0

Efficient Algorithms and a Software Tool for Scheduling Parallel Computation

Apostolos Gerasoulis, Rutgers University
Tao Yang, University of California at Santa Barbara

0.1 Introduction

In this chapter, we consider the partitioning and scheduling problem for directed acyclic program task graphs (DAG). We emphasize algorithms for scheduling parallel architectures based on the asynchronous message passing paradigm for communication. Such architectures are becoming increasingly popular but programming them is very difficult since both the data and the program must be partitioned and distributed to the processors. The following problems are of major importance for distributed memory architectures: (1) The program and data partitioning and the identification of parallelism. (2) The mapping of the data and program onto an architecture. (3) The scheduling and co-ordination of the task execution.

From a theoretical point of view all problems above are extremely difficult in the sense that finding the optimum scheduling solution is NP-complete in general [Ch89a, CD73, LR78, PY90]. In practice, however, parallel programs are written routinely for distributed memory architectures with excellent performance. Thus one of the grand challenges in parallel processing is if a compiler can be built that will *automatically* partition and parallelize a sequential program and then produce a schedule and generate the target code for a given architecture. For a specialized class of sequential program definitions, the identification of parallelism becomes simpler. However, choosing good partitions even in this simple case is difficult and requires the computation of a schedule.

Scheduling Theory and Its Applications. P. Chretienne, E. G. Coffman, J. K. Lenstra, Z. Liu (Eds.)
©1994 John Wiley & Sons Ltd

We discuss the techniques of data and program partitioning, and present an overview of the scheduling problem. We emphasize static scheduling over dynamic, because it is still an open problem how to reduce the run-time overhead of dynamic scheduling for distributed memory architectures. We have addressed the issues of static scheduling and developed algorithms along with a software system named PYRROS [YG92]. PYRROS takes as an input a task graph and produces schedules for message passing architectures such as nCUBE-II. The current PYRROS prototype has a low complexity and can handle task graphs with millions of tasks.

An automatic system for scheduling and code generation is useful in many ways. If the scheduling is determined at compile time then the architecture can be utilized better. Also a programmer does not have to get involved in low level programming and synchronization. The system can be used to determine a good program partitioning before actual execution. It can also be used as a testbed for comparing manually written scheduling with an automatically generated scheduling.

0.2 Program Partitioning and Data Dependence

We start with definitions of the task computation model and architecture:

- **A directed acyclic weighted task graph** (DAG) is defined by a tuple $G = (V, E, \mathcal{C}, \mathcal{T})$ where $V = \{n_j, j = 1 : v\}$ is the set of task nodes and $v = |V|$ is the number of nodes, E is the set of communication edges and $e = |E|$ is the number of edges, $\mathcal{C}$ is the set of edge communication costs and $\mathcal{T}$ is the set of node computation costs. The value $c_{i,j} \in \mathcal{C}$ is the communication cost incurred along the edge $e_{i,j} = (n_i, n_j) \in E$, which is zero if both nodes are mapped in the same processor. The value $\tau_i \in \mathcal{T}$ is the execution time of node $n_i \in V$.
- **A task** is an indivisible unit of computation which may be an assignment statement, a subroutine or even an entire program. We assume that tasks are convex, which means that once a task starts its execution it can run to completion without interrupting for communications, Sarkar [Sar89].
- **The static macro-dataflow** model of execution, Sarkar [Sar89], Wu and Gajski [WG88], El-Rewini and Lewis [EL90]. This is similar to the dataflow model. The data flow through the graph and a task waits to receive all data in parallel before it starts its execution. As soon as the task completes its execution it sends the output data to all successors in parallel.

0.2.1 Program partitioning

A program partitioning is a mapping of program statements onto a set of tasks. Since tasks operate on data, their input data must be *gathered* from a data structure and *transmitted* to the task before execution, then operated by the task and finally transmitted and *scattered* back to the data structure. If the data structure is distributed amongst many processors, then the *gather/scatter* and *transmission* operations are costly in terms of communication cost unless the data are *partitioned* properly.

We present an example for partitioning the Gaussian Elimination (GE) algorithm without pivoting. Fig. 0.1(a) shows a fine fine grain partitioning where tasks are defined

at the statement level

$$u_{ij,k} : \{ \quad a(i,j) = a(i,j) - a(i,k) * a(k,j)/a(k,k) \ \}.$$

This partitioning fully exposes the parallelism of the GE program but a fine grain machine architecture is required to exploit this parallelism. For coarse grain architectures, we need to use coarse grain program partitionings. Fig. 0.1(b) shows a coarse grain partitioning where the interior loop is taken as one task U_k^i. Each task U_k^i modifies row i using row k.

(a) kij fine grain partitioning

```
for k = 1 : n − 1
  for i = k + 1 : n
    for j = k + 1 : n
      uij,k
    end
  end
end
```

(b) kij coarse grain partitioning

```
for k = 1 : n − 1
  for i = k + 1 : n
    U_k^i : { for j = k + 1 : n
                uij,k
              end}
  end
end
```

Figure 0.1 The kij - fine and coarse grain partitionings for GE.

0.2.2 Data dependence graph

Once a program is partitioned, data dependence analysis must be performed to determine the parallelism in the task graph. For $n = 4$ the fine and coarse grain dependence graphs corresponding to Fig. 0.1 are depicted in Fig. 0.2. The statement-level fine grain graph has the dependence edges between node $u_{ij,k}$ for $k = 1 : 3$ and $i, j = k + 1 : 4$. Notice that task $u_{33,2}$ must begin execution after $u_{22,1}$ is completed since it uses the output of $u_{22,1}$. The direction of the dependence arrow shown in the DAG is determined by using the sequential execution of the kij program in Fig. 0.1. However, there is no dependence between $u_{22,1}$ and $u_{23,1}$ and they may be executed in parallel. All transitive edges have been removed from the graph.

