallel time per iteration and the ratio of parallel time to the number of points. The total number of time steps of the simulation is 200, at which point the vortex sheet is settled into a shape with many turns at the tips.

**The first 100 iterations**

The first 100 iterations are listed in table 6.1, the unit of parallel time is millisecond.

At this point the ration $R = PT/N$ has increased substantially (25%), so we would like to reschedule.

We find that the fast schedule readjustment method (algorithm 1) will not reduce the parallel time. If we look carefully at this problem, we can see the reason behind such failure: The insertion of points are almost all occurring near the two tips of the sheet. This is because the roll-up is limited to those regions. And because the boxes containing the tips have very large weight from the beginning until the 100th iteration,

| iteration no. | 101 | 110 | 120 | 130 | 140 | 150 |
|---|---|---|---|---|---|---|
| parallel time | 572 | 642 | 717 | 754 | 793 | 832 |
| No of blobs | 473 | 494 | 511 | 519 | 527 | 538 |
| PT/No of blobs | 1.21 | 1.30 | 1.40 | 1.45 | 1.50 | 1.55 |

Table 6.2: The 101-150th iteration of test case 1

they essentially lie on the critical path(s) of the task graph during the whole iterative process. Without breaking up of these boxes, it's not possible to reduce parallel time by adjustment of tasks among processors.

**The first reschedule attempt**

Thus the nature of the problem indicates the need of repartition, which will split the boxes containing the highly curved tip regions. Such repartition, coupled with incremental scheduling heuristic, yields a parallel time of 572ms at iteration 101, down from 928ms for the previous iteration, which is really an impressive improvement. The overhead is merely 20ms. If we use global rescheduling on the spawned task graph we get a new parallel time of 581ms with scheduling cost 300ms. This clearly shows that incremental clustering is no worse than global clustering with only a fraction of its cost.

**Iterations 101 through 150**

After this rescheduling step, we continue our simulation, recording the information of iterations 101 through 150 in table 6.2.

**The second reschedule attempt**

At the 150th iteration, with parallel time reaching 832ms, the user may decide that a new round of rescheduling is beneficial. The situation is similar to the 100th iteration, where the points insertions has been concentrated at boxes at the tips (now the boxes are of finer size.) So as before, without repartitioning and box subdivision we can not expect to achieve parallel time reduction. The method of repartition plus incremental rescheduling still performs well, achieving a new parallel time of 587ms.

**The last 50 iterations**

| iteration no. | 151 | 160 | 170 | 180 | 190 | 200 |
|---|---|---|---|---|---|---|
| parallel time | 587 | 601 | 714 | 785 | 846 | 878 |
| No of blobs | 540 | 547 | 578 | 609 | 639 | 655 |
| PT/No of blobs | 1.09 | 1.10 | 1.24 | 1.29 | 1.32 | 1.34 |

Table 6.3: The 151-200th iteration of test case 1

The last 50 iterations are recorded in table 6.3, with the parallel time finally settled at 878ms, with 655 blobs on the vortex sheet.

**The total effect of rescheduling.**

If the initial partitioning and schedule is used throughout the computation, the parallel time can climb to as large as 1549ms, 76% higher than the result achieved by using the incremental rescheduling method twice. This is illustrated in figure 6.1.

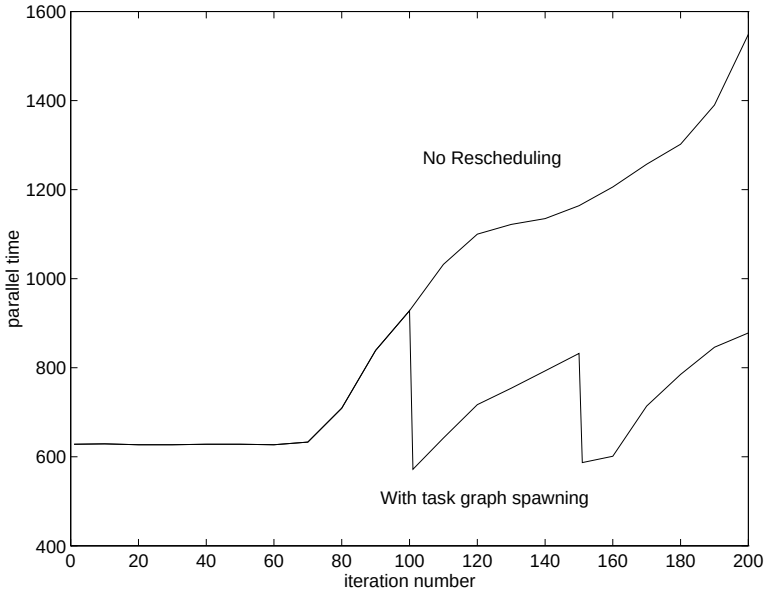



Figure 6.1: Parallel time per iteration for test case 1

### 6.3.2 Test case 2

**The input configuration**

This test case is similar to test case 1, with more vortex blobs and smaller $\sigma$. Thus the sheet movement is still concentrated at the tips.

| iteration no. | 1 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 | 110 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| parallel time | 969 | 971 | 975 | 969 | 969 | 963 | 959 | 948 | 945 | 939 | 980 | 1203 |
| number of blobs | 801 | 801 | 801 | 801 | 801 | 801 | 801 | 801 | 801 | 801 | 849 | 895 |
| PT/No of blobs | 1.21 | 1.21 | 1.22 | 1.21 | 1.21 | 1.20 | 1.20 | 1.18 | 1.18 | 1.17 | 1.15 | 1.34 |

Table 6.4: The first 110 iteration of test case 2

Here we have the following parameters: $\Delta t = 0.0008$, $\sigma = 0.01$, $\epsilon = 0.005$. The initial number of points is 801 and 16 processors are used.

**The first 110 iterations**

The first 110 iterations of the execution are listed in table 6.4.

Here the performance of the schedule stays very stable and there is even some minimal decrease in parallel time. But after 100th iteration new points are inserted at an accelerating speed, and the parallel time increases fast during 100 to 110th iterations. Although the absolute increase in $R$ is not very big (11% over the 1st iteration), the rapid growth during the last 10 iterations does imply a need to reschedule.

**The first reschedule attempt**

As in test case 1, we found that only by repartitioning can we reduce the parallel time. By splitting the boxes at the tip and incremental rescheduling, we can make substantial gain in performance: parallel time improves to 922ms at iteration No. 111, down from 1203ms, with scheduling cost at about 40ms.

**Iterations 111 through 180**

After this reschedule, The parallel time is little changed for the next 50 iterations even as the number of blobs grows, but it finally grow to 1177ms at the 180th iteration. See table 6.5.

It is noticeable that in this test case the parallel time can remain unchanged or even slightly decrease as more points are inserted, unlike the first test case. This is due to the finer partition of the space. The initial box sizes are smaller than the first case, and become still finer after one splitting operation. So the roll-up of vortex sheet tip

| iteration no. | 111 | 120 | 130 | 140 | 150 | 160 | 170 | 180 |
|---|---|---|---|---|---|---|---|---|
| parallel time | 922 | 915 | 905 | 899 | 892 | 938 | 1105 | 1177 |
| No blobs | 899 | 928 | 965 | 988 | 1005 | 1020 | 1033 | 1049 |
| PT/No blobs | 1.03 | 0.99 | 0.94 | 0.91 | 0.89 | 0.92 | 1.07 | 1.12 |

Table 6.5: The 111-180th iteration of test case 2

could move vortex blobs across box boundaries more easily, resulting in a temporary dispersal of computation load. Nevertheless, eventually the insertion creates substantial performance degeneration.

**The second reschedule attempt and the last 20 iterations**

At this point, a repartition and local reschedule can still help reducing the parallel time back to 950ms at iteration 181. The final (200th) iteration takes 1074ms, with 1126 total points.

**The total effect of rescheduling**

Without repartition and reschedule, the performance on this case would be poor: Our test shows that the parallel time can jump to 3100ms at the last iteration, which is 189% over what we obtained by rescheduling. Here the effect of dynamic support is more dramatic than the first test case. See figure 6.2.

Summing up the first two cases, we see both similarities and differences: The movement of the vortex sheet is localized at the tips for both cases, and spawning technique can be applied to such cases, whereas in the second test case smaller box sizes contribute to slightly different trend in the parallel time.

After examine the elliptic loading input distribution, we turn to the simulated fuselage-flap initial conditions, where the vortex sheet stretches in the middle section and the movement is much more global.

### 6.3.3 Test case 3

**The input configuration**

Test case 3 is a fuselage -flap distribution. The input parameters are as follows:

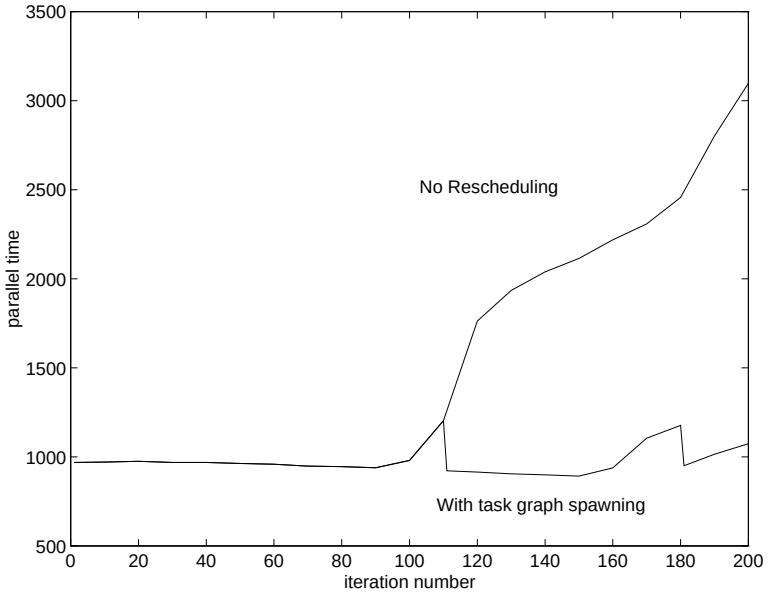



Figure 6.2: Parallel time per iteration for test case 2

| iteration no. | 1 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 |
|---|---|---|---|---|---|---|---|---|---|---|
| parallel time | 628 | 628 | 628 | 628 | 632 | 640 | 674 | 667 | 611 | 601 |
| number of blobs | 401 | 401 | 401 | 401 | 401 | 401 | 401 | 401 | 401 | 401 |
| PT/No of blobs | 1.57 | 1.57 | 1.57 | 1.57 | 1.58 | 1.60 | 1.68 | 1.66 | 1.52 | 1.50 |

Table 6.6: The first 90 iteration of test case 3

$\Delta t = 0.002$, $\sigma = 0.02$, $N = 401$, and $\epsilon = 0.02$. 8 processors are assigned.

The initial conditions chosen are identical to that of test case 1 except for the different functional values of $\Gamma(\alpha)$. Because the stretch of the vortex sheet is more extensive in this case, $\epsilon$ should be larger than the value used in the simple elliptic loading case to avoid excessive on-line vortex creation.

**The first 130 iterations**

During the first 90 iterations, there is no point insertion, and the parallel time first increases slightly from the initial value of 628ms, then shows some decline. The shape of the vortex sheet changes from straight line to wave-like curve, which causes considerable amount of vortex blobs to cross box boundaries. This could help disperse the load among processors. See table 6.6.

After the 90th iteration there start to be some point insertions. The parallel time

| iteration no. | 100 | 110 | 120 | 130 |
|---|---|---|---|---|
| parallel time | 629 | 644 | 753 | 997 |
| number of blobs | 422 | 475 | 503 | 533 |
| PT/No of blobs | 1.49 | 1.36 | 1.50 | 1.87 |

Table 6.7: The 91-130th iteration of test case 3

increases slowly at first, then at an accelerating rate. It climbs to almost 1 second at the 130th iteration, a very high level. See table 6.7.

**The first and the second reschedule attempts**

At this point the rapidly degrading performance calls for rescheduling. Contrary to the elliptic case where we repartition the boxes and make use of local clustering to handle the spawning of task graph, we now apply the approach of fast schedule adjustment without repartition. This is because there is no excessive amount of vortex blobs in any of the boxes. In fact, the maximal amount of blobs contained in any single box is less than the level at the initial configuration, despite the insertion of 132 new blobs. A closer examination of the vortex sheet reveals that the insertions are all in the middle section of the curve, where the initial vortex density is low but later wave-like shape is formed.

So the performance loss is due to load imbalance induced by global movement of the vortex sheet, especially in the middle section, rather than the concentration of vortex blobs. Such a situation should be corrected by the fast schedule adjustment algorithm. Indeed it does reduce the parallel time to 820ms at the 131th iteration, with a cost of 16ms. (Reschedule from scratch can reduce the parallel time to 815ms with a cost of 300ms.) This in effect pushes the $R = PT/N$ ratio back to about 1.5.

But then the parallel time starts climbing rapidly again due to swift curve movements, reaching 967ms in just 5 iterations. To curb the alarming trend of performance deterioration, we decide to use the fast reschedule algorithm once more, achieving a reduction to 910ms at iteration No. 137, with a cost of 16ms.

Although the reduction itself is modest, we will see that this second schedule adjustment does help contain the speed of parallel time increase. This will be clearer when

| iteration no. | 137 | 140 | 150 | 160 |
|---|---|---|---|---|
| parallel time | 910 | 966 | 1108 | 1249 |
| no. blobs | 541 | 555 | 610 | 655 |
| PT/no. blobs | 1.68 | 1.74 | 1.82 | 1.91 |

Table 6.8: The 137-160th iteration of test case 3

| iteration no | 161 | 170 | 180 | 190 | 200 |
|---|---|---|---|---|---|
| parallel time | 1026 | 1104 | 1148 | 1264 | 1431 |
| no of blobs | 657 | 692 | 732 | 763 | 812 |
| PT/no. blobs | 1.56 | 1.60 | 1.57 | 1.66 | 1.76 |

Table 6.9: The 161-200th iteration of test case 3

we compare the result to the situation where no reschedule attempt is made.

**Iterations 137 through 160**

The next table 6.8 describes the record of iterations 137 – 160:

**The third reschedule attempt and the last 40 iterations**

At iteration 161, we carry out fast schedule adjustment for the third time. This time we are again very successful, achieving a performance gain to 1026ms at iteration 161. Thus we improve the $R$ ratio back to about 1.56, with a cost of 20ms. As usual the cost of rescheduling from scratch is much higher (350ms) and it results in a parallel time of 1041ms, which is even a little bit worse.

Table 6.9 shows the record of iterations 161 to 200.

**The total effect of rescheduling**

For comparison purpose, we list the parallel time of iterations 130 through 200 if we continue the simulation without any rescheduling, in table 6.10. The comparison of parallel time is plotted in figure 6.3.

Giving the rapid climbing of parallel time after iteration 130, we can appreciate how well our reschedule method has performed. The final parallel time is 126% higher without rescheduling. And notice that the performance loss is especially striking between iterations 130 and 170, which suggests that our application of fast reschedule

| iteration no | 130 | 140 | 150 | 160 | 170 | 180 | 190 | 200 |
|---|---|---|---|---|---|---|---|---|
| parallel time | 997 | 1311 | 1642 | 1997 | 2282 | 2610 | 2824 | 3240 |
| No of blobs | 533 | 555 | 610 | 655 | 692 | 732 | 763 | 812 |
| PT/no. blobs | 1.87 | 2.36 | 2.69 | 3.05 | 3.30 | 3.57 | 3.70 | 3.99 |

Table 6.10: The last 70th iteration of test case 3, without rescheduling

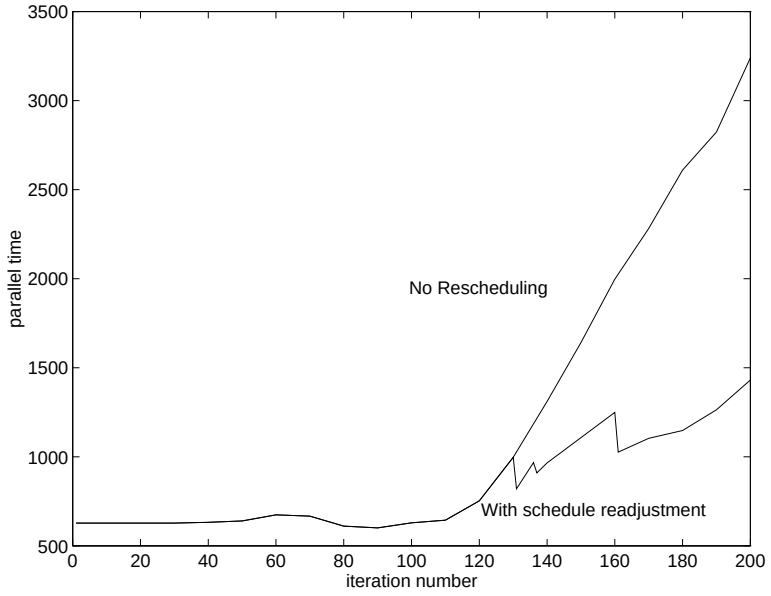



Figure 6.3: Parallel time per iteration for test case 3

| iteration no. | 1 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| parallel time | 969 | 978 | 978 | 991 | 991 | 995 | 1006 | 1031 | 1058 | 1065 | 1065 |
| no of blobs | 801 | 801 | 801 | 801 | 801 | 801 | 801 | 801 | 801 | 801 | 801 |
| PT/no. blobs | 1.21 | 1.22 | 1.22 | 1.24 | 1.24 | 1.24 | 1.26 | 1.29 | 1.32 | 1.33 | 1.33 |

Table 6.11: The first 100 iterations of test case 4

| iteration no. | 110 | 120 | 130 | 140 | 150 |
|---|---|---|---|---|---|
| parallel time | 1081 | 1066 | 1043 | 1082 | 1364 |
| No of blobs | 807 | 813 | 817 | 871 | 965 |
| PT/no.blobs | 1.34 | 1.31 | 1.28 | 1.24 | 1.41 |

Table 6.12: The 101-150th iteration of test case 4

twice during iterations 130-140 did contain the performance loss to acceptable levels.