The coarse grain graph is shown in Fig. 0.2 in ovals by aggregating several $u_{ij,k}$ into a coarser grain task U_k^i. We combine the edges between two oval tasks and a clear picture of the dependence task graph is shown as the U-DAG in Fig. 0.4.

0.2.3 Algorithms for partitioning

Partitioning algorithms require a cost function to determine if a partitioning is good or not. One widely used cost function is the minimization of the parallel time. Unfortunately, for this cost function the partitioning problem is NP-complete in most cases, [Sar89]. However, instead of searching for the optimum partitioning, we can search for a partitioning that has sufficient parallelism for the given architecture and also satisfies additional constraints. The additional constraints must be chosen so that the search space is reduced. An example of such a constraint is to search for tasks of a given maximum size that have no cycles. This is known as the *convexity constraint* in the literature, [Sar89]. A *convex* task is nonpreemptive in the sense that it receives

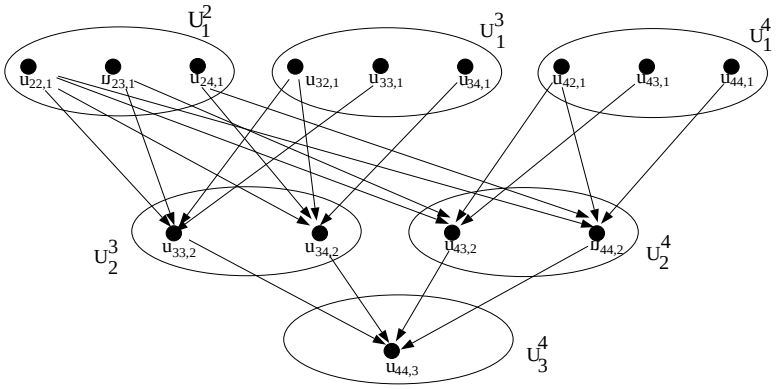


Figure 0.2 The fine grain DAG for GE and $n = 4$. Ovals show a coarse grain partitioning by aggregating small computation units $u_{i,j,k}$.

all necessary data items before starting execution and completes its execution without any interruption. After that, it sends the data items to the successor tasks that need those data.

Top-down: One methodology for program partitioning is to start from the top level (the program) and go down the loop nesting levels until sufficient parallelism is discovered. At each loop level a partitioning is defined by mapping everything below that level in a task. Then a data dependence analysis is performed to find the parallelism at that level. If no sufficient parallelism is found at that level then program transformations such as loop interchange can be performed and test again the new loop for parallelism. Incorporating this loop interchange program transformation technique can also change the data access pattern of each task.

We show how the top-down approach works for the GE example. There are three nesting loop levels in the program of Fig. 0.1. Starting from the top (outer loop) we see that there is no parallelism. At the next level there is parallelism for some of the loops but the task convexity constraint sequentializes the task graph so we must go to the next level which is the interior loop level for our program. At the interior loop the tasks are convex and there is sufficient parallelism for coarse grain architectures as shown in the U-DAG in Fig. 0.4.

By loop interchanging loops j and i in the kij GE program and taking the interior loop as a task, the result is the kji form of GE algorithm shown in Fig. 0.3. The dependence graph is the T-DAG in Fig. 0.4. Each task T_k^j uses column k to modify column j.

Bottom-up: One difficulty with the top-down approach is that this approach follows the program structure level to partition and it is difficult to identify an appropriate level other than the statement level that has sufficient parallelism. Thus this approach will usually end up with a fine grain statement level task partitioning. If that is the case and we are interested in coarse grain partitioning then we must go bottom-up to determine such partitioning. Finding an optimal partitioning is NP-complete and heuristics must be used.

We show an example of the bottom-up approach for Fig. 0.2. Given the fine grain DAG the partitioning in the ovals is a mapping corresponding to U-DAG coarse grain

```
for k = 1 : n − 1
  for j = k + 1 : n
    T_k^j : { for i = k + 1 : n
                u_ij,k
              end}
  end
end
```

Figure 0.3 The kji - coarse grain partitioning for GE.

DAG. Another coarse grain partitioning is to aggregate $u_{22,1}, u_{32,1}$, $u_{42,1}$ into T_1^2 and so on; this results in the T-DAG shown in Fig. 0.4. The T-DAG and U-DAG have the same dependence structure but different task definitions. The two partitionings are also known as row and column partitionings because of their particular data access patterns.

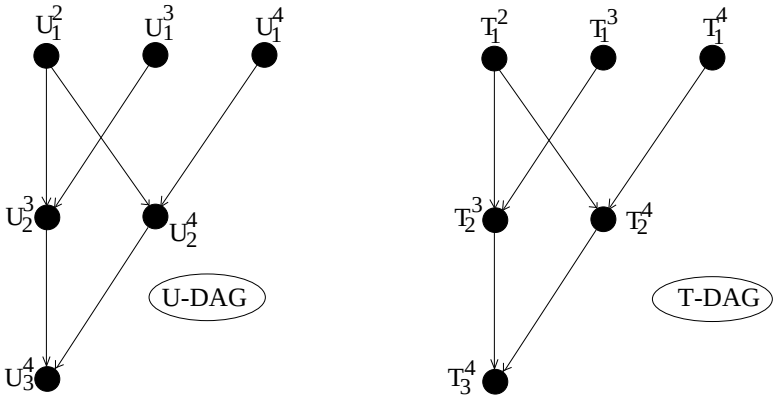


Figure 0.4 Dependence task graphs corresponding to two coarse grain partitioning. U-DAG with row data access pattern and T-DAG with column data access pattern

0.2.4 Data partitioning

For shared memory architectures, the data structure is kept in a common shared memory while for distributed memory architectures, the data structure must be partitioned into *data units* and assigned to the local memories of the processors. A data unit can be a scalar variable, a vector or a submatrix block. For distributed memory architectures large grain data partitioning is preferred because there is a high communication startup overhead in transferring a small size data unit. If a task requires to access a large number of distinct data units and data units are evenly distributed among processors, then there will be substantial communication overhead in fetching a large number of non-local data items for executing such task. Thus the following property can be used to determine program and data partitionings:

Consistency. The program partitioning and data partitioning are consistent if sufficient parallelism is provided by partitioning and at the same time the number of distinct units accessed by each task is minimized.

Let us assume that the fine grain task graph in Fig. 0.2 is given and also that the data unit is a row of the matrix. Then the program partitioning shown in ovals is consistent with such a data partitioning and it corresponds to the U-DAG in Fig. 0.4. The resulting coarse grain tasks U_k^j access an extensive number of data elements of rows k and j in each update. Making the data access pattern of a task consistent with data partitioning results in efficient reuse of data that reside in the local cache or the local memory.

Let us now assume that the matrix is partitioned in column data units. Then each task U_k^j needs to access n columns for each update, which results in excessive data movement. On the other hand, T-DAG task partitioning in Fig. 0.4 is consistent with column data partitioning since each task T_k^j only accesses 2 columns (k and j) for each update.

0.2.5 Computing the weights for the DAG.

Sarkar [Sar89] on page 139 has proposed a methodology for the estimation of the communication and computation cost for the macro dataflow task model. The computation cost is the time E for a task to execute on a processor. The communication cost consists of two components:

1. *Processor component:* The time that a processor participates in communication. The cost is expressed by the reading and writing functions R and W.
2. *Transmission delay component:* The time D for the transmission of the data between processors. During that time the processors are free to execute other instructions.

The weights can be obtained from

$$\tau_i \approx E_i, \qquad c_{ij} \approx R_i + D_{i,j} + W_j .$$

The parameters R_i, $D_{i,j}$, W_j are functions of the *message size*, the *network load* and the *distance* between the processors. When there is no network contention, a very common approximation to $c_{i,j}$ is the *linear model*:

$$c_{ij} \approx (\alpha + k\beta)d(i,j)$$

where α is known as the startup time, β is the transmission rate and k is the size of the message transmitted between tasks n_i and n_j and $d(i,j)$ is the processor distance between tasks n_i and n_j. This linear communication model is a good approximation to most currently available message passing architectures, see Dunigan [Dun91]. For the nCUBE-II hypercube we have $\alpha = 160\mu s$ and $\beta = 2.4\mu s$ per word transfer for single precision arithmetic.

For the GE example, if ω is the time that it takes for each $u_{ij,k}$ operation, then $\tau_{kj} = (n-k)\omega$ for task T_k^j in the T-DAG (or U_k^i in the U-DAG) of Fig. 0.4. The communication weights are all equal to $(\alpha + (n-k)\beta)d(T_k^j, T_{k+1}^j)$, since only $(n-k)$ elements of the data unit are modified in T_{k+1}^j.

Of course, for some task graphs the computation and communication weights or even the dependence structure can only be determined at run time. For such cases run-time scheduling techniques are useful, e.g. [Sal90].

0.3 Granularity and the Impact of Partitioning on Scheduling

0.3.1 Scheduling and clustering definitions

Scheduling is defined by a processor assignment mapping, $PA(n_j)$, of the tasks onto the p processors and by a starting times mapping, $ST(n_j)$, of all nodes onto the real positive numbers set. Fig. 0.5(a) shows a weighted DAG with all computation weights assumed to be equal to 1. Fig. 0.5(b) shows a processor assignment using 2 processors. Fig. 0.5(c) shows a *Gantt chart* of a schedule for this DAG. The Gantt chart completely describes the schedule since it defines both $PA(n_j)$ and $ST(n_j)$. The scheduling problem with communication delay has been shown to be NP-complete for a general task graph in most cases, Sarkar [Sar89], Chretienne [Ch89a] and Papadimitriou and Yannakakis [PY90].

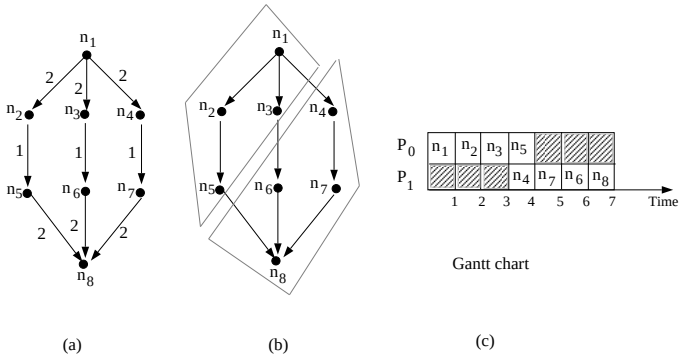


Figure 0.5 (a) A DAG with node weights equal to 1. (b) A processor assignment of nodes. (c) The Gantt chart of a schedule.

Clustering is a mapping of the *tasks* onto *clusters*. A *cluster* is a set of tasks which will execute on the same processor. Clusters are not tasks, since tasks that belong to a cluster are permitted to communicate with the tasks of other clusters immediately after completion of their execution. The clustering problem is identical to processor assignment part of scheduling. Sarkar [Sar89] calls it an *internalization prepass*. Clustering is also NP-complete for the minimization of the parallel time [Ch89a, Sar89].