This test case gives strong evidence that fast schedule readjustment should be applied when the input configuration exhibits global changes and causes large amount of computation load transfer across space partitions. The next test case will deal with the same fuselage-flap distribution with more vortex blobs.

### 6.3.4 Test case 4

#### The input configuration

This distribution is a fuselage-flap input configuration with the following parameters: $\sigma = 0.015$, $\delta t = 0.0012$, $N = 801$, $\epsilon = 0.01$. The number of processors is 16.

#### The first 150 iterations

The first 100 iterations are listed in table 6.11. The parallel time is quite stable, with very slight increase. The number of vortex blobs remains at 801.

During the next 50 iteration there are point insertions. The parallel time stays below 1100ms before the 140th iteration, but eventually the load imbalance starts to appear and leads to rapid performance loss at 140-150th iterations, during which many points are inserted. See table 6.12.

#### The first reschedule attempt

| iteration no. | 151 | 160 | 170 | 180 |
|---|---|---|---|---|
| parallel time | 1146 | 1269 | 1426 | 1579 |
| No of blobs | 969 | 998 | 1027 | 1065 |
| PT/no blobs | 1.18 | 1.27 | 1.39 | 1.48 |

Table 6.13: The 151-180th iteration of test case 4

We employ our fast schedule readjustment algorithm, achieving a very substantial performance gain. The parallel time is improved to 1146ms at iteration 151, with a minimal cost of 40ms.

**Iterations 151 through 180 and the second reschedule attempt**

During the next 30 iterations the parallel time increases rapidly again because of drastic global vortex sheet movement. At iteration 180 the ratio $R$ reaches 1.48, and a new round of rescheduling act is called for. See table 6.13.

This time our fast reschedule algorithm is able to reduce the parallel time to 1325ms, and the $R$ value is pushed back to 1.23.

**The last 20 iterations and the third reschedule attempt**

At this stage of the simulation, the vortex sheet has a basically settled shape, with large vortices forming in the middle with many turns. In the later stage of the simulation the big vortices will continue to twist, but the global shift of the sheet becomes minimal, so does the load transfer across boxes. On the other hand, the big vortices keep rotating into complex turning structure, which leads to many points insertions and heavy concentration of the vortex blobs in these regions. The situation starts to resemble that of the elliptic loading case.

Indeed, during iterations 180-190, there are as many as 132 points inserted, causing the parallel time to skyrocket to as high as 1945ms at iteration 190 from 1325ms at iteration 181, and the $R$ value hits 1.62.

We found that fast schedule adjustment without splitting boxes can not improve the performance anymore, just as what is expected because the insertions are local within the middle vortex structures. We turn to repartition and incremental task graph clustering method for help. After splitting boxes with excessive number of vortex blobs

| iteration no. | 150 | 160 | 170 | 180 | 190 | 200 |
|---|---|---|---|---|---|---|
| parallel time | 1346 | 1918 | 1974 | 2102 | 2607 | 2956 |
| No. of blobs | 965 | 998 | 1027 | 1065 | 1197 | 1237 |
| PT/no blobs | 1.39 | 1.92 | 1.92 | 1.97 | 2.18 | 2.39 |

Table 6.14: The last 50 iterations of test case 4 without rescheduling

and incremental reschedule, the parallel time is pushed back to 1539ms, and it settled at 1655ms at iteration 200. the final total number of vortex blob is 1237, so the $R$ ratio is at 1.34, an acceptable level.

**The total effect of rescheduling**

If no reschedule is attempted, the final parallel time is 2956ms, 79% higher than the result with reschedule. The difference is made within the last 50 iterations since during the first 150 iterations the changes in the input configuration did not lead to significant performance loss. See table 6.14 for the data on the last 50 iterations without reschedule, and figure 6.4 for a illustration.

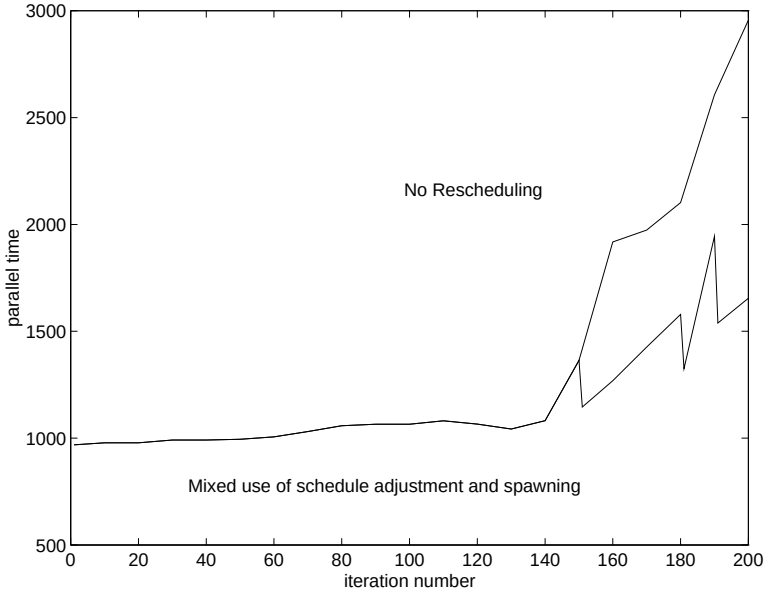



Figure 6.4: Parallel time per iteration for test case 4

This last test case provides a successful example of mixed usage of both fast reschedule without repartition, and incremental reschedule with repartition during various

stages of the simulation.

### 6.3.5 Improvement of speedup performance

We can also see the effect of reschedule from a different perspective, which is the improvement of speedup over the sequential execution. Since the number of vortex blobs in the simulation is small, the absolute speedup figure is not expected to be high. However, with a given number of processors, it should steadily climb due to the insertion of new blobs, which causes an increase in computation cost and granularity. But such improvement can only be maintained with the help of rescheduling, otherwise the effect of imbalance of processor load will take over and the performance degenerates.

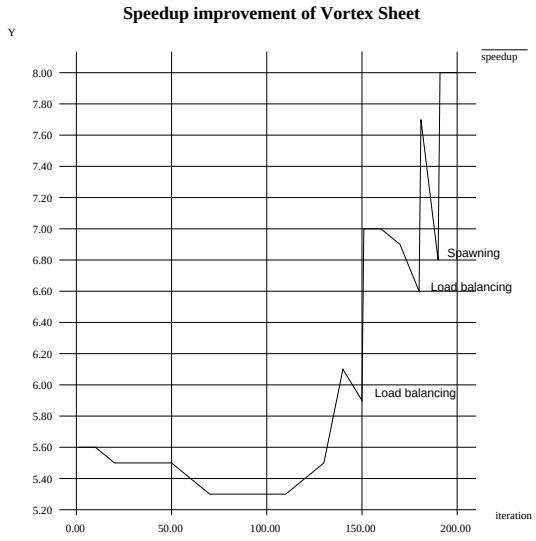



Figure 6.5: Speedup improvement due to rescheduling and point insertion

We show the trend of speedup on 16 processors for test case 4. Initially the speedup is at 5.6 and slightly declines to about 5.3, before going back up to 6.1 due to insertions of new points, but this improvement can not be maintained and the speedup starts to drop. A fast readjustment step brings the speedup to 7.0, and the next readjustment again breaks the declining trend and moves it up to 7.7. When the speedup drops to

6.8 again, the last rescheduling step corrects the imbalance and the final speedup is at 8.0. See figure 6.5

Without rescheduling, the final speedup will be lower than the initial value of 5.6 despite insertion of new points.

## 6.4 Conclusions

In this chapter we parallelized the Fast Multipole Method for the Vortex Sheet Roll-up problem using scheduling paradigm on the Ncube2s machine. Our major goal is to test the effectiveness of our dynamic support system on this particular problem, where the movement of vortex sheet is more predictable than the classical N-body problem. We have succeeded in gaining considerable insight to the applicability of dynamic rescheduling through the simulation process. We briefly summarize our experience as follows:

1. Run-time rescheduling support is important for the kind of irregular and dynamic problems like the vortex-sheet and classical N-body computation. Although the initial schedule can usually be reused for a substantial number of iterations, as we found before, eventually the dynamic nature of the problem will cause large deterioration of performance and we must provide run-time support when needed.

2. Run-time rescheduling is very effective for such problems if applied appropriately. Our test cases showed tremendous performance gains when rescheduling is attempted. The cost of fast localized algorithms for reschedule is only a fraction of that of scheduling from scratch, and such fast algorithms deliver competitive performance. The parallel time reductions are usually very significant and far exceed the overhead paid for rescheduling.

3. The nature of problem and the nature of the partitioning determines the run-time behavior of the simulation, and the choice of run-time support method. When the increase in parallel time is due to the insertion of points heavily clustered in a few boxes, and there is little global movement of the underlying configuration, the best way to achieving performance gain is to repartition and split those boxes

with excessive computation load, and incrementally schedule the spawning task graph, which is the Algorithm 2 described in chapter 4. On the other hand, if the cause of performance loss is not the concentration of particles in certain localized areas, but the global movement and load transfer across box boundaries, we expect that fast schedule readjustment, Algorithm 1 in chapter 4, will help balance the computation load in each processor and correct the imbalance.

Sometimes the choice of repartition or load balancing without repartition is not clear-cut, and both can lead to parallel time reductions. A example is the first 2 test cases we used in the classical N-body simulation in chapter 5. In those cases there are many more particles and the movements are more complex and lack a definite pattern. Both trends exist in those cases: There are significant number of particles moving across boxes boundaries, and there are also particles clustering in the boxes with higher density. In such situation, parallel time reduction may be achieved either by correcting load imbalance, or by splitting high density particle clusters into finer partitions. We have indeed seen that both approaches work in chapter 5. In addition, as we observed in test case 4, during different phases of the simulation the patterns of configuration change could also shift, requiring a mix of both approaches during the whole simulation process.

4. Generally, the movements of the input configuration happen gradually, so the initial schedule may be reused for a significant number of iterations. But there could be some short periods during which the cumulative effect of the dynamics reaches a critical point and something dramatic happens, like many particles crossing box boundaries, or a burst of new vortex insertions. During such highly chaotic periods of the simulation, more effort must be made to curb the loss of performance, such as repeated use of fast schedule readjustment in test case 3, or a mixed application of schedule readjustment and box splitting in iterations 180-190 in test case 4.

Our experience has shown the importance and applicability of dynamic rescheduling support system. To the best of our knowledge, this particular approach of handling

irregular and dynamic task graphs has not been investigated by any other researcher, since existing works mainly deal with load balancing without consideration of task graph dependences. Therefore, much remains to be done for a more in-depth characterization of dynamic behavior of different problems, especially those problems that could have entirely different underlying patterns and require new approaches to correct run-time performance deterioration. In the next chapter of the thesis we will look at the contour dynamics problem in Computation Fluid Dynamics, which is one of such applications.