A clustering is called *nonlinear* if two independent tasks are mapped in the same cluster, otherwise is called *linear*. In Fig. 0.6(a) we give a weighted DAG, in Fig. 0.6(b) a linear clustering with three clusters $\{n_1,\ n_2,\ n_7\}$, $\{n_3,\ n_4,\ n_6\}$, $\{n_5\}$ and in Fig. 0.6(c) a nonlinear clustering with clusters $\{n_1,\ n_2\}$, $\{n_3,\ n_4,\ n_5,\ n_6\}$ and $\{n_7\}$. Notice that for the nonlinear cluster independent tasks n_4 and n_5 are mapped in the same cluster.

In Fig. 0.7(a) we present the Gantt chart of a schedule for the nonlinear clustering

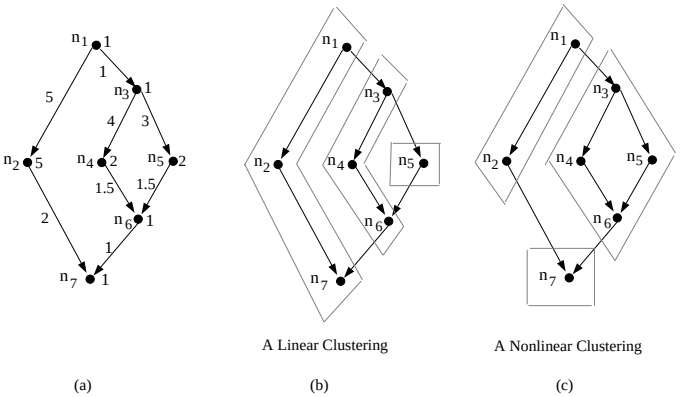


Figure 0.6 (a) A weighted DAG. (b) A linear clustering. (c) A nonlinear clustering.

of Fig. 0.6(c). Processor P_0 has tasks n_1 and n_2 with starting times $ST(n_1) = 0$ and $ST(n_2) = 1$. If we modify the clustered DAG as in [Sar89] by adding a zero-weighted pseudo edge between any pair of nodes n_x and n_y in a cluster, if n_y is executed immediately after n_x and there is no data dependence edge between n_x and n_y, then we obtain what we call a *scheduled DAG*. Fig. 0.7 (b) is a scheduled DAG and the dashed edge between n_4 and n_5 shows the pseudo execution edge.

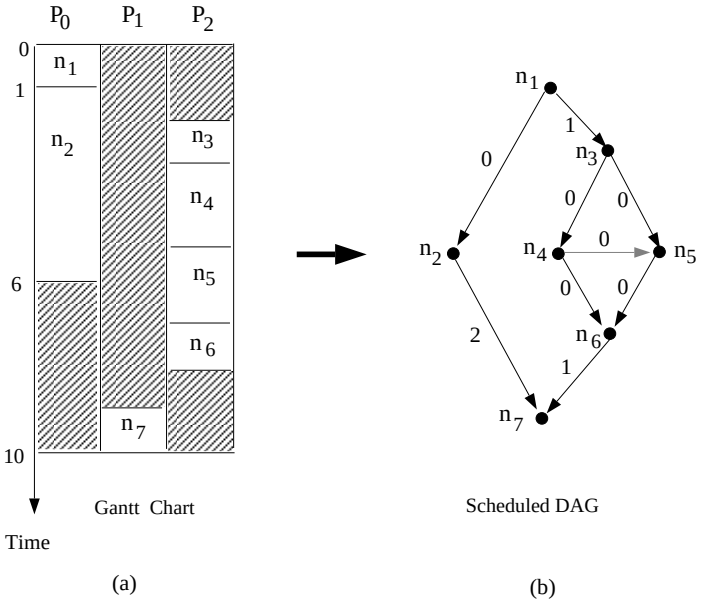


Figure 0.7 (a) The Gantt chart of a schedule for Fig. 1(c). (b) The scheduled DAG.

We call the longest path of the scheduled DAG the *dominant sequence* (DS) of the clustered DAG, to distinguish it from the *critical path* (CP) of a clustered but not scheduled DAG. For example, the clustered DAG in Fig. 0.6(c) has the sequence

$< n_1, n_2, n_7 >$ as its CP with length 9, while a DS of this clustered DAG is $DS =< n_1, n_3, n_4, n_5, n_6, n_7 >$ and has length 10 using the schedule of Fig. 0.7(b). In the case of linear clustering, the DS and CP of the clustered DAG are identical, see Fig. 0.6(b).

0.3.2 The Granularity theory

One goal of partitioning is to produce a DAG that has sufficient parallelism for a given architecture. Another is to have a partition that minimizes the parallel time. These two goals are in conflict because having a partitioning with a high degree of parallelism does not necessarily imply the minimization of the parallel time, unless communication cost is zero. It is therefore the communication and computation costs derived by a partitioning that will determine the "useful parallelism" which minimizes the parallel time. This has been recognized in the literature as it can be seen by the following quote from Heath and Romine [HR88] p. 559:

> " Another important characteristic determining the overall efficiency of parallel algorithms is the relative cost of communication and computation. Thus, for example, if communication is relatively slow, then coarse grain algorithms in which relatively large amount of computation is done between communications will be more efficient than fine-grain algorithms."

Let us consider the task graph in Fig. 0.8. If the computation cost w is greater or equal to communication cost c then the parallel time is minimum when n_2 and n_3 are executed in two separate processors as shown in Fig. 0.8(c). In this case all parallelism in this partitioned graph can be fully exploited since it is "useful parallelism". If on the other hand we assume that $w < c$, then the parallelism is not "useful" since the minimum parallel time is derived by sequentializing the tasks n_2 and n_3 as shown in Fig. 0.8(b).

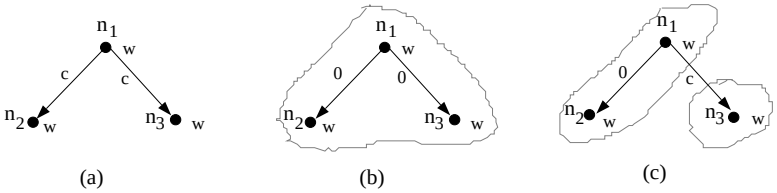


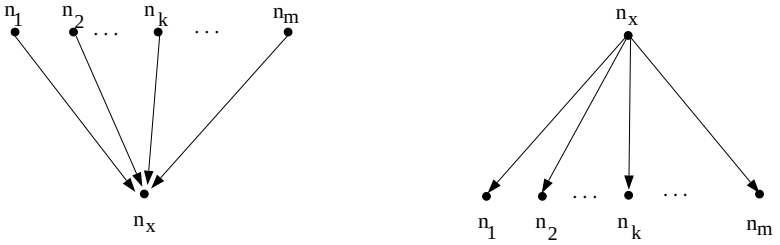
Figure 0.8 Sequentialization vs. parallelization. (a) A weighted DAG. (b) Sequentialization using nonlinear clustering. (c) Parallelization using a linear clustering.

Notice that linear clustering preserves the parallelism embedded in the DAG while nonlinear clustering does not. We make the following observation:

> *If the execution of a DAG uses linear clustering and attains the optimal time, then this indicates that the program partitioning is appropriate for the given architecture; otherwise the partitioning is too fine and the scheduling algorithm still needs to execute independent tasks together in the same processor using the nonlinear clustering strategy.*

It is therefore of interest to know when we can fully exploit the parallelism in a given task graph. In this section, we assume that the architecture has an unbounded number of processors which are completely connected (clique).

In Fig. 0.8 we saw the impact of the ratio w/c on scheduling a simple DAG. An interesting question arises: can this analysis be generalized to arbitrary DAGs. In Gerasoulis and Yang [GY93] we have introduced a new notion of *granularity* using a ratio of the computation to communication costs taken over all fork and joins subgraphs of a task graph. The importance of this choice of granularity definition will become clear later on.



(a) Join set J_x (b) Fork set F_x

Figure 0.9 Fork and join sets.

A DAG consists of *fork* or/and *join* sets such as the ones shown in figure 0.9. The join set J_x consists of all immediate predecessors of node n_x. The fork set F_x consists of all immediate successors of node n_x. Let $J_x = \{n_1, n_2, \ldots, n_m\}$ and $F_x = \{n_1, n_2, \ldots, n_m\}$ and define

$$g(J_x) = \min_{k=1:m}\{\tau_k\}/\max_{k=1:m}\{c_{k,x}\} \qquad g(F_x) = \min_{k=1:m}\{\tau_k\}/\max_{k=1:m}\{c_{x,k}\}.$$

We define the *grain* of a task n_x and the *granularity* of a DAG as

$$g_x = \min\{g(F_x), g(J_x)\}, \quad g(G) = \min_{x=1:v}\{g_x\}.$$

We call a DAG *coarse grain* if $g(G) \geq 1$ otherwise *fine grain*. If all task weights are equal to R and all edge weights are equal to C then the granularity reduces to R/C which is the same as Stone's [Sto87]. For coarse grain DAGs each task receives or sends data with a small amount of communication cost compared to the computation cost.

For example, the granularity of the graph in Fig. 0.6(a) is $g = 1/5$ which is derived as follows: The node n_1 is a fork and its *grain* is $g_1 = 1/5$, the ratio of the minimum computation weights of its successors n_2 and n_3, and the maximum communication cost of the outgoing edges. The node n_2 is in both a fork and a join and the grain for the join is $1/5$ which is the computation weight of its only predecessor n_1 and the cost of the edge (n_1, n_2), while the grain for the fork is the weight of n_7 over the weight of (n_2, n_7) which is again $1/2$. Continuing we finally determine the granularity as the minimum grain over all nodes of the graph which in our case is $g = 1/5$.

In [GY93] we prove the following theorems:

Theorem 1 *For a coarse grain task graph, there exists a linear clustering that minimizes the parallel time.*

The above theorem is true only for our granularity definition and that is the reason for choosing it. We demonstrate the basic idea of the proof by using the example in Fig. 0.8. We show in [GY93] that for any nonlinear clustering we can extract a linear clustering whose parallel time is less than or equal to the nonlinear clustering. If we assume that $w \geq c$ in Fig. 0.8, then the parallel time of the nonlinear clustering in Fig. 0.8(b) is $3w$. By extracting n_3 from the nonlinear clustering and making it a new cluster, we derive a linear clustering shown in 0.8(c) whose parallel time is $2w+c \leq 3w$. We can always perform this extraction as long as the task graph is coarse grain.

Theorem 1 shows that the problem of finding an optimal solution for a coarse grain DAG is equivalent to that of finding an optimal linear clustering. Picouleau [Pic92] has shown that the scheduling problem for coarse grain DAGs is NP-complete, therefore optimal linear clustering is NP-complete.

Theorem 2 *Determining the optimum linear clustering is NP-complete.*

Thus even though linear clustering is a nice property for task graphs, determining the optimum linear clustering is still a very difficult problem. Fortunately, for coarse grain DAGs, any linear clustering algorithm guarantees performance within a factor of two of the optimum as the following theorem demonstrates.