## 6.5 The Shape of Vortex Sheet During Iterations of the Test Cases.

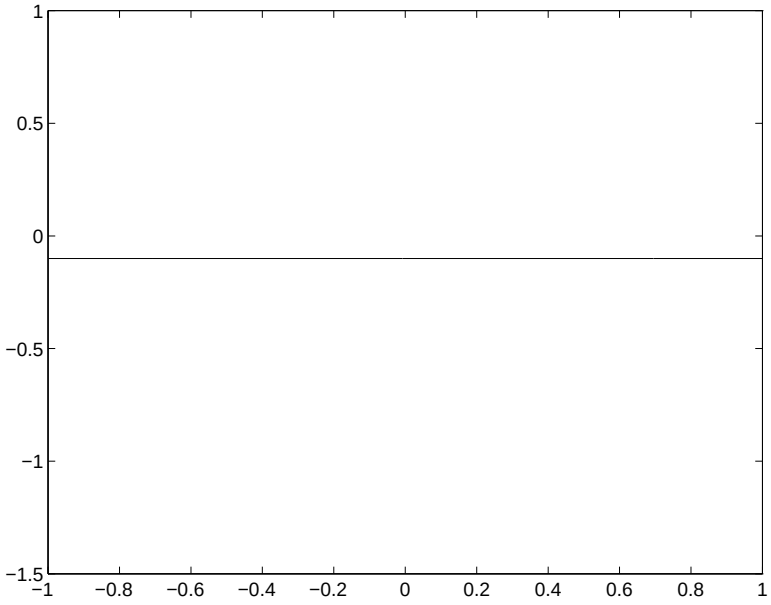



Figure 6.6: The initial vortex sheet line for all test cases

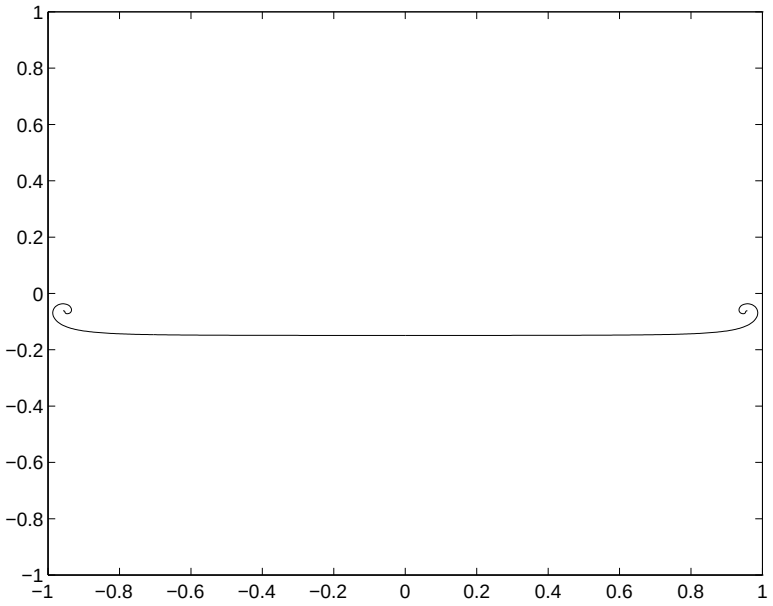



Figure 6.7: Test case 1, the vortex sheet after 50 iterations

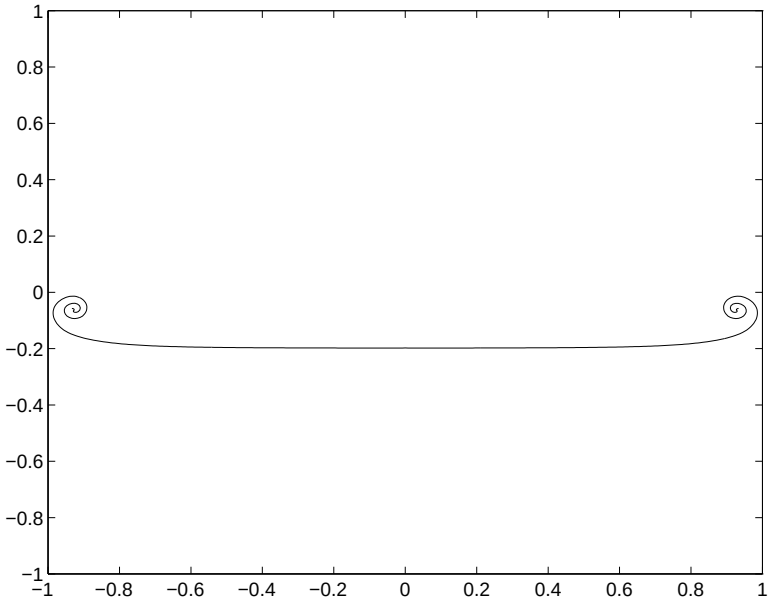



Figure 6.8: Test case 1, the vortex sheet after 100 iterations

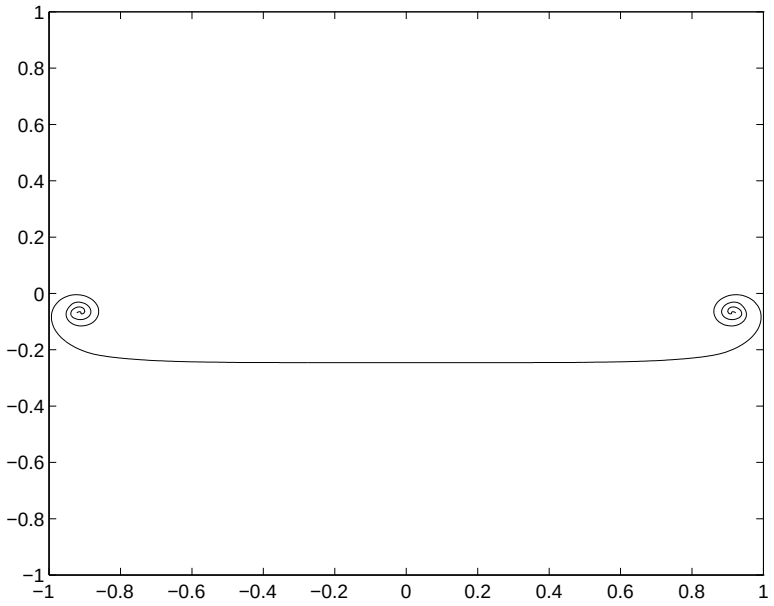



Figure 6.9: Test case 1, the vortex sheet after 150 iterations

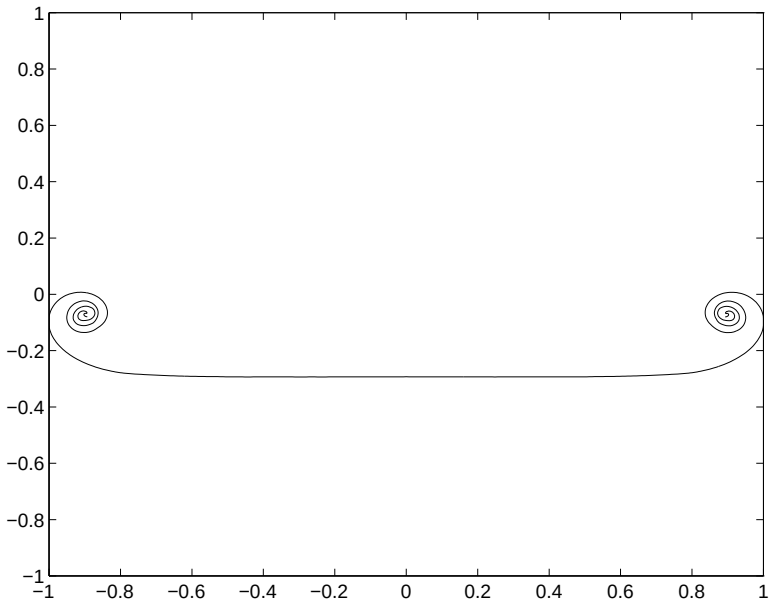



Figure 6.10: Test case 1, the vortex sheet after 200 iterations

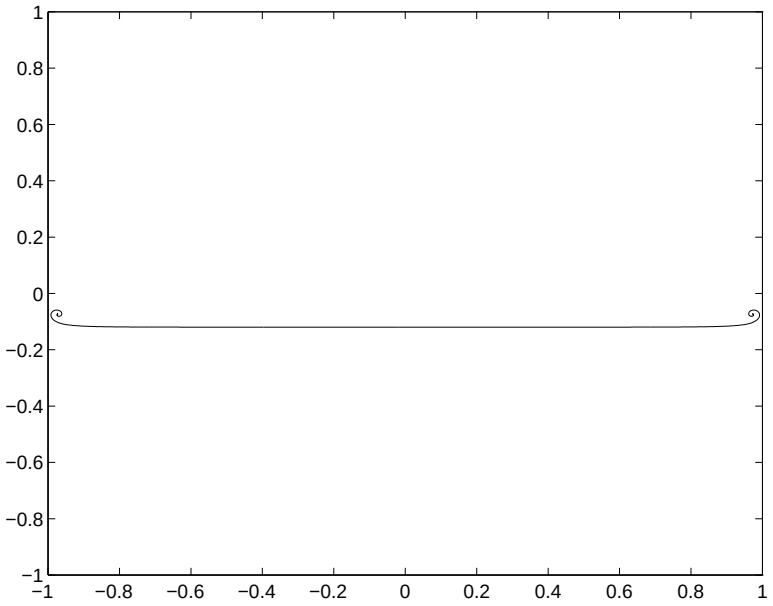



Figure 6.11: Test case 2, the vortex sheet after 50 iterations

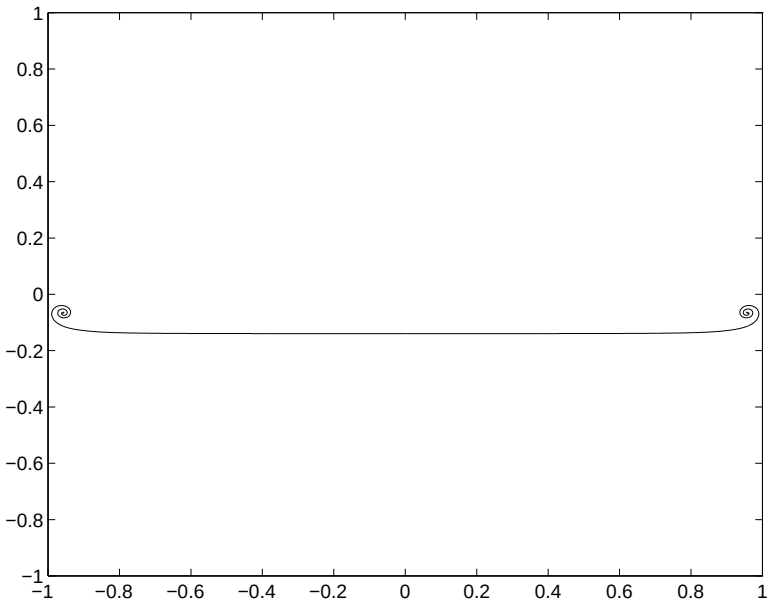



Figure 6.12: Test case 2, the vortex sheet after 100 iterations

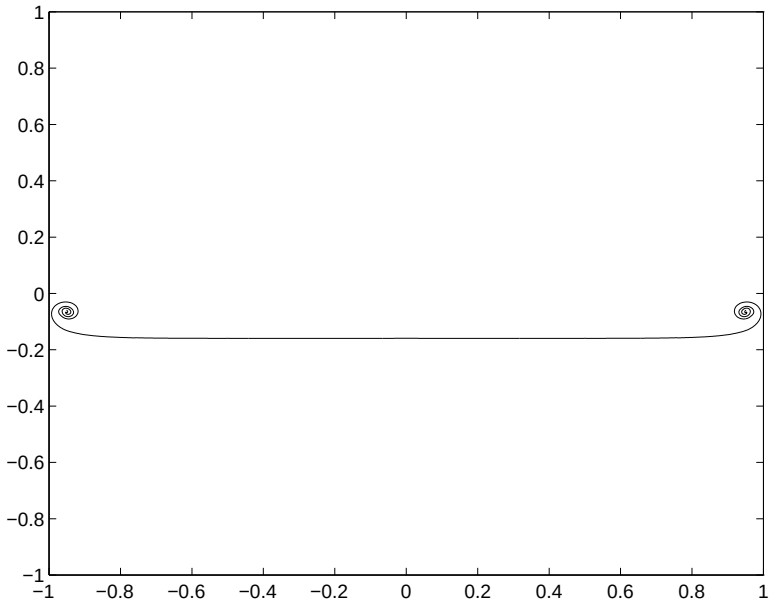



Figure 6.13: Test case 2, the vortex sheet after 150 iterations

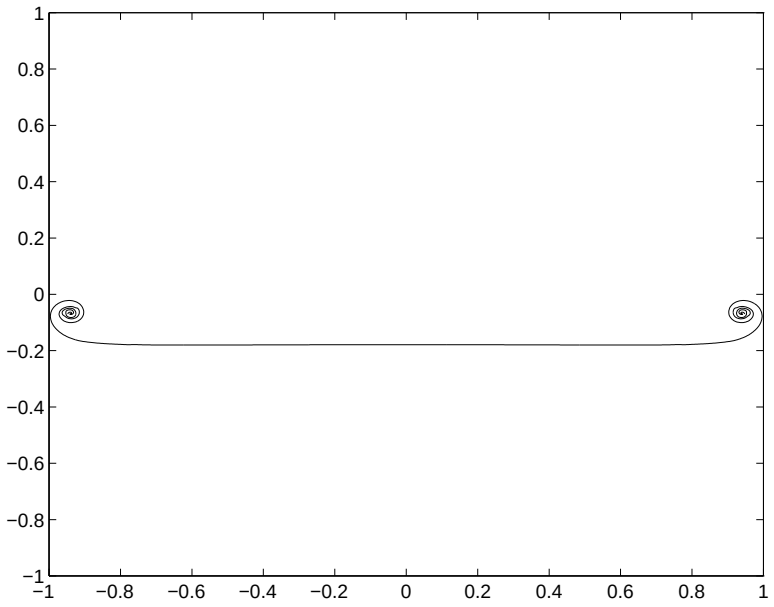



Figure 6.14: Test case 2, the vortex sheet after 200 iterations

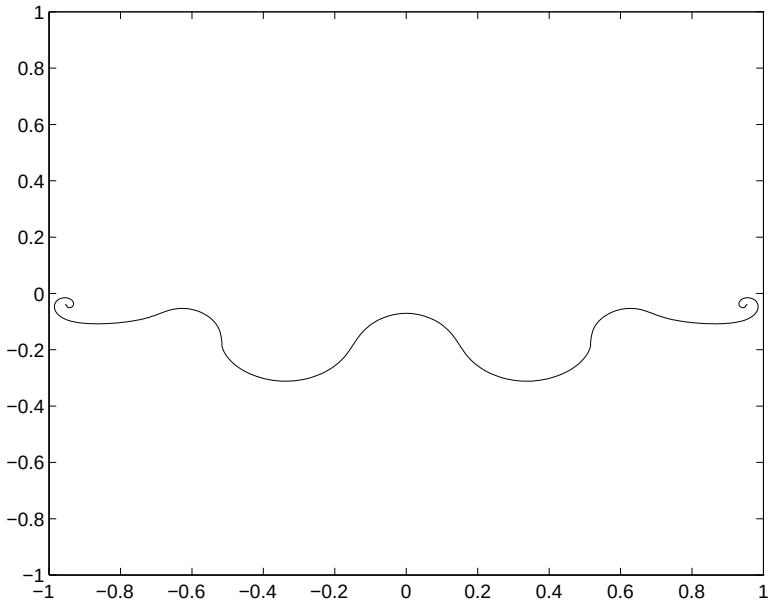



Figure 6.15: Test case 3, the vortex sheet after 50 iterations

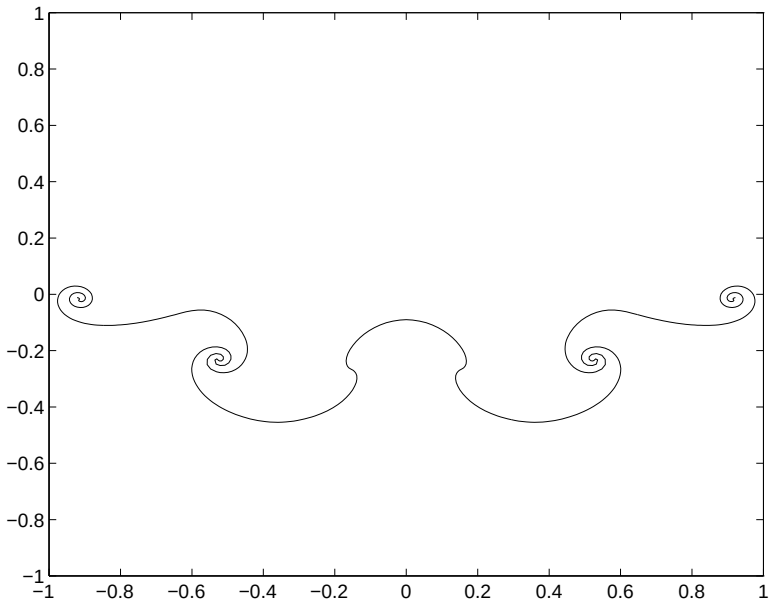



Figure 6.16: Test case 3, the vortex sheet after 100 iterations

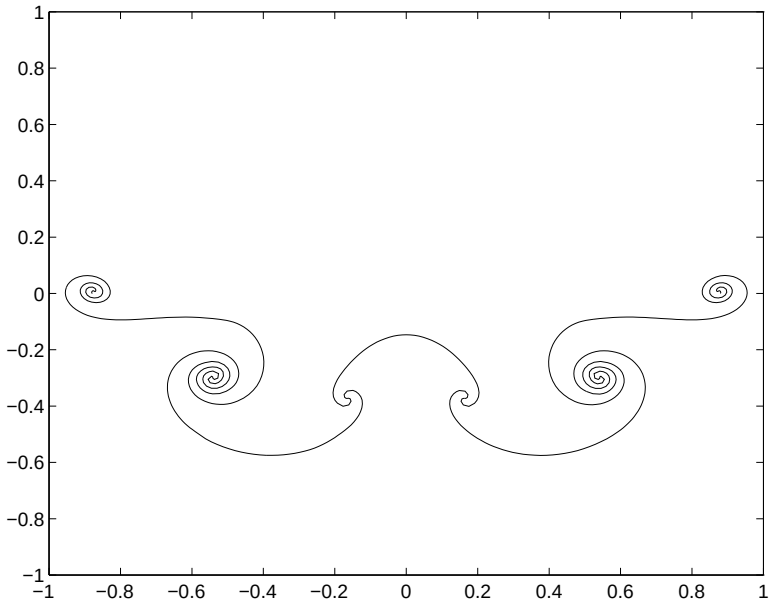



Figure 6.17: Test case 3, the vortex sheet after 150 iterations

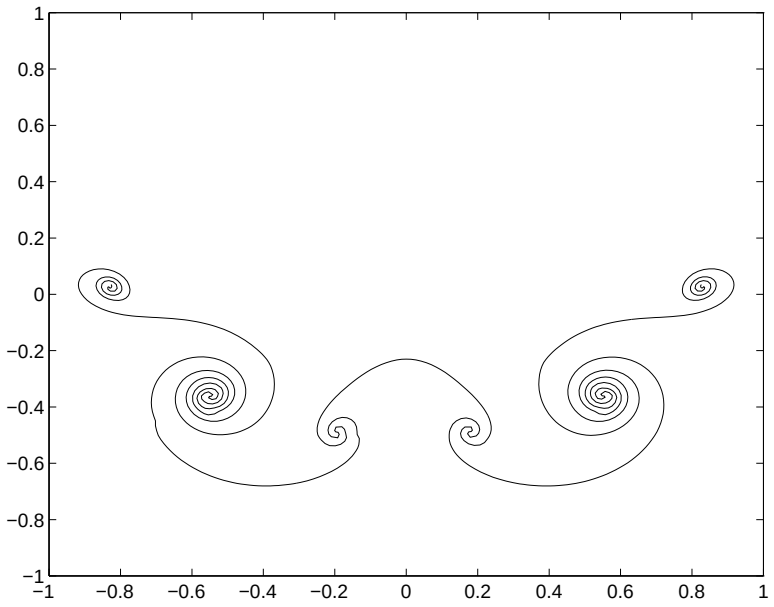



Figure 6.18: Test case 3, the vortex sheet after 200 iterations

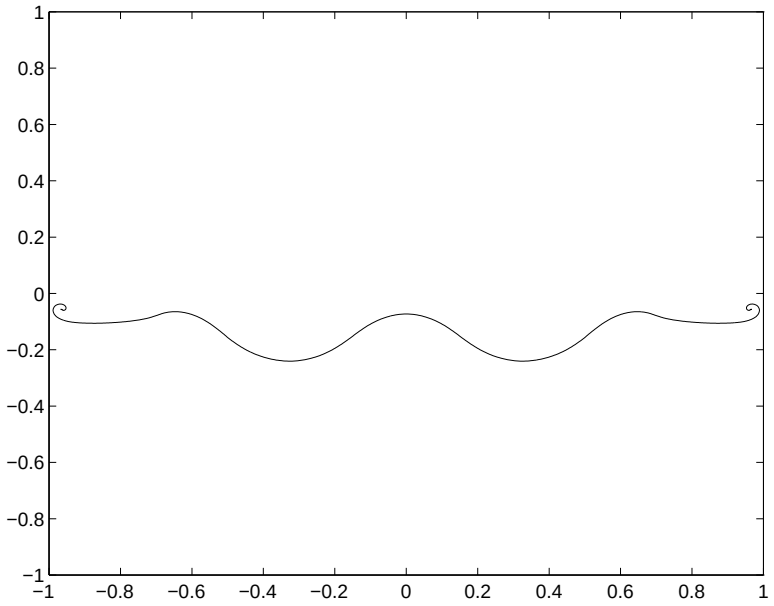



Figure 6.19: Test case 4, the vortex sheet after 50 iterations

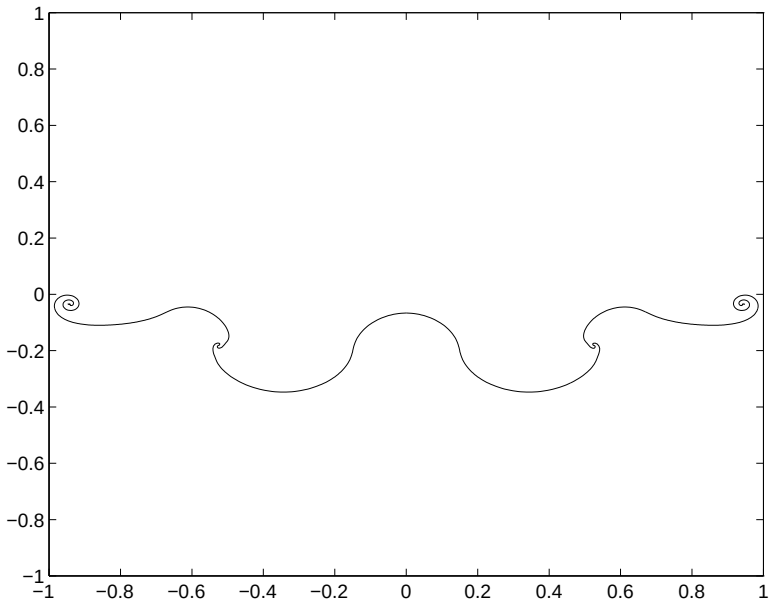



Figure 6.20: Test case 4, the vortex sheet after 100 iterations

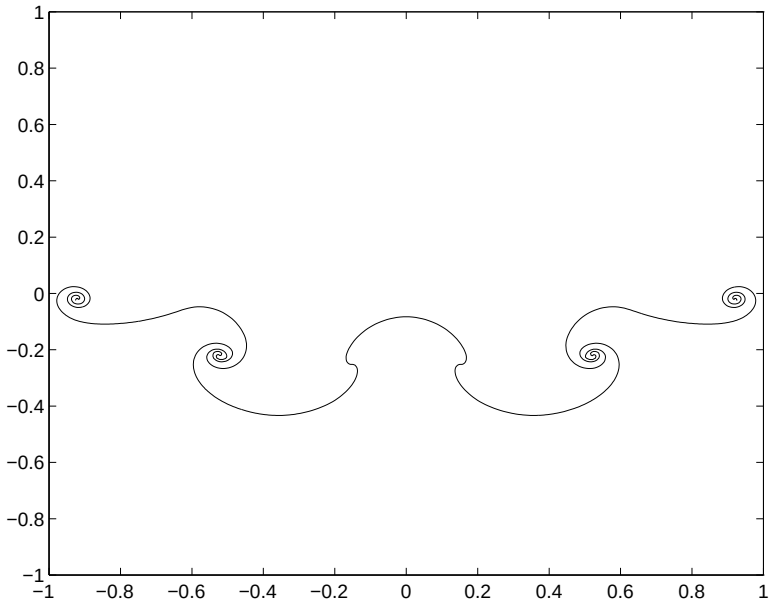



Figure 6.21: Test case 4, the vortex sheet after 150 iterations

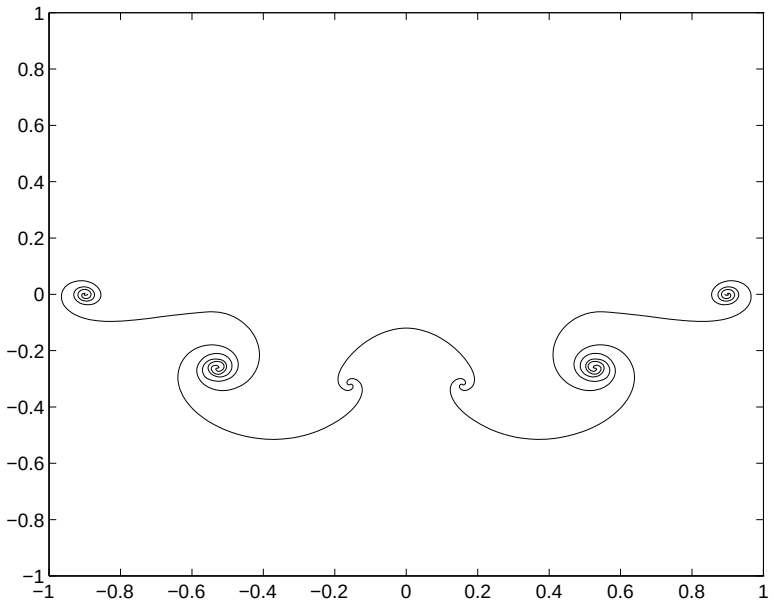



Figure 6.22: Test case 4, the vortex sheet after 200 iterations

# Chapter 7

# Task Graph Structure of the Contour Dynamics Computation and Its Runtime Behavior

Contour Dynamics is another model of fluid flow, besides the vortex sheet model we discussed in the last chapter. The fluid is viewed as interactions between distinct closed contours where there are vorticity jumps across the boundaries. Contours can be imagined as floating in the homogeneous background and exerting influence upon each one, which is an analogous situation to that of the classical N-body problem. Dritschel [18] has developed a fast algorithm using expansions (He named them "moments" instead). The algorithm was inspired partly by the FMM algorithm of Greengard described in [28]. On the other hand there are significant differences because the Dritschel algorithm does not use space partitions. Instead the expansions are centered at each of contours. We study this problem because its task graph structure is different from that of the FMM algorithm, and its dynamic behavior requires a new class of rescheduling algorithm that can not rely on local adjustment. The development of such algorithms remains an open research subject.

## 7.1 Introduction and Problem Definition

Now we consider contours floating in fluid. Take any pairs of contours $\mathcal{C}$ and $\mathcal{C}'$, we need to derive the influence by $\mathcal{C}'$ on $\mathcal{C}$. The velocity field induced by $\mathcal{C}'$ at any point $z$ in the complex plane is:

$$\frac{dz^*}{dt} = q^* = -\frac{\omega'}{4\pi} \oint_{\mathcal{C}'} log|z - z'|^2 dz'^* \qquad (1)$$

Where $z'$ goes through points on $\mathcal{C}'$, $\omega'$ is the vorticity jump on the boundary of $\mathcal{C}'$, and * denotes complex conjugation. If the two contours are well separated, we can

write $q$ in the form of series of expansions around the centroid Z of $\mathcal{C}$:

$$q^*(z) = \sum_{n=0}^{\infty} \beta_n (z - Z)^n \qquad (2)$$

where $\beta_n$ are the coefficients of the expansion to be determined. This form of expansion corresponds to the "local expansion" used in FMM algorithm, where the expansion is centered in the box that the potential field is been **applied to**, as oppose to the multipole expansion, which is centered in the box where the potential is been **generated**.

To arrive at the local expansion terms, we begin from:

$$log|z - z'|^2 = log(z - z') + log(z - z')^* \qquad (3)$$

For the first term, notice $(z - z') = (z - Z) - (z' - Z)$. Assuming well-separated condition $|(z - Z)/(z' - Z)| < 1$ for all $z' \in \mathcal{C}'$, we get:

$$log(z - z') = log[-(z' - Z)] - \sum_{n=1}^{\infty} \frac{1}{n} (\frac{z - Z}{z' - Z})^n \quad (4)$$

Next notice $-(z' - Z) = (Z - Z') - (z' - Z')$ and assume $|(z' - Z')/(Z - Z')| < 1$

$$log[-(z' - Z)] = log(Z - Z') - \sum_{m=1}^{\infty} \frac{1}{m} (\frac{z' - Z'}{Z - Z'})^m \quad (5)$$

Furthermore:

$$\frac{1}{[-(z' - Z)]^n} = \frac{1}{(Z - Z')^n} \sum_{m=0}^{\infty} C_{n-1}^{m+n-1} * (\frac{z' - Z'}{Z - Z'})^m \qquad (6)$$

Here we will see why the expansions are centered at each contour: If we expand $log(z - z')^*$ in the analog fashion, and integrate with respect to $dz^*$ as in (1), these term will vanish because we are integrating along the closed contour curve. This also happens to the constant terms such as $log[(Z - Z')]$. Collecting the remaining terms we get:

$q^*(z) =$

$$\frac{\omega'}{4\pi} \oint_{\mathcal{C}'} [\sum_{m=1}^{\infty} \frac{1}{m} (\frac{z' - Z'}{Z - Z'})^m + \sum_{n=1}^{\infty} \frac{(-1)^n}{n} (\frac{z - Z}{Z - Z'})^n \sum_{m=1}^{\infty} C_{n-1}^{m+n-1} * (\frac{z' - Z'}{Z - Z'})^m] dz'^* \quad (7)$$