Theorem 3 *For any linear clustering algorithm we have*

$$PT_{opt} \ \leq \ PT_{lc} \ \leq \ (1 + \frac{1}{g(G)})PT_{opt}$$

where PT_{opt} is the optimum parallel time and PT_{lc} is the parallel time of the linear clustering. Moreover for a coarse grain DAG we have

$$PT_{lc} \leq 2 \times PT_{opt}.$$

Notice that when communication tends to zero then $g(G) \rightarrow +\infty$ and $PT_{opt} = PT_{lc}$. The above theorems provide an explanation of the advantages of linear clustering which has been widely used in the literature particularly for coarse grain dataflow graphs, e.g. [GH86, KB88, Kun88, Ort88, Saa86]. We present an example.

Example. A widely used assumption for clustering is "the owner computes rule" [CK88], i.e. a processor executes a computation unit if this unit modifies the data that the processor owns. This rule can perform well for certain *regular* problems but in general it could result in workload imbalances especially for unstructured problems. The "owner compute rule" has been used to cluster both the U-DAG and the T-DAG in Fig. 0.4, see Saad [Saa86], Geist and Heath [GH86] and Ortega [Ort88]. This assumption results in the following clusters for the U-DAG shown in Fig. 0.10:

$$M_j = \{U_1^j, U_2^j, \ldots, U_k^j, \ldots, U_{j-1}^j\}, \quad j = 2 : n.$$

For each cluster M_j row j remains local in that cluster while it is modified by rows $1 : j-1$ (similarly for columns in the T-DAG). The tasks in M_j are chains in the task graph in Fig. 0.10 which imply that linear clustering was the result of the "owner computes rule". We call this special clustering, the *natural linear clustering.*

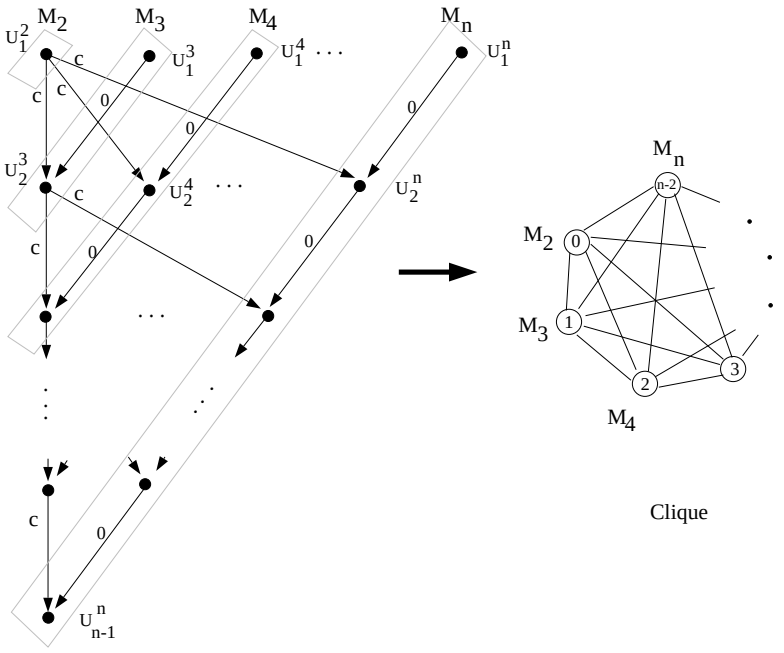


Figure 0.10 The natural linear clustering for the U-DAG executed on a clique with $p = n - 1$ processors

What is so interesting about the natural linear clustering? Let us assume that the computation size of all tasks is equal to τ, and communication weights are equal to c in the U-DAG. Then the following theorem holds:

Theorem 4 *The natural linear clustering is optimal for executing the U-DAG on a clique architecture with* $(n-1)$ *processors provided the granularity* $g = \frac{\tau}{c} \geq 1$.

An application of Theorem 4 is the kji column partitioning form of Gauss-Jordan (GJ) algorithm. At each step of the GJ algorithm all n elements of a column are modified and then transmitted to the successor tasks. The weights are then given by

$$\tau = n\omega, \qquad c = \alpha + n\beta$$

and as long as $n\omega/(\alpha + n\beta) \geq 1$, the GJ natural linear clustering is optimum. For the GE DAG, the weight of a task U_k^i in U-DAG or T_k^j in T-DAG is $(n-k)\omega$ and its incoming edge weights are $\alpha + (n-k)\beta$. For large n, only a small portion in the bottom of the DAG is fine grain and the natural clustering is asymptotically optimal by ignoring the insignificant low-order computation cost in this bottom portion.

We summarize our conclusions of this section as follows. For a program with coarse grain partitioning, linear clustering is sufficient to produce a good result. For a program with fine grain partitioning, linear clustering that preserves the parallelism of a DAG could lead to high communication overhead.

The granularity theory is a characterization of the relationship between partitioning and scheduling. In a real situation, some parts of a graph could be fine and others

coarse. In such cases clustering and scheduling algorithms are needed to identify such parts and use proper clustering strategies to obtain the shortest parallel time. We consider these problems next.

0.4 Scheduling Algorithms for MIMD architectures

We distinguish between two classes of scheduling algorithms. The one step methods schedule a DAG directly on the p processors. The multistep methods perform a clustering step first, under the assumption that there are unlimited number of completely connected processors, and then in the following steps the clusters are merged and scheduled on the p available processors. We consider heuristics that have the following properties: (1) They do not duplicate the same tasks in two different processors. (2) Backtracking is allowed only if the cost is small.

0.4.1 One step scheduling methods

We present two methods. One is the classical list scheduling and another is the Modified Critical Path (MCP) heuristic proposed by Wu and Gajski [WG88].

0.4.1.1 The classical list scheduling heuristic

The classical list scheduling schedules free[1] tasks by scanning a priority list from left to right. More specifically the following steps are performed:

1. Determine a priority list.
2. When a processor is *available* for execution, scan the list from left to right and schedule the first free task. If two processors are available at the same time, break the tie by scheduling the task in the processor with the smallest processor number.

When communication cost is zero, a good choice for a priority list is the Critical Path (CP) priority list. The priority of a task is its *bottom up level*, the length of the longest path from it to an exit node. The CP list scheduling possesses many nice properties when communication cost is zero. For example, it is optimum for tree DAGs with equal weights and for any arbitrary DAG with equal weights on 2 processors [CD73]. For arbitrary DAGs and p processors any list scheduling including CP is within 50% of the optimum. Moreover, the experimental results by Adam et. al. [ACD74] show that CP is near optimum in practice in the sense that it is within 5% of the optimum in 90% of randomly generated DAGs. Unfortunately, these nice properties do not carry over to the case of nonzero communication cost.

In the presence of communication, it is extremely difficult to identify a good priority list. This is because the communication edge weight becomes zero when its end nodes are scheduled in the same processor and this makes the computation of the level priority information non-deterministic.

[1] A task is *free* if all of its predecessors have completed execution. A task is *ready* if it is free and all of the data needed to start its execution are available locally in the processor where the task has been scheduled.

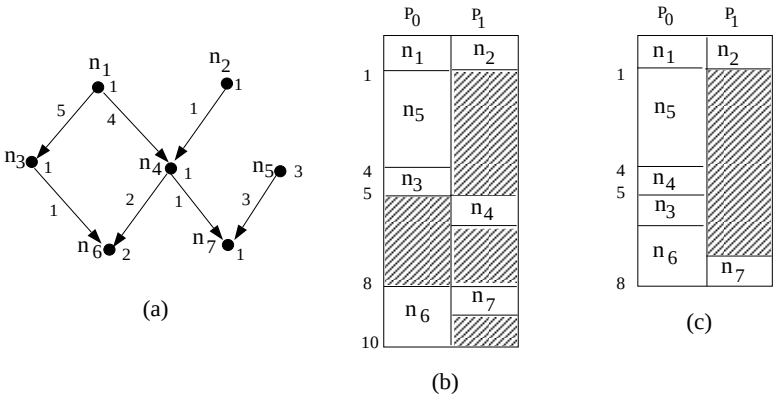


Figure 0.11 (a) A DAG. (b) The schedule by CP. (c) The schedule by MCP.

Let us consider the CP algorithm in the case where the level computation includes both edge communication and node computation. For example, a task graph is shown in Fig. 0.11(a) along with a list schedule based on the highest level first priority list. The level of n_6 is 2 and the level of n_3 is 4 which is equal to the maximum level of all successor tasks, which is 2, plus the communication cost in the edge (n_3, n_6), which is 1, plus the computation cost of n_3, which is 1. The resulting priority list is $\{n_1, n_2, n_5, n_4, n_3, n_6, n_7\}$. Both n_1 and n_2 are free and the processors P_0 and P_1 available. At time 0, n_1 is scheduled in P_0 first and in the next step n_2 is scheduled in the only available processor P_1. At time 1, the tasks n_3, n_4 and n_5 are free and since n_5 has the highest priority it is scheduled in processor P_0 while the next highest priority n_4 is scheduled in processor P_1. Even if n_4 was scheduled in P_1 it needs to wait 4 unit times to receive the data from P_0 and thus n_4 is ready to start its execution at time 5. The task n_5 scheduled in P_0 can start execution immediately since the data are local in that processor. Continuing in a similar manner we get the final schedule shown in Fig. 0.11(b) with $PT = 10$.

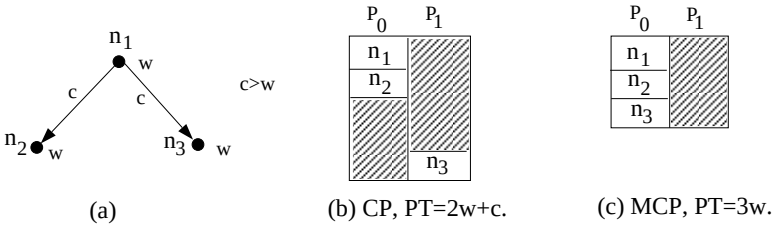


Figure 0.12 (a) A fork DAG. (b) The schedule by CP. (c) The schedule by MCP.

One problem with the CP heuristic in the presence of communication is that it schedules a *free* task when a processor becomes available, even though this task is not *ready* to start execution yet. This could result in poor performance as shown in Fig. 0.12(b). Task n_3 is scheduled in P_1 since it becomes free at time w. When $c > w$ a better solution is to schedule n_3 to P_0 shown in Fig. 0.12(c). We now present a modification to the CP heuristic.

0.4.1.2 The modified critical path (MCP) heuristic

Wu and Gajski [WG88] have proposed a modification to the CP heuristic. Instead of scheduling a free task in an available processor, the free task is scheduled in the available processor that allows the task to start its execution at the *earliest* possible time. The computation of the priorities uses again the highest bottom up level including both communication and computation costs. For the example in Fig. 0.12(a), the priority list is $\{n_1, n_2, n_3\}$. The schedule is shown in Fig. 0.12(c). The task n_3 becomes free at time w and it is scheduled in processor P_0 because it can start its execution at time $2w$ which is earlier than the time $w + c$ since $c > w$.