Let

$$\alpha'_m = (-1)^m \frac{\omega'}{4\pi m} \oint_{\mathcal{C}'} (z' - Z')^m dz'^* \quad (8)$$

We can say that the $\alpha$'s are "moments" (expansion terms) of $\mathcal{C}'$. They depend only on the shape of $\mathcal{C}'$, so they correspond to the multipole expansions in FMM.

Applying (8) in (7) and comparing with (3), we have the "local expansion" coefficients:

$$\beta_n = \sum_{m=1}^{\infty} C_{n-1}^{m+n-1} \frac{\alpha' m}{(Z' - Z)^{m+n}} \quad (9)$$

The above described the velocity field induced by $\mathcal{C}'$ on any point along $\mathcal{C}$. When implementing the algorithm, we have to truncate the terms to length $p$, instead of using infinite series.

The goal of the algorithm is to reduce the complexity of simulation, just as the FMM algorithm for the N-body problem. However, because no space partition is used, the complexity of this algorithm is different. Let there be $n$ points distributed along $N$ contours (so $N << n$) , and the number of terms used in the expansion is $M$. If the direct method is applied, each pair of point has to be accounted for, resulting in $O(n^2)$ complexity. The moments expansion algorithm will distinguish between far-away and nearby contour pairs. The cost of generating the multipole expansion for each contour is $O(Mn)$, since each point belongs to only one contour. The conversion of far-away multipole to local expansion translations (i.e. calculating $\beta$'s from $\alpha$'s) is done between each pair of contours and the cost is $O(M^2N^2)$. Finally the cost of evaluating the total local expansions at each point is again $O(Mn)$. So the complexity of the algorithm is:

$O(M(n + N^2)) + Cost_{near}$.

Where $Cost_{near}$ is the total cost of computing interactions between contours that are not well-separated. This cost is difficult to bound and in fact can be as high as $O(n^2)$ in bad pathological cases where all contours are closed together. But usually the number of pairs of the near-by points is far less the total number of pairs of points and therefore the total cost of the algorithm improves significantly over the direct method.

## 7.2 Task Graph Structure of the Contour Dynamics Computation

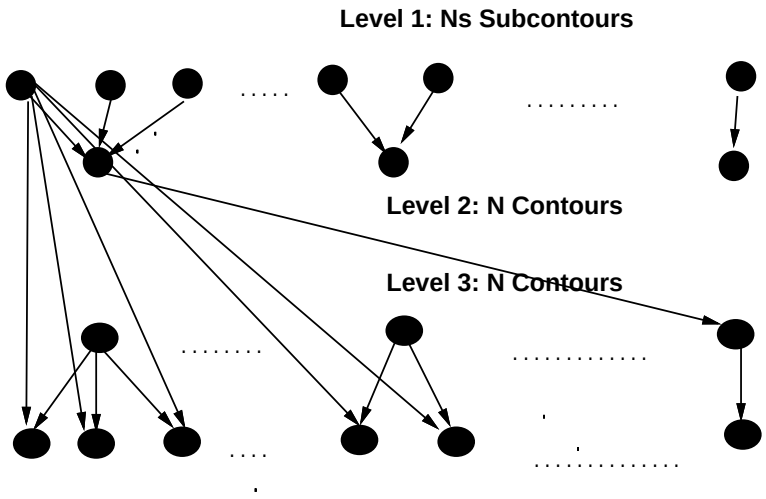



Figure 7.1: The task graph structure of contour dynamics with moments expansions

We now discuss the parallelization of the contour dynamic computation using the task graph scheduling method. First we notice that in a typical computation there may be many points on a few contours, and the number of points on each contour may vary considerably. If we take each contour as a basic unit of the computation node, there may not be enough parallelism to exploit. Therefore we perform the following contour splitting step first:

The splitting will create sub-contours that are parts of the original contours. Choose an integer $P$. If a contour $C_i$ has $n_i \leq P$ points, we leave the whole contour intact, i.e, there is only one sub-contour for this contour. Otherwise we split the contour into $\lceil \frac{n_i}{P} \rceil$ sub-contours so that each sub-contour contains a continuous piece of the contour, i.e. if the points on the original contour are numbered counter-clockwise along the contour as $p_1^i, p_2^i, ...p_{n_i}^i$, then the first sub-contour contains $p_1^i, p_2^i, ...p_P^i$, the next one

contains $p^i_{P+1}, ...p^i_{2P}$ and so forth. The last one may contain less than $P$ points. These subcontours are named $SC^i_1, ...SC^i_{\lceil \frac{n_i}{P} \rceil}$. let $N_s$ be the total number of sub-contours created.

We are now ready to present the task graph after contour split. The task graph consists of 4 levels, See figure 7.1. Level 1 and level 4 each has $N_s$ tasks, each of which corresponds to a sub-contour. For level 2 and 3, there are $N$ tasks at each level corresponding the $N$ whole contours. Each task has the information of the centroid position of the contour it represents, Or the centroid of its parent contour if it falls in level 1 or 4.

1.) Take a node at level 1, assuming it represents sub-contour $SC^i_k$, which belongs to contour $C_i$. The functionalities of this task are:

- Computing the (partial) multipole expansion centered at the centroid $Z_i$ of $C_i$ from the points in $SC^i_k$.
- Sending the (partial) multipole expansion to its parent contour $C_i$ in level 2.
- Sending the points positions to all the sub-contours derived from all contours $C_j$ such that $C_i$ and $C_j$ are not well-separated.

  This includes all the sub-contours derived from the same contour $C_i$.

2.) The function of a task corresponding to contour $C_i$ at level 2 are as follows:

- Receiving the partial multipole expansions from its children sub-contours at level 1 and summing them up, resulting in the total multipole expansion of this contour.
- Sending the multipole expansion to all contours $C_j$ at level 3 such that $C_j$ and $C_i$ are well-separated, together with centroid position $Z_i$.

3.) A task for contour $C_i$ at level 3 does the following:

- Receiving the multipole expansions and centroid positions from each $C_j$ at level 2 such that $C_j$ and $C_i$ are well-separated.
- Translating each of the multipole expansion received into a local expansion around its own centroid $Z_i$, then summing up these expansions, yielding the total local expansion for $C_i$.
- Sending the total local expansion to all tasks at level 4 that represent the children sub-contours of $C_i$.