For the example in Fig. 0.11(a), the priority list is the same as in CP: $\{n_1, n_2, n_5, n_4, n_3, n_6, n_7\}$. After n_1, n_2 and n_5 are scheduled, task n_4 has the highest priority and is free at time 2 but is not ready at that time unless it is scheduled at P_0. Now n_4 is picked up for scheduling and it is scheduled in processor P_0 because it can start executing at time 4 which is earlier than time 5 if it was scheduled in P_1. The parallel time reduces to $PT = 8$ as depicted in Fig. 0.11(c).

Even though the MCP performs better than CP, it could still perform poorly as can be seen in the scheduling of a join DAG shown in Fig. 0.13. MCP gives the same schedule as CP and if the communication cost is greater than computation cost the optimum schedule executes all tasks in one processor. The MCP cannot recognize this since it uses the earliest starting time principle and it starts both n_2 and n_3 at time 0. One weakness of such one-pass scheduling is that the task priority information is non-deterministic because the communication cost between tasks will become zero if they are allocated in the same processor.

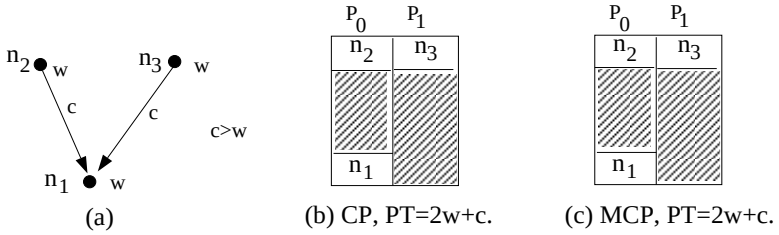


Figure 0.13 (a) A join DAG. (b) The schedule by CP. (c) The schedule by MCP.

It has been argued in the literature by Sarkar [Sar89] and Kim and Browne [KB88] that a better approach to scheduling when communication is present is to perform scheduling in more than one steps. We discuss this approach next.

0.4.2 Multistep scheduling methods

0.4.2.1 Sarkar's approach

Sarkar's heuristic [Sar89] is based on the assumption that a scheduling pre-pass is needed to cluster tasks with high communication between them. Then the clusters are scheduled on p available processors. To be more specific Sarkar advocates the following two step method:

1. Determine a clustering of the task graph by using scheduling on an unbounded number of processors and a clique architecture.
2. Schedule the clusters on the given architecture with a bounded number of processors.

Sarkar [Sar89] uses the following heuristics for the two steps above:

1. Zero the edge with the highest communication cost. If the parallel time does not increase then accept this zeroing. Continue with the next highest edge until all edges have been visited.
2. After u clusters are derived, schedule those clusters to p processors by using a priority list. Assuming that the v task nodes are sorted in a descending order of their priorities and the nodes are scanned from left to right. The scanned node, along with the cluster that it belongs to, is mapped on one of the p processors that results in *the minimum increase in parallel time*. The parallel time is determined by executing the examined clusters in the physical processors and the unexamined clusters in virtual processors.

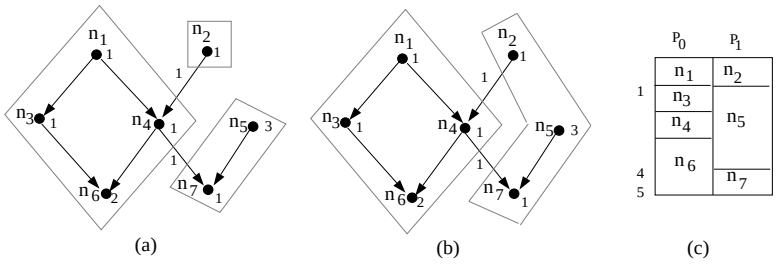


Figure 0.14 (a) The clustering result. (b) Clusters after merging. (c) The schedule with $PT = 5$.

Let us see how this two step method works for the example in Fig. 0.11(a). Initially the parallel time is 10. Sarkar's first clustering step zeroes the highest communication edge (n_1, n_3) and the parallel time does not increase and this zeroing is accepted. The next highest edge (n_1, n_4) is zeroed and the parallel time reduces, by executing n_3 either before or after n_4, so that this zeroing is also accepted. Assume that n_3 is executed before n_4. Next the edge (n_5, n_7) is zeroed and after that the edge (n_4, n_6) and the parallel time reduces to 5 which is determined by a $DS =< n_1, n_3, n_4, n_6 >$. By zeroing both (n_2, n_4) or (n_4, n_7) the parallel time increases and these zeroings are not accepted. The final result is three clusters: $M_1 = \{n_1, n_3, n_4, n_6\}, M_2 = \{n_2\}$, and $M_3 = \{n_5, n_7\}$ shown in Fig. 0.14(a).

Assume there are two processor P_0 and P_1 available. The second step in Sarkar's' algorithm determines a priority list based on the highest level first principle. The initial list is $\{n_2, n_1, n_5, n_3, n_4, n_6, n_7\}$ because the level of n_2 is 5 while the level of n_1 is 4 and so on. The algorithm first picks n_2 to schedule and let us assume that it is scheduled in processor P_1. Next the task n_1 is chosen to be scheduled. If it is scheduled to P_0 then all nodes in M_1 are scheduled to P_0 and PT is 5. If it is scheduled to P_1 then PT becomes 6 since now n_1 and n_2 must be sequentialized. Thus we assign M_1

to P_0. Next n_5 is scanned and it is scheduled to P_0 otherwise scheduling to P_1 will make $PT = 9$. Next n_3 is scanned, if it is assigned to P_1 then all other nodes in M_1 will be re-assigned to P_1 and $PT = 10$. Thus n_3 remains in P_0. Finally we have a schedule shown in Fig. 0.14(c).

0.4.2.2 PYRROS's multistep scheduling algorithms

The PYRROS tool [YG92] uses a multistep approach to scheduling:

1. Perform