4.) Finally, a task node at level 4 representing sub-contour $SC_k^i$:

- Receives the point positions from all sub-contours at level 1 that are derived from contours $C_j$ so that $C_j$ and $C_i$ are nearby, including $C_i$ itself.
- Computes the total nearby interaction using the direct method.
- Receives the local expansion of $C_i$ from its parent task at level 3.
- Computes the total faraway interaction using the local expansion, and finally sums up the near and far interactions, obtaining the total velocity field on each point on sub-contour $SC_k^i$.
- Using the velocity field, advances the positions of the points on the sub-contour.

The weights of the tasks are as follows:

- For a task at level 1, assuming there are $p$ points in the subcontour, the cost is $O(Mp)$ where $M$ is the number of expansion terms.
- For a task at level 2, the cost is $O(Ms)$ where s is the number of sub-contours belonging to this contour.
- For a task at level 3, the cost is $O(M^2c)$, where $c$ is the number of contours that are well-separated from this contour. Each translation of multipole to local expansion costs $O(M^2)$.

- For a task at level 4, the cost is $O(p * p_r + p * M)$, where $p$ is the number of points in this sub-contour and $p_r$ the total number of points on the contours that are not well-separated from its parent contour (including itself). This reflects the total cost of near and far interactions.

The need for splitting contours into sub-contours can be seen from the width of the task graph. If we do not perform the split, there will be only $N$ tasks at level 1 or 4, since each contour has only one sub-contour. This will limit the width of the graph and the potential for parallelism. However the splitting parameter $P$ need not to be too small, since a task graph with a width far exceed the number of processors available will only incur more scheduling overhead.

We should elaborate on the "well-separated" condition. Assuming we are using a expansion of order $M$. The truncation error introduced by using only $M$ term grows when the pair of contours $\mathcal{C}$ and $\mathcal{C}'$ move closer, relative to their sizes. Let $R = max|z - Z|, z \in \mathcal{C}$ and $R' = max|z' - Z'|, z' \in \mathcal{C}'$, and $\epsilon$ be the max error tolerance. In [18], a detailed analysis is given, and the well-separate condition is chosen as:

$$\frac{2|\tilde{\omega}|}{M\pi\omega_{max}}\left(\frac{max(R, R')}{|Z' - Z| - R' - R}\right)^M < \epsilon \quad (10)$$

where $\omega_{max}$ is the peak vorticity magnitude in the fluid. Note that this condition will possibly result in an asymmetric situation where $\mathcal{C}'$ is considered well-separated from $\mathcal{C}$ but $\mathcal{C}$ is not well-separated from $\mathcal{C}'$, or vise versa. This is not a desirable case for the task graph generation (although still tractable). But usually in the simulation the vorticity jump values $\tilde{\omega}$ are chosen as unity, i.e. 1, which ensures symmetric relation between any pair of contours.

In the actual implementation the condition is usually rewritten as an inequality about the distance of the centers $|Z' - Z|$. Assuming unit vorticity jump for all contour boundaries ($\tilde{\omega} = 1, \omega_{max} = 1$ ), let $E = (\frac{\epsilon\pi M}{2})^{\frac{-1}{M}}$, A pair of contours with centers $Z$, $Z'$ and radius $R$, $R'$ are well-separated iff:

$$|Z - Z'| \geq (R + R' + E * max(R, R')) \quad (11)$$

It's easy to verify that (10) and (11) are equivalent.

| # proc | Time-run (s) | Speedup |
|---|---|---|
| p=2 | 49.0 | 2.0 |
| p=4 | 25.3 | 3.87 |
| p=8 | 13.3 | 7.37 |
| p=16 | 8.0 | 12.3 |
| p=32 | 4.7 | 20.9 |
| p=64 | 2.9 | 33.8 |

Table 7.1: Speed up for a 50 contour, 2992 points example

We tested the scheduling system for a case with 50 contours and total 2992 points. The results are for 2-64 processors and are listed in table 7.1.

## 7.3 Implementation of Scheduling Method and Dealing with the Dynamic Nature of the Problem

The static structure of the task graph is less complicated than the FMM task graph due to limited levels. However the irregularity is hidden in the more arbitrary well-separated condition: Unlike the space partitioning structure in the FMM algorithm, the contours can be of any shape and placed anywhere in space. But the more challenging aspect of the problem lies in the way the task graph dynamically changes, which is tractable by neither fast schedule readjustment nor the task graph spawning and reschedule heuristic.

Where the task graph is executed iteratively, the contours start to move. But as long as the nearby and faraway lists of each contour remain the same, there is no change in the task graph, and the same schedule can simply be re-applied without any loss in performance. However, when the nearby or faraway list of any contour is modified due to either

1.) A faraway contour moves closer to become a nearby contour.

Or

2.) A nearby contour move away to become a faraway contour.

the task graph structure is altered and rescheduling is mandatory.

If we examining the alteration carefully, we will see that the pattern can be characterized as "edge reshuffling" between levels 1, 4 and levels 2, 3. This is because a contour can exit from the nearby lists of certain group of contours and may join other contours' nearby list, and by the same token, for the faraway lists. The reshuffling may be localized if the change is minor, but in many cases it can be global and involve a large number of contours, and it is not predictable whether an impeding reshuffling event is global or local.

Currently we are resorting to complete reschedule when such reshuffling occurs. It's an interesting opening question whether any lower complexity algorithm can handle edge reshuffling incrementally, based on the original schedule. This seems difficult, because even localized edge change can severely skew the critical path and precedence relations, rendering the original schedule seriously off-balance, or even violating the newly imposed dependences. The reason behind the success of our previous two dynamic support heuristics lies in the incremental nature of the changes in task graph. For the fast re-balancing heuristic, the task graph structure does not change, only the weights. Although the weight changes can result in a migration of the critical path, such drift is often caused by large increase of weights of certain tasks and our heuristic addresses the problem by moving some tasks outside of those processors affected and attempting to reduce their start time. Although it does not imply any performance guarantee, the heuristic behaves well on the average. The second heuristic is even more convincing: If the task graph spawns some new components, and the granularity is not too fine, these new components should be clustered locally and merged into the old schedule. However for the edge reshuffling problem, it's unclear how to derive a schedule based on the old version, since the old schedule may not even stay valid.

Giving the above difficulties, we are more concerned with the overhead paid for rescheduling. The important issue is the number of iterations that a schedule can be recycled.

The situation, unfortunately, does not always work out to our advantage. We found that sometimes a schedule may be reused for tens of iterations, which is a reasonable amount. But for other examples the schedule is applicable for merely several iterations.

The latter occurs when some pairs of contours are near the boundary condition (11), and a small drift of any one of the pair could result in a change of near or far relation between the pair. Worst of all, some pairs of contours may keep oscillating between nearby and faraway status, causing frequent reshuffling in the task graph.

We propose the following "buffer zone" solution to the rapid task graph reshuffling problem. The idea is that once a pair of contours is classified as "nearby", there must be sufficient departure in the distance to make them "faraway". And by the same token, if they are "faraway", a tiny amount of motion can not easily make them nearby. More specifically, let $\sigma < 1$ be a parameter that will determine the buffer zone width. we modify the original well-separated condition (11) as follows. For the sake of simplicity we assume unit vorticity jump for all contours, which is true for almost all simulations.

For any pair of contours $\mathcal{C}$ and $\mathcal{C}'$, we have:

1. If

$$|Z - Z'| \geq (1 + \sigma)(R + R' + E * max(R, R'))$$

   Then we decide $\mathcal{C}$ is well-separated from $\mathcal{C}'$.

2. If

$$|Z - Z'| \leq (1 - \sigma)(R + R' + E * max(R, R'))$$

   Then $\mathcal{C}$ is nearby to $\mathcal{C}'$.

3. If neither of the above is true, i.e:

$$(1 - \sigma)(R + R' + E * max(R, R')) \leq |Z - Z'| \leq (1 + \sigma)(R + R' + E * max(R, R'))$$

Then the pair relationship is in the buffer zone. Our goal is to preserve the status quo longer in order to better amortize the cost of each rescheduling effort.

For the initial condition we may decide $\mathcal{C}$ to be well-separated from $\mathcal{C}'$ using the original inequality (11). But later our rule is that if the distance of the two contours lies in the buffer zone, the pair always retain their current "nearby" or "faraway" status.

So if a pair is classified as well-separated, even if condition 1 is violated they stay well-separated. Only if the movement is so significant that condition 2 becomes true are they reclassified as nearby. Once they are nearby, a slight increase of distance will not make them faraway. Only if condition 1 is satisfied again do they become well-separated.

We hope that the buffer zone can delay the edge reshuffling considerably, and especially prevent the oscillation back and forth between "near" and "far". Therefore it helps to extend the number of iterations that a schedule may be reused. There could be some effect on the truncation error bound. But since the computation is an approximation in any case, and if the buffer zone width $\sigma$ is small, the computation result will not be significantly different from what was obtained by using (11) as the separation criterion.

As a test example, we continue to use the 50-contour input upon which table 1 is based. If no buffer zone is present, i.e. condition (11) is applied, with $\delta t = 0.0001$, the initial schedule can be reused for merely 6 iterations before edge reshuffling occurs.

But if we adopt the buffer zone technique and choose the parameter $\sigma = 0.05$, the number of iterations that the initial schedule remains valid increases to 58. Indeed, for these 58 iterations, if no buffer zone is adopted, we must reschedule for 6 times. We see that even a small buffer zone size can help reduce the reschedule effort dramatically.

Nevertheless, it should be pointed out that if the error bound allowed is very strict, this technique may not satisfy the precision requirement, since the introduction of $\sigma$ only guarantees a larger error bound. To remedy this loss we could reduce $\epsilon$ as a way of compensation. This will reduce the scheduling cost, but the reduction of $\epsilon$ might result in a higher probability of a pair being classified as nearby, and increase the per iteration execution time. So there is a kind of performance vs. overhead trade-off that have not been observed in any other applications studied previously, and we don't believe a definite answer to this problem is available, since the situation is completely input dependent. We can talk about saving in rescheduling cost and possible increase in parallel time only for any given example, but not in a generic sense.

A more satisfying way to handle the reschedule problem, of course, is a low cost reschedule algorithm, which remains an open problem.

## 7.4 Conclusion

We have described the task graph for the contour dynamics computation and the dynamic property that is more elusive than those in the classical N-body or the vortex sheet application. Our result here is more preliminary than the ones in the previous chapters, and important open problems remain. The solution to such problems will greatly extend the practical impact of scheduling to a larger class of dynamic problems, which are still beyond current capabilities of parallel processing research.

# Chapter 8

# Toward Automatic Parallelization of Sequential Code: The PlusPyr /PYRROS Integration

## 8.1 The Goal of Automatic Parallelization and Overview of the PlusPyr System

The PYRROS scheduling and code generation system can exploit parallelism from regular and irregular task graphs. However, it is the user's responsibility to describe the task graph as the input to the system. This has to be done in one of the following ways:

- For regular task graphs, which can be described symbolically in a compact, parameterized form, PYRROS requires that the user write an input file using a kind of task graph language which explicitly specifies the weights and send/receive procedures for each task. See [59].
- For irregular task graphs that can't be parameterized, the user needs to generate the complete task graph from the particular problem and save it in a data file readable by PYRROS. This amounts to designing a task graph generator for each class of computation, such as the FMM algorithm.

In either case, the user has to derive the task graph by analyzing the sequential code or algorithm. For a naive user who is unfamiliar with such a responsibility, this can turn out to be very difficult. Therefore, such requirements could severely limit the widespread usage of scheduling method.

Our ultimate goal is a completely automatic parallelization system. Figure 8.1 illustrates the blueprint of such a parallelization tool. This idealized system is capable

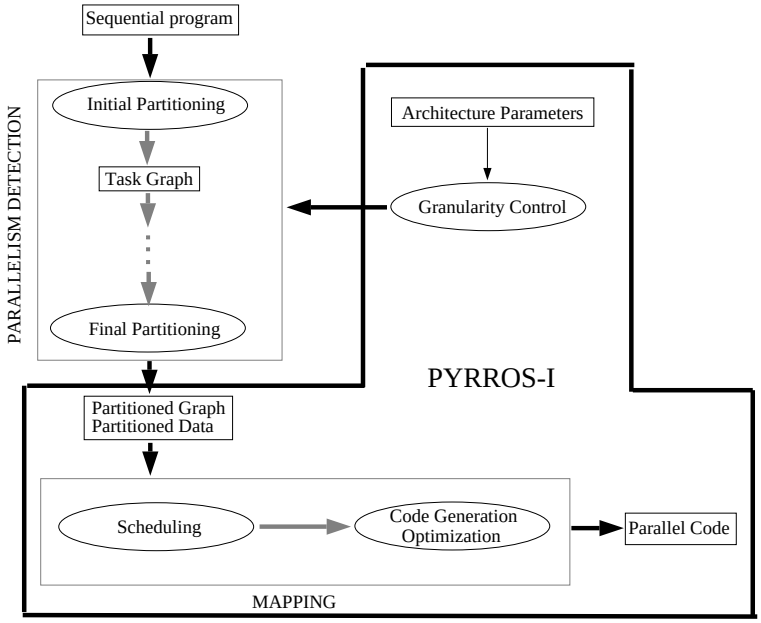



Figure 8.1: System architecture of an automatic parallelizing compiler

of taking sequential code written by a naive user who knows nothing about parallel programming, and carrying out the following procedure without any user intervention:

1. Deciding program partitioning.

   In this step the code is partitioned into tasks. The issue here is the granularity of the tasks. A partitioning that is extremely fine-grain will result in a large fine grain task graph and unsatisfying performance, while a partitioning that is too coarse leads to limited parallelism. This step, currently handled manually, may be done by picking an initial partition and using feedbacks from the scheduling system to adjust the partition until certain conditions are met. These feedbacks may include parameters of the parallel platform and the predicted performance data. The design of such a granularity control mechanism will be a future research topic.

2. Analyzing the computation and communication, producing the task graph.

   Given the task definitions, we need to derive the communication edges between tasks, as well as computation and communication weights. One central issue here is dependence analysis. Currently there are many techniques for statement level

analysis, what remains is to produce task level communication. In addition, there are issues in deriving the weights as well.

3. Task graph scheduling and code generation.

   This step is well studied. PYRROS is such a system for efficient scheduling for both regular and irregular task graphs.

This goal of achieving black box parallelization is really a "Grand Challenge" in the parallel processing area, and is currently beyond our, or anybody's, capabilities. This is particularly true for irregular problems which are so complex that automatic task graph generation is poorly understood. Current research effort, therefore, is starting from regular task graphs, hoping to extend the results to general task graphs in the long term future. Such regular codes usually have affine loop bounds. See e.g. [2, 43].

Even when we narrow down to only regular task graphs, the effort needed is still great. As far as we know no existing system can accomplish completely automatic parallelization even for basic regular task graphs. Step 1 above can be done manually with relative ease but automatic partitioning is very difficult. It will not be discussed in this thesis, rather it's left as a future research direction. We therefore assume manual partitioning and focuses on steps 2 and 3.

For step 2, there have been some studies of automatic extraction of functional parallelism, such as [29]. The PlusPyr ([15]) system, developed in France, is a significant attempt toward the goal of automatic parallelization. It was indeed inspired by PYRROS, with the purpose of providing an automatic task graph generation front-end to PYRROS. Its input is a piece of sequential code written in the Tiny language ([55]), which has PASCAL-like syntax and has been used in dependence analysis research. The basic structures in the code are nested loops, and the system leaves the task partitioning to the user, who may use **TASK** and **ENDTASK** annotation to delimit tasks enclosing certain levels of loop nests.

As a simple example, we list the input code for the Gauss-Jordan algorithm to PlusPyr below:

```
param n
real a(n,n+1)

for k = 1 to n do
    for j = k + 1 to  n + 1 do

      TASK
        a(k,j) = a(k,j) / a(k,k)
        for i1 = 1 to k-1 do
            a(i1,j) = a(i1,j) - a(k,j) * a(i1,k)
        endfor
        for i2 = k + 1 to n do
            a(i2,j) = a(i2,j) - a(k,j) * a(i2,k)
        endfor
      ENDTASK

    endfor
endfor
```

In this example, all loop bounds are affine, and there is a single generic task, or "parametric task" as called in [15], defined to enclose the two innermost loops with indices $i1$ and $i2$. This generic task, delimited by the TASK and ENDTASK annotations, is in turn enclosed by the $k$ and $j$ loops at higher level of the loop nest. Here $n$ is the dimension of the matrix, and serves as a parameter to unwind the symbolic representation into a real task graph.

Let's assume the generic task is named $T1$. For any given value of $n$, there is a total of $\frac{n(n+1)}{2}$ tasks. The tasks are labeled by the enclosing loop indices $k$ and $j$, as $T1(k, j)$, with $k = 1..n$ and $j = k + 1..n + 1$. For more complex examples there can be more than one generic task defined in the code and many more tasks after unwinding using the parameter(s).

Once the task definitions are provided, the system uses the following procedure, as illustrated in figure 8.2, for task graph generation: This is only an outline, details can be found in the original paper [15] written by the French group.

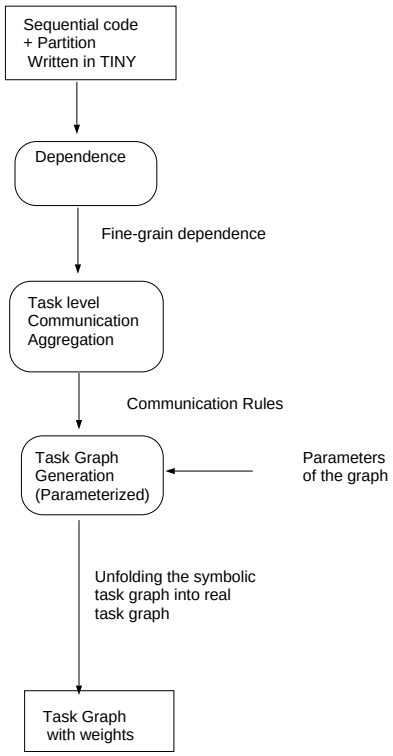



Figure 8.2: Outline of Task Graph Generation Steps in PlusPyr

1. Statement level dependence analysis.

This is done at the individual statement level, without regard to the task definitions. It's a classic problem which has been extensively studied in the literature, and many algorithms have been proposed, such as [57, 54, 42]. The system currently uses the Omega test ([43]), which is an exact algorithm.

2. Task level communication formation.

   After the fine grain dependences are extracted, they are then grouped according to the task definitions to produce task level coarse grain dependences, or communication rules. Such rules, aggregated from finer statement level dependencies, specify send and receive operations between tasks. The grouping of fine-grain dependence is basically as follows: Within each task definition there are certain number of statements. Between a sender task and the corresponding receiver, there can be multiple dependences, according to the result of statement level analysis. The multiple edges must be merged into a single one, which represents the aggregated communication between this pair of tasks. This step is important because 1.) It turns a multi-graph into a real task graph for scheduling purpose. 2.) In practice we always aggregate communication to avoid start-up cost of data transmission .

   Due to the regularity of the program, the communication edges may be represented by symbolic *communication rules*, which will be discuss later in this chapter.

3. Task and edge weight generation.

   This step is also done symbolically in PlusPyr. Task weights are calculated by determining the volume of the polytope in the loop index hyperplane that correspond to a specific task. Communication weights are calculated similarly, by the volume of the polytope in the communication rule. The details are very mathematically involved, and will not be described here. Interested reader may refer to [15].

Such information regarding task, edge weights and communication rules can be described symbolically, without the actual parameters of the task graph, such as the dimension of matrix for linear algebra task graphs in Gauss-Jordan elimination. Later these parameters have to be feed into the system, and a task graph is then produced and can be visualized in the graphic interface.

For the above Gauss-Jordan example, letting $n = 5$, PlusPyr will obtain a task graph shown in figure 8.3. Although the weights are not shown in the figure, they can be viewed using the graph interface, written in the Tcl/Tk script language. The user may also save the task graph, together with the weight information, in the adjacency list form for scheduling purpose.

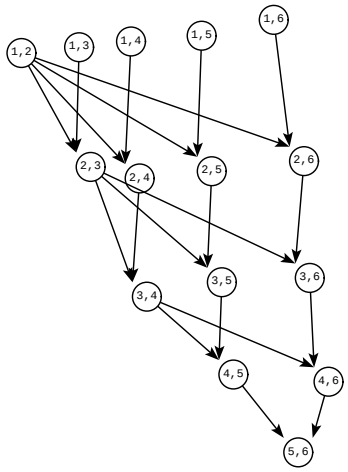



Figure 8.3: The Gauss-Jordan task graph produced by PlusPyr, with $n = 5$

After extensive testing of this system, we appreciate its usefulness and the ability of deriving correct task graphs, which can be highly non-trivial. On the other hand the system is not able to produce executable code, and its output is not directly readable by PYRROS.

Our research effort in this direction therefore, is first to bridge the missing link between PlusPyr and PYRROS, thus connect steps 2 and 3 in the process. With user-provided sequential code and partitioning, the whole process can then be achieved without user intervention. We discuss our work in the following sections.

## 8.2 Basic Architecture of an Integrated Automatic Parallelizing Compiler

As discussed above, PlusPyr can take serial code and produce task graphs. However, a complete parallelizing compiler must be able to produce executable code for real multiprocessors. To achieve this purpose we need the following items, in addition to the task graph.

- Subroutines defining the functions of tasks.

  In a task graph, tasks are represented by nodes with weights. This is sufficient for a scheduler such as PYRROS. But when executing, the task definition code must be available to be invoked by the main program produced by the PYRROS code generator.

- Data items sent along each edge.

  For the same reason as above, edges with weights are sufficient for scheduling purpose but not for execution. We must also know what data are sent along each edge to ensure correctness.

To clarify these issues, we draw the blueprint of the components of the automatic parallelizing compiler based upon PlusPyr/PYRROS integration in figure 8.4. The ovals in the diagram represent functional components and the rectangles are data components being processed and generated.

As we see in the chart, the input source to the entire system is just the same as the input to PlusPyr: Serial code with task annotations, written in TINY language. The major components are, of course, PlusPyr and PYRROS systems. These two alone, however, are not sufficient for execution purpose. The global code produced by PYRROS needs task definition procedures, which are created by the parser scanning serial TINY code and extracting the C subroutines for each generic task. In addition, a Data Item Analyzer will examine the communication rules and the task graph in order to arrive at the precise message content of each edge. Finally, an executor will take all

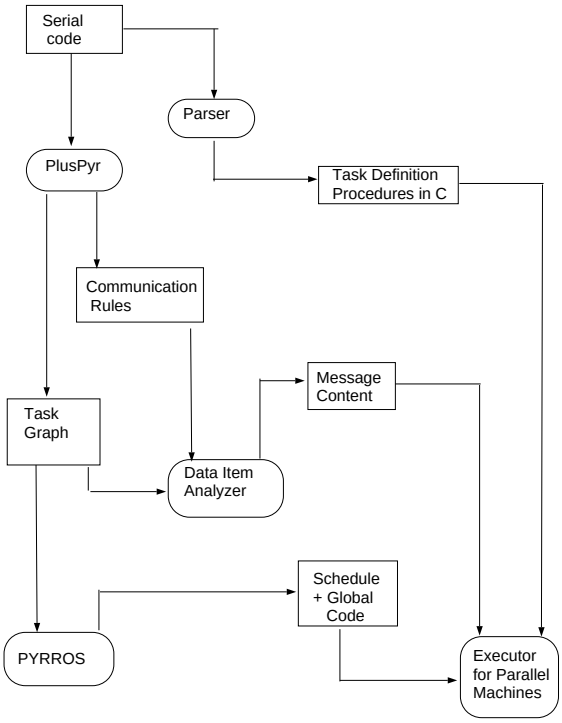



Figure 8.4: The components of the integrated parallelizing compiler

the relevant information into account and carry out a test run on a given number of processors.

We begin building the missing components by dealing with the above mentioned issues separately as below.

## 8.3 Extraction of Task Definition Codes

In the input to PlusPyr, the **task** and **endtask** annotations delimit the body of a task. Therefore, this body should constitute the subroutine defining the function of the task and may be invoked by the main program.

This is basically straightforward. The main issue is that the input to PlusPyr is written in TINY while we need C subroutines for execution. A TINY-to-C converter is constructed in order to parse the TINY code, resulting in a Abstract Syntax Tree (AST) representing the code, and then the portions corresponding to each of the tasks are extracted and the C subroutines produced. The parser is written using Bison.

Stick with the same Gauss-Jordan example and the TINY serial code shown earlier, our parser produces the following task definition code:

```
T1(int tid, int k,  int j, int n )
{
 int i1,i2;

  a[k][j] = a[k][j] / a[k][k];
    for( i1 = 1; i1 <= k - 1; i1++ ) {
          a[i1][j] = a[i1][j] - a[k][j] * a[i1][k];
           }
    for( i2 = k + 1; i2 <= n; i2++ ) {
      a[i2][j] = a[i2][j] - a[k][j] * a[i2][k];
       }
}
```

This is mostly a word-for-word translation of the code inside the task definition. Since there is only one generic task defined, we have only one procedure called $T1$. We will have multiple procedures in case there are more generically defined task.

The arguments the procedure task are explained as follows: $n$ is the dimension of the matrix, which is the parameter of the task graph. After unfolding of the task graph, each task have a numerical id called $tid$, and $k$ and $j$ are the loop variables enclosing the task definition, which uniquely define a real task after the unfolding operation using parameter $n$. So for any real task $T1(k, j)$ with a given $tid$, this procedure will be invoked accordingly, enabling correct execution.

## 8.4 Extraction of Data Content Along Communication Edges

As we mentioned before, PlusPyr does produce task graphs, which include tasks, edges and their weights in the adjacency list form. This task graph is sufficient for symbolic scheduling purpose, but not for code generation. The reason is that we don't know what data items are sent along each of the communication edges. Therefore the data content need to be extracted. We propose the following method, and implement our proposal:

### 8.4.1 The communication rules produced by PlusPyr

First we extract the communication rules from the system. The rules take on the following general form:

$$Sendname(Tasksubscript_1..Tasksubscript_n)$$
$$-->Receivename(Tasksubscript1..Tasksubscript_m):$$
$$Dataname(Datasubscript_1...Datasubscript_k)|$$
$$Lowbound_1, Highbound_1...Lowbound_p, Highbound_p.$$

For example, a communication rule extracted from the Gauss-Jordan sequential code with task definitions is as follows:

$T15(v_0 - 1; v_0)|v_0 - 2; -v_0 + n - 1$
$- - > T15(v_0; y_0)|y_0 - v_0 - 1; -y_0 + n + 1 :$
$a(x_0; v_0)|x_0 - v_0 - 1; -x_0 + n$

Here the names of both send and receive tasks is T15, which is automatically assigned by PlusPyr, and corresponds to the name $T1$ given in the last section.

Now we have the following explanations for this rule, with $v_0$, $x_0$, $y_0$ as variables and $n$ as the task graph parameter, more specifically the dimension of the matrix in this case:

- The sendtask is therefore, $T15(v_0 - 1, v_0)$ and the receiver $T15(v_0, y_0)$. Note that this communication rule also contains lower and upper bounds of the task subscripts : $v_0 - 2 \geq 0$, $-v_0 + n - 1 \geq 0$ and also $y_0 - v_0 - 1 \geq 0$, $-y_0 + n + 1 \geq 0$. (This is equivalent to $2 \leq v_0 \leq n - 1$ and $v_0 + 1 \leq y_0 \leq n + 1$.)

- The data transmitted in this communication is $a(x_0, v_0)$, with lower bound $x_0 - v_0 - 1 \geq 0$ and upper bound $-x_0 + n \geq 0$. These are essentially bounds for $x_0$, since for any particular instance of communication the value $v_0$ is fixed, as we shall see later.

Given a particular communication edge between two tasks, the communication rules described above can be used to extract data content. The methods employed are basically rule matching and variable assignment:

1. Check the task names of the sender and receiver, making sure that they match the ones in the rule.

2. If so, begin assigning variables in the communication rule, based on the subscripts of the given sender and receiver. Check for consistency during this process: If a variable was previously assigned but takes on a different value, it's clear that the rule does not match the communication edge. Also make sure that the task subscripts lie within the

bounds specified by the rule.

3. Based on the variable assignment obtained in step 2, extract the data name, data subscripts, and lower and upper bounds for free variables.

Let's assume $n = 5$ and consider an edge between T15(1,2) and T15(2,5):

Step 1 assures that the task names match. Then the variables are assigned as $v_0 = 2$, $y_0 = 5$. Note that if the sender were T15(1,3) the value of $v_0$ would be inconsistent and so there would be a mismatch between the rule and the edge. We also know that this set of value is within the bounds, which are $2 \leq v_0 \leq 4$ and $v_0 + 1 \leq y_0 \leq n + 1$.

Finally, we discover that the data content for this instance of communication is $a[x_0, 2]$, with bounds $x_0 - 2 - 1 \geq 0$ and $-x_0 + n \geq 0$, which is equivalent to $3 \leq x_0 \leq 5$. So we know the matrix entries sent are $a[3, 2]$, $a[4, 2]$ and $a[5, 2]$, i.e, the part of column 2 in the matrix $a$ from row 3 to row 5.

### 8.4.2 Generation of message id and message content

The above discussion demonstrated the matching of an edge in the task graph and a single rule of communication. In reality, multiple rules can match a single edge. In such cases, we have to loop through all communication rules that match the edge and collect the data content extracted from each of these rules. In the matrix algebra setting, these data can be individual entries of the matrix, partial column, or a block of partial matrix. These data items must be aggregated into one single message for transmission.

For instance, let's consider the same Gauss-Jordan task graph with matrix dimension $n = 5$. The edge from T15(1,2) to T15(2,5) will match 3 different communication rules:

- Rule 1:

  $T15(v_0 - 1; v_0)|v_0 - 2; -v_0 + n - 1 - - > T15(v_0; y_0)|y_0 - v_0 - 1; -y_0 + n + 1 :$
  $a(x_0; v_0)|x_0 - v_0 - 1; -x_0 + n$

  The matrix entries extracted from this rule are a[3,2], a[4,2] and a [5,2].

- Rule 2:

$T15(v_0-1;v_0)|v_0-2;-v_0+n--> T15(v_0;y_0)|y_0-v_0-1;-y_0+n+1:$

$a(x_0;v_0)|x_0-1;-x_0+v_0-1$

Data extracted: a[1,2].

- Rule 3:

  $T15(v_0-1;v_0)|v_0-2;-v_0+n--> T15(v_0;y_0)|y_0-v_0-1;-y_0+n+1:$

  $a(v_0;v_0)|$

  Data extracted: a[2,2].

The result of aggregation will be sending the complete column 2 of the matrix $a$ from T15(1,2) to T15(2,5).

Although in this case the aggregation leads to a continuous chunk of data, we can imagine that for certain task graphs the data items may be fragmented. To handle the communication in a general framework, we introduce the data structures of task list and message list:

**Task List:**

This is simply a task number/task name lookup table. For scheduling purpose the task graph is described in a way such that all tasks are numbered 1 through $V$. But task names and subscripts are needed for the sake of execution and also the above communication data extraction preprocessing. So we establish this list.

**Message List:**

This list includes the complete information of communication data content, so that the sender/receiver can pack/unpack the message buffer based on the entry about a specific message, and move data correctly across processors even if the data contained in the message is fragmented.

We first define data fragments. A *data fragment* is a piece of data extracted by the matching of a specific task graph edge/communication rule pair. For example, the partial column of length 3, $a[3,2]$ to $a[5,2]$ generated by the matching in the Gauss-Jordan example is considered one data fragment. Data fragments are ordered in a list

as they are generated by the matching process, and duplicated entries are eliminated, since different pairs of communication rule/edge matching could result in the same data content, but duplicated communication should be avoided.

On the higher level are the *messages*. A message is the total set of data items transmitted along an edge in the task graph. For instance, the whole column 2 in the Gauss-Jordan example is treated as one message, One could therefore see that a message is consists of one or more data fragments. Different messages may share certain data fragments, but as long as they are not identical they should be given different message id's . Therefore, for each edge, after all its data fragments have been identified, we set links from this message to all the data fragments that belong to it, and assign a new message id, unless this message has exactly the same components as a previous message, which could happen when there is a broadcasting in the task graph.

After building the task and message lists in this matching and grouping procedure, we can ensure that the right data content arrives at each task during actual execution as follows:

1. The code generator produces an SPMD code based on the schedule, which contains task numbers and their mapping and ordering. When executing each task, the Task List is used to map task numbers to task names, and task names are then used to invoke the proper subroutine with the right parameters. These subroutines were the definition of tasks within the TINY input code. We have developed a TINY-to-C converter which can produce C subroutines defining the functions of each generic task, and append these subroutines to the main body of code produced by PYRROS for execution purpose.

2. During communication, each edge has been assigned a message id, which is known to the executing tasks. The sending task will check the message list, obtaining the links to all the data fragments contained in this message. It then starts to pack the message buffer by going through all these related data fragments. For each of them, it looks up the data fragment table (which is part of the message list), maps the data fragment id to the real data content, and appends the data

to the buffer. Finally the whole aggregated buffer is transmitted. Similarly, for the receiving end, it is message buffer unpacking based on the message id: The table lookup will identify the data fragments for this message, and extracts the correct data items from the homogeneous message buffer in the same order as the packing was done at the sender. It does not matter whether the total data content forms a regular shape, such as a column of the matrix, in all cases the right data will be transmitted to the right task.

## 8.5 Automatic Generation of I/O Procedures and Other Issues

The above procedure enabled us to generate message content correctly, without user intervention. However, to carry out a real execution on parallel machine, the initial data must be distributed to appropriate tasks, and the end result must be collected and the output displayed. The task graph itself does not contain such information, and in the original PYRROS, the user must write programs to initialize and collect data.

PlusPyr proposes a way to handle this problem, by extending the task graph. An *Extended Task Graph* is a task graph with two additional nodes: the Input Node (IN) and the Output Node (OUT). They essentially act as host nodes: the IN node contains the entire initial data set, and the OUT node collects all of the updated data set. By inserting these two tasks into the code being screened by the dependency analyzer, the desired I/O dependencies will be created, and additional edges will appear in the Extended Task Graph between IN/OUT nodes and the normal task graph nodes.

For the same Gauss-Jordan example, with $n = 5$, the Extended Task Graph generated by PlusPyr is shown in figure 8.5.

Given an Extended Task Graph, the matching/grouping procedure can be enhanced to work on this task graph and handle the additional edges from IN to other nodes or from other nodes to OUT. The former case indicates a send of initial data, while the latter represents the acceptance of final result. The messages could be numbered along with all the others within the task graph in a uniform scheme. As long as we know the symbolic names of the IN and OUT tasks, we can distinguish between a Host/Node

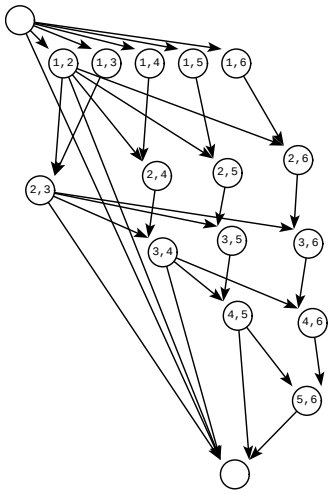



Figure 8.5: The Extend Task Graph for the Gauss-Jordan example

communication and a normal send, and the preprocessing will produce lists of Host send/receive, in addition to the Task and Message Lists.

There are other issues that we must overcome to produce a fully automatic parallelizer. The most important one is that the granularities of the task graph must be sufficiently large in order for us to obtain any reasonable performance. Loops containing simple statements operating on individual matrix entries generally is very fine-grain, so we need to increase the chunk size of the computational units.

We want to be able to add certain loop transformations such as tiling (or strip-mining) to the system. Tiling is a useful loop transformation technique to expose more parallelism. It can also be used to increase the granularity of the tasks. Tiling is extensively employed in the SUIF compiler. See [2, 54] for detailed discussions about titling transformation. Here we just give a simple example to illustrate how the granularity may be increased. Take a double loop with task defined on the inner statement: (For the sake of simplicity we set $a[i1, i2] = 0$ if either $i1$ or $i2$ is negative.)

```
for i = 1 to n do
   for j = 1 to n do
```

```
    TASK
       a(i,j) = (a(i-1,j)+a(i,j-1)+a(i+1,j)+a(i,j+1))*0.25
    ENDTASK

    endfor
  endfor
```

The generic task is very fine-grain. However if we tile the loop with tile size $m$ such that $m * N = n$ we can revise the code as:

```
for ki = 1 to N do
    for kj=1 to N do

    TASK
           for i =(ki-1)*m+1 to ki*m do
             for j=(kj-1)*m+1 to kj*m do
            a(i,j) = (a(i-1,j)+a(i,j-1)+a(i+1,j)+a(i,j+1))*0.25
           endfor
        endfor
    ENDTASK

    endfor
 endfor
```

The task defined now is much more computationally intensive and the total task graph size is reduced by a factor of $m^2$.

We would like to make use of the automatic tiling component of SUIF and enhance the PlusPyr system. We have previously tried to feed the tool with manually tiled loops, but the parametric linear integer programming (PLIP) engine in PlusPyr can

not handle them. The reason is not yet entirely clear, but most likely due to the complexity of the algorithm, both in time and space. So another direction is to explore other state-of-the-art dependence analysis engines.

Nevertheless, we can use a rather simple approach of blocking matrix entries, i.e. defining the operations on submatrices. Note that for block algorithms, some extensions to the system are needed. We take the Gauss-Jordan example. If the basic unit of data is preset to be $r * r$ submatrix, then in the code:

```
for k = 1 to n do
    for j = k + 1 to  n + 1 do

      TASK
        a(k,j) = a(k,j) / a(k,k)
        for i1 = 1 to k-1 do
            a(i1,j) = a(i1,j) - a(k,j) * a(i1,k)
        endfor
        for i2 = k + 1 to n do
            a(i2,j) = a(i2,j) - a(k,j) * a(i2,k)
        endfor
      ENDTASK

    endfor
endfor
```

We can define each element $a[i][j]$ as an $r * r$ submatrix, and all the operations are now defined on matrices. The system is able to generate the following node program for this genetic task based on the serial code:

```
a[k][j] =  matmult( a[k][j], matinv( a[k][k] ) );
```

```
for( i1 = 1; i1 <= k - 1; i1++ )
   a[i1][j] =  matsub( a[i1][j], matmult( a[k][j],a[i1][k] ) );
for( i2 = k + 1; i2 <= n; i2++ )
   a[i2][j] =  matsub( a[i2][j], matmult( a[k][j],a[i2][k] ) );
```

Here *matinv*, *matmult* and *matsub* are matrix inversion, matrix multiplication, and matrix subtraction, respectively. The block size $r$ can be predefined in the header file for compilation.

The reason why block algorithms are considered is because of the poor granularity of non-block version of the algorithms. With submatrix as the basic data unit, the computation to communication ratio can be improved drastically, while the data dependence relations remain the same, with no complications to the PlusPyr part of the system, except that the communication unit also becomes a block of data instead of a scalar data item. Block algorithms are widely used in linear algebra computation, including many packages such as BLAS-3 ([16]).

## 8.6 Current Results and Future Directions in the Automatic Parallelizing Compiler Project

With these features, we have achieved the integration of PlusPyr/PYRROS tools and an automatic procedure from sequential code with task annotations to executable parallel code. The new system, which we call PYRROS+, has been tested on certain representative problems, such as Block Gauss-Jordan algorithm and Block LU decomposition algorithm. For these problem, we have achieved fully automatic parallelization and very good speedup that are competitive to even the hand-optimize parallel code specifically design for these problems. In tables 8.1 and 8.2 we listed some performance data for 1000*1000 dense matrix, with block size $r = 10$, which makes the width of the task graphs for both GJ and LU exceed the number of processor available, which is 64. The sequential running time for one node of the Ncube2s are about 1150 seconds for Block GJ and 800 seconds for Block LU.

| # Proc | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|
| Time(s) | 288 | 159 | 87.4 | 51.0 | 37.1 |
| Speedup | 4.0 | 7.2 | 13.2 | 22.5 | 31.0 |

Table 8.1: The performance of the integrated system on Block GJ algorithm with $n = 1000$ and $r = 10$

| # Proc | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|
| Time(s) | 211 | 116 | 66.6 | 44.9 | 31.1 |
| Speedup | 3.8 | 6.9 | 12.0 | 17.8 | 25.8 |

Table 8.2: The performance of the integrated system on Block LU algorithm with $n = 1000$ and $r = 10$

Figure 8.6 illustrates the speedup results for the Block GJ algorithm. It can be seen that the performance obtained from PYRROS+, a fully automatic system, is almost as good as PYRROS, which requires user-supplied task graphs and communication contents. The performance is also comparable to very good hand-made parallel code.

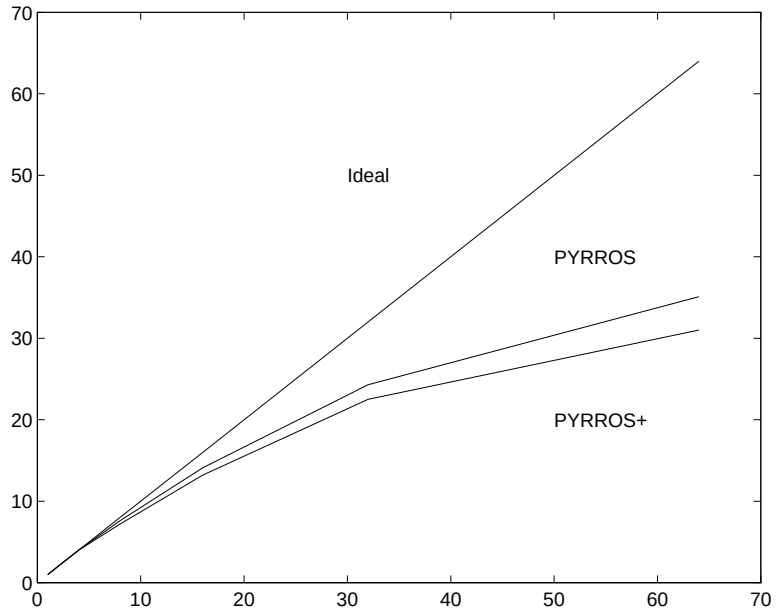



Figure 8.6: Speedup for Block GJ with $n = 1000$ and $r = 10$

Future research directions in this area includes:

- Incorporating other dependence analysis algorithms into the PlusPyr component of the software tool. Conceivably, some low complexity algorithms, such as the

one proposed by Psarris in [42], can be used.

- Adding loop transformations, especially automatic tiling into the PlusPyr front-end. Although block operations can increase the granularity, this approach does not apply to all problems. We should count on tiling to be a more general to create coarse-grain tasks.

- Applying the unified PlusPyr/PYRROS tool to certain Benchmark problems, in order to demonstrate the feasibility of automatic task graph generation and scheduling.

For more long term goals, we can state :

- Development of automatic dependence analysis and task graph generation techniques for irregular problems. This undertaking is considered a grand challenge, and one may hope to start from certain special cases. Regular affine loop nests can be considered a very restricted case, can the result be extended to a larger class of problems?

- Automatic partitioning of sequential code. This involves choosing a suitable granularity and defining task bodies. Currently there is no system available, although there have been some research effort, such as [38]. We believe that although optimal partitioning is difficult to achieve, there can be methods that choose a reasonable task granularity and find the appropriate loop levels to define tasks. These methods may need feedbacks from the scheduler in order to adjust the granularity of the partitioning based on the predicted performance.

# Chapter 9

# Conclusions and Related Work

Throughout this thesis, we have investigated many aspects of the efficient parallelization of irregular/dynamic problems, including initial mapping of computation onto multiprocessors, dynamic adjustment of the mapping given the changes in underlying structures, and various issues in applying these techniques. Our research has provided strong evidence for the success of the combined *Static Scheduling/Dynamic Rescheduling* paradigm. On another front, our effort on automatic parallelization of serial code is a positive step forward, and is related to our main theme. Although current state-of-the-art techniques can only handle regular problems, the implication for irregular problems is significant.

## 9.1 Related Works

Irregular problems have not been understood as thoroughly as regular problems due to their inherent complexity, but recently this topic has become one of the most important research subjects in parallel computing.

The Sparse Matrix Computation is one of the most widely investigated problems, and it's notorious difficult to obtain performance for this class of problems. Many methods has been proposed. See for example, work done by Venugopal [51], Rothberg [44]. One of the noteworthy works, which is very closely related to our approach, is done by Chong et al [11], where the DSC clustering algorithm from PYRROS was applied to the sparse triangular solution iterative problem. In that paper, it was shown that the DSC algorithm can improve the parallel time per iteration by almost 75% over cyclic mapping. And the issue of performance/overhead trade-off also exist in that problem, although there is no dynamic change involve since the task graph is static

over all iterations. Because the computation cost per iteration is tiny compare to the scheduling cost (which implies that the graph is fine grain), up to 1000 iterations are needed to amortize the initial penalty of scheduling. In our case, the cost of computation per time step in the FMM N-body and its variants in CFD is on the same order as the initial scheduling cost so it takes much less time steps to recover.

The N-body simulation has also been of long-standing research interest. Most of the parallel implementations has been for the Barnes-Hut algorithm [5] due to its relative simplicity and reasonable complexity, but efforts are also been made on the Fast Multipole Algorithms. Singh [48] has proposed a very problem-specific technique called "cost-zone", which can be applied to both Barnes-Hut and FMM algorithms. He also employed the same argument about the slow and gradual movement of the particles to justify his approaches. On the other hand, the "cost-zone" partition is essentially a more involved load balancing, and to our knowledge our thesis work is the first one to model the FMM algorithm as a task graph and solve the problem in an automatic scheduling/rescheduling framework.

The CHAOS/PARTI system [34] deals with irregular DOALL loop parallelism. It adheres to the data parallelism model and tries to optimize the array partitioning in order to reduce execution time. The Inspector/Executor approach is used to capture the input-dependent nature of irregular problems, and is similar in spirit to our data structures used for the adaptive FMM algorithm.

The Fortran D compiler [1] has also been extended in an attempt to deal with irregular problems, in the thesis of Reinhard von Hanxleden [30]. The Inspector/Executor model is used for generating input-dependent data mapping.

There are many other heuristics for task graph scheduling in the literature. PYRROS belongs to the class of multistage methods, which also includes the Sarkar's algorithm [47], among many others. Some other methods are one-stage, in the sense that the schedule is produced directly for the given number of processors. The ETF algorithm [33] is one of the most efficient among one-stage methods. One of our papers [24] shows that ETF can be very efficient with small or moderate number of processors but its complexity can be prohibitive for large number of processors. In comparison, PYRROS

performs competitively and its complexity is not seriously affected by the number of processors, since the clustering stage does not limit the processor number.

Runtime support for dynamic problems has been traditional studied in a framework of load balancing strategies, where the works assigned to processors are not assumed to be interdependent as in a task graph. The amount of literature for load balancing is very large, such as [35, 53], among many others. But these are not directly applicable to our model. We are the first to propose dynamic support algorithms for task graphs. In fact, we can call our schedule readjustment algorithm for the changes in task weights "Generalized Load Balancing", since it attempts load balancing while taken into account the communication edges. A related work at MIT by Leiserson's group on the run-time system Cilk [7] also studies "dynamic spawning", but their model is the multithreaded computation, and the spawning is referring to the dynamic creation of threads, which is a realistic description of shared memory multiprocessing adopted by many SMP workstation platforms, but different from our basic model.

On the topic of automatic parallelizing compiler, many systems have been developed, including, but not limited to, the Fortran D compiler [1] at Rice, which is based on data parallelism model and carries out optimizations on data partitioning and layout. The SUIF compiler [2] at Stanford, which involves fine grain dependence analysis for regular loops and certain simple scheduling techniques for processor mapping. For coarse-grain level, they have suggested inter-procedural analysis [31], which is still an open research area. The Paradigm system [4] developed at Illinois is also able to perform dependence analysis and data partitioning. Recently the model of hierarchical task graph has been proposed in [41], which allows data parallelism within tasks, in an attempt to combine data and task parallelism. Other works on generalized task graph models include scheduling for iterative task graphs, as presented in [64], where cross-iteration dependences exist and the scheduling is done symbolically without unwinding of the iterations. Most recently in [10], Chakrabarti et. al. consider global communication analysis in the High Performance Fortran compiler, which can reduce message passing overhead. Their work may also be useful for coarse grain dependence analysis required by our functional parallelism model.

## 9.2 Future Directions

The following are some conceivable directions for future work. Since the parallelization of irregular/dynamic problems is such a challenging subject, we expect that many aspects of the results in this thesis can be extended. But we will just mention the following:

- Application of scheduling paradigm to other real-world complex computations. We have demonstrated that our paradigm can be beneficial for these problems, so we expect it can explore parallelism from more applications. For some, such as the 3-D FMM algorithm, the challenge lies in the implementation of the complex computations within the tasks; For others, such as the sparse matrix computations, parallelization issues are more vital.

- Implementation of the scheduling/rescheduling tool across more platforms. Our results are mainly obtained on the Ncube2 system. Recently, message-passing library such as PVM and MPI ([50, 52]) have gained acceptance in many machine platforms, such as Cray T3D and IBM SP2, as well as the Network of Workstations (NOW) environment where the packages originated.

  We have been working on a PVM version of the scheduling system. For the tightly-coupled multiprocessor systems such as Cray T3D, the issue is mainly coding, although attention should be paid to the interconnection network, which could be different from the hypercube architecture employed by the Ncube and Intel machines. (For example, the Cray T3D uses 3-D torus.) For distributed networks, however, obtaining good performance is much more difficult. This is due to the time-sharing nature of the operating system, the unpredictability of network traffic, and the relatively large communication latency. For such systems, fine granularity will result in poor scalability, and the scheduling algorithm itself may not work well due to the unpredictable weights, which depend on such factors as processor and network loads at a particular point in time. The challenge here is to design a system that produces code that is efficient under "average" or "normal"

conditions. If a dedicated workstation cluster is available, the unreliability can be reduced and existing techniques may apply.

- Development of new and improved rescheduling algorithms.

  Dynamic rescheduling for task graphs is an entirely new subject. In this thesis, two fast rescheduling heuristics are developed and implemented, and have been demonstrated to perform competitively for random and real task graphs. On the other hand, there is considerable room for improvement. First, these existing algorithm do not have theoretical performance bounds except for very restricted cases, so it will be interesting to see whether there are some other methods which do attain nice bounds. Second, for fine grain graphs, the existing heuristics can perform poorly, due to the fact that for such graphs localized clustering and incremental methods could derive scheduling far inferior to the optimal. Another potential problem in the fast schedule readjustment algorithm to handle weight alterations is that in some pathological cases with bounded number of processors, even if all the conditions are satisfied, merging the newly moved task chain with the lightest loaded processor is not necessarily the best choice, and could even lead to parallel time increases. The same problem is actually inherent in the PYRROS cluster merging step, since it's difficult to know whether such merge based on cluster load is good. On the other hand, such load-based merge has low complexity and performs well on average cases, so it's widely adopted. See [19].

  It will be very exciting if any of these problem can be overcome by some improved algorithm, even if it's still of heuristic nature.

  Another aspect of this direction is the development of fast algorithms for more complex and global changing patterns, like the one discovered in chapter 7 in the contour dynamics computation. As long as the complexity of the algorithm is lower than scheduling from scratch and performs at the same level, it should be put into use.

- Continuing effort toward the grand challenge goal of automatic program parallelization.

The future prospects of this work have been discussed in the last chapter concerning PlusPyr/PYRROS integration. In the short term, we have planned to incorporate loop transformations in the front-end. Coarse-grain dependence analysis theory is another interesting line of work.

It should be noted that, although the existing parallelization systems are far from adequate to handle real world code, they can still be used to model the program execution and guide our parallelization effort.

One example is a large scale industrial code called Lamp, described in [25], which is designed to simulate the motion of ships in sea waves. It's impossible for the PlusPyr/PYRROS integrated system to take the real code as input. But if one studies the subroutines of the program, it's possible to model the program execution using simple pseudo-code, which is readable by the system. The tool is then able to create and display a task graph that is devoid of low level details but used as a profile of the components in the program. This skeleton task graph can then guide the redesign of the code for efficient executing on multiprocessors. In the case of Lamp, the skeleton task graph exhibits fork-join patterns in several most computationally intensive subroutines, leading to the decision of assigning the concurrent tasks onto processors in a balanced fashion. In practice, we used the PVM message-passing protocol for execution on workstation clusters and Cray T3D, and almost linear speedups were achieved on the Cray T3D machine. This work is not discussed in detail in this thesis because it is theoretically very straight-forward, but is of considerable interest in the sense of application of High Performance Computing to industrial problems.

# References


[1] V. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C Koelbel, U. Kremer, J. Mellor-Crummey, C-W. Tseng and S. Warren. Requirements for data-Parallel programming environments. *IEEE Parallel and Distributed Technology*, Vol. 2, No. 3, pp. 48-58, 1994.

[2] S.P. Amarasinghe, J. M. Anderson, M.S. Lam and C. W. Tseng, The SUIF compiler for scalable parallel machines. *Proceedings of the seventh SIAM conference on parallel processing for scientific computing*, 1995.

[3] V. Balasundaram, A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *Journal of Parallel and Distributed Computing*, Vol. 9, No. 2, pp 154-170, June 1990.

[4] P. Banerjee, J. A. Candy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy and E. Su, The PARADIGM compiler for distributed-Memory multicomputers. *IEEE Computer,* Vol 28, No. 10, pp 37-47, 1995.

[5] J. Barnes and P. Hut, A hierarchical $O(N \log N)$ force calculation algorithm, *Nature* Vol. 324 P. 446 , 1986.

[6] S. Bhatt, P. Liu, V. Fernandez, and N. Zabusky, Tree codes for vortex dynamics: Application of a programming framework. *Proc of the 1st Workshop on Solving Irregular Problems on Parallel Machines, International Parallel Processing Symposium*, Santa Barbara, CA 1995.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K.H. Randall and Y. Zhou, Cilk: An efficient multithreaded runtime System. *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, pp. 207-216*, Santa Barbara, CA, 1995.

[8] S. Bokhari, On the mapping problem, *IEEE Trans. Comput.* C-30 3(1981), 207-214.

[9] D. A. Case, Computer simulation of protein dynamics and thermodynamics, *Computer*, pp. 47-57, Oct. 1993.

[10] S. Chakravarti, M. Gupta, J-D Choi, Global communication analysis and optimization, *1996 ACM conference on Programming Language Design and Implementation*, pp. 68-78, Philadelphia, PA. May 1996.

[11] F. T. Chong, Shamik D. Sharma, Eric A. Brewer and Joel Saltz, Multiprocessor runtime support for fine-grained irregular DAGs, Draft, 1994.

[12] F. T. Chong and R. Schreiber, Parallel sparse triangular solution with partitioned inverses and prescheduled DAGs, Tech Report, 1994. Appeared in *Proc. of 1st IPPS Workshop on Solving Irregular Problems,* Santa Barbara, 1995.

[13] Ph. Chretienne, Task Scheduling over Distributed Memory Machines, *Proc. of Inter. Workshop on Parallel and Distributed Algorithms*, (North Holland, Ed.), 1989.

[14] E.G. Coffman and R.L. Graham, Optimal scheduling for two-processors systems, *Acta Informatica*, 3 (1972), 200-213.

[15] M. Cosnard and M. Loi, Automatic task graph generation techniques. *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, Vol II, pp 113-122, 1995.

[16] J. J. Dongarra, J. D. Croz, I. Duff and S. Hammarling, A set of Level 3 Basic Linear Algebra Subprograms, *ACM Trans. on Math. Software,* 16(1), 1-17, March 1990.

[17] C. I. Dragchicescu, An efficient implementation of particles methods for the incompressible Euler equations. *SIAM Journal on Numerical Analysis,* Vol. 31, No. 4, pp. 1090-1108, August 1994.

[18] D. G. Dritschel, A fast contour dynamics method for many-vortex calculations in two-dimensional flows. *Physics of Fluids A 5(1)*, 1993.

[19] A. George, , M.T. Heath, and J. Liu, Parallel Cholesky factorization on a shared memory processor, *Lin. Algebra Appl.,* 77(1986), pp. 165-187.

[20] A. Gerasoulis, J. Jiao and T. Yang, Scheduling of structured and unstructured computation. DIMACS book series Vol. 21, pp. 139-172.

[21] A. Gerasoulis, J. Jiao and T. Yang, A multistage approach for scheduling task graphs on parallel machines. DIMACS book series Vol. 22, pp.81-103.

[22] A. Gerasoulis and I. Nelken, Static scheduling for linear algebra DAGs. *Proc. of 4th Conf. on Hypercubes*, Monterey, Vol. 1, 1989, 671-674.

[23] A. Gerasoulis, J. Jiao and T. Yang, Scheduling of structured and unstructured computation. *Book Series on Interconnection Networks and Mapping and Scheduling Parallel Computations*, American Mathematics Society,1995.

[24] A. Gerasoulis, J. Jiao and T. Yang, A multistage approach to scheduling task graphs. *Book Series on Parallel Processing of Discrete Optimization Problems*, American Mathematics Society,1995.

[25] A. Gerasoulis, J. Jiao, W. Lin, Parallelization of a ship motion simulation system, Under preparation. Results presented at *1995 International Mechanical Engineering Congress*, San Francisco, CA.

[26] A. Gerasoulis and T. Yang, A comparison of clustering heuristics for scheduling DAGs on multiprocessors, *J. of Distributed and Parallel Computing,* special issue on scheduling and load balancing, Vol. 16, No. 4, pp. 276-291 (Dec. 1992).

[27] A. Gerasoulis and T. Yang, Performance bounds for parallelizing Gaussian-Elimination and Gauss-Jordan on message-passing machines, *Applied Numerical Mathematics Journal,* 16(1994) 283-297.

[28] Leslie Greengard, The rapid evaluation of potential fields in particle systems, Ph.D thesis, Yale University, 1987.

[29] M. Girkar and C. Polychronopoulos, Automatic extraction of functional parallelism from ordinary programs, *IEEE Trans. on Parallel and Distributed Systems,*. Vol. 3, No. 2, pp. 166-178, 1992.

[30] Reinhard von Hanxleden, Compiler support for machine-independent parallelization of irregular problems, Ph.D. Thesis, Rice University, Dec. 1994.

[31] M. W. Hall, B. R. Murphy, S. P. Amarasinghe, S. Liao, and M. S. Lam, Interprocedural analysis for parallelization, *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing (LCPC95)*, August 1995.

[32] P. Havlak and K. Kennedy, An implementation of interprocedural bounded regular section analysis, *IEEE Transaction on Parallel and Distributed Systems*, Vol 2, No.3, pp. 350-360, 1991.

[33] J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee, Scheduling precedence graphs in systems with interprocessor communication times, *SIAM J. Comput.,* pp. 244-257, 1989.

[34] Y.S. Hwang, B. Moon, S. Sharma, R. Das, J. Saltz, Runtime support to parallelize adaptive irregular programs, *Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing*, 1994.

[35] J. D. Keyser and D. Roose, Load balancing data-parallel programs on distributed memory computers, *Parallel Computing*, Vol. 19, No. 11, pp. 1199-1219, Nov. 1993

[36] S.J. Kim and J.C Browne, A general approach to mapping of parallel computation upon multiprocessor architectures, *Proc. of Int'l Conf. on Parallel Processing*, vol 3, pp. 1-8, 1988.

[37] R. Krasny, Computation of vortex sheet roll-up in the trefftz plane, *Journal of Fluid Mechanics*, Vol 184, pp. 123-155, 1987.

[38] C. McGreary and H. Gill, Automatic determination of grain size for efficient parallel processing, *Communications of the ACM*, Vol. 32, pp. 1073-1078, Sept. 1989,

[39] K. A. Murthy, Y. Li and P.M. Pardalos. A local search algorithm for quadratic assignment problem. *Informatica*, Vol 3, No. 4, pp 524-538, 1992.

[40] J.M. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems*, New York:Plenum, 1988.

[41] G.N.S. Prasanna, A. Agarwal and B. R. Musicus, Hierarchical compilation of macro dataflow graphs for multiprocessors with local memory. *IEEE Transactions on Parallel and Distributed Systems*, July 1994.

[42] K. Psarris, Linear time exact methods for data dependence analysis in practice,*Proc. 1995 International Conference on Parallel Processing*, Vol. II, pp.1-8, 1995.

[43] W. Pugh and D. Wonnacott, An exact method for analysis of value-based array data dependences, *Proc. of the Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, 1992.

[44] E. Rothberg and A. Gupta, An efficient block-oriented approach to sparse Cholesky factorization on the iPSC/860 and Paragon multicomputers. *Proceedings of the 1994 Scalable High Performance Computing Conference*, May 1994.

[45] Y. Saad, Gaussian elimination on hypercubes, in *Parallel Algorithms and Architectures*, Cosnard, M. et al. Eds., Elsevier Science Publishers, North-Holland, 1986.

[46] J. K. Salmon, Parallel Hierarchical N-body Methods. Ph.D thesis, California Institute of Technology, 1990.

[47] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*, The MIT Press, 1989.

[48] J. P. Singh, Parallel hierarchical N-body methods and their implications for multiprocessors, Ph.D thesis, Stanford University, 1993.

[49] J. P. Singh, C. Holt, J. L. Hennessy and A. Gupta, A parallel adaptive fast multipole method. *IEEE Supercomputing 93*, pp. 54-65, 1993.

[50] V. S. Sunderam, G. A. Geist, J. Dongarra and R. Manchek, The PVM concurrent computing system: evolution, experiences, and trends. *Parallel Computing*, Vol 20, No. 4, pp. 531-547, April 1994.

[51] S. Venugopal and V. K. Naik, SHAPE: A parallelization tool for sparse matrix computations. Technical Report DSC-TR-290, Department of Computer Science, Rutgers University, New Brunswick, NJ. June 1992.

[52] D. W. Walker, The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, Vol 20, No. 4, pp.657-675, April 1994.

[53] M. Willebeek-LeMair and A. Reeves, Strategies for dynamic load balancing on highly parallel computers, *IEEE Trans. on Parallel and Distributed Systems,* Vol. 4, No. 9, pp. 979-993, Sept. 1993.

[54] M. E. Wolf and M.S. Lam, A loop transformation theory and an algorithm to maximize parallelism, *IEEE Trans. on Parallel and Distributed Systems,* Oct. 1991, pp. 452-471.

[55] M. Wolfe, The Tiny loop restructuring research tool, In *Proc. International Conference on Parallel Processing*, 1991.

[56] M. Wolfe, Optimizing supercompilers for supercomputers, Pitman, London and The MIT press, 1989.

[57] M. Wolfe and U. Banerjee, Data dependence and its application to parallel processing, *International Journal of Parallel Programming,* Vol. 16, No. 2, pp. 137-178, 1987.

[58] M. Y. Wu and D. Gajski, Hypertool: A programming aid for message-passing systems, *IEEE Trans. on Parallel and Distributed Systems,* vol. 1, no. 3, pp.330-343, 1990.

[59] T. Yang, Scheduling and code generation for parallel architecture, Ph.D Thesis, DCS-TR 299, Rutgers University, May 1993.

[60] T. Yang and A. Gerasoulis, PYRROS: Static task scheduling and code generation for message-passing multiprocessors, *Proc. of 6th ACM Inter. Conf. on Supercomputing,* Washington D.C., July, 1992, pp. 428-437.

[61] T. Yang and A. Gerasoulis, A Fast Scheduling Algorithm for DAGs on an Unbounded Number of Processors, Proc. of IEEE Supercomputing 91, pp. 633-642.

[62] T. Yang and A. Gerasoulis, List scheduling with and without communication delay, *Parallel Computing*, Vol 19, 1993, pp. 1321-1344.

[63] T. Yang and A. Gerasoulis, DSC: Scheduling parallel tasks on an unbounded number of processors, *IEEE Transactions on Parallel and Distributed Systems,* Vol. 5, No. 9, 951-967, 1994.

[64] T. Yang, C. Fu, A. Gerasoulis and V. Sarkar, Mapping iterative task graphs on distributed-memory machines. *Proc. of 24th Inter. Conference on Parallel Processing,* Vol. II, pp. 151-158, Aug. 1995.

# Vita

## Jia Jiao

| | |
|---|---|
| **1985** | Graduated from Xian High School, Xian, China. |
| **1985-90** | Attended Tsinghua University, Beijing, China. |
| **1990** | B.S in Computer Science, Tsinghua University. |
| **1991-96** | Graduate work in Computer Science, Rutgers, The State University of New Jersey, New Brunswick, New Jersey. |
| **1991-94** | Teaching Assistant, Department of Computer Science. |
| **1993** | M.S. in Computer Science, Rutgers, The State University of New Jersey. |
| **1994-96** | Research Assistant, Department of Computer Science. |
| **1996** | Ph.D. in Computer Science, Rutgers, The State University of New Jersey. |

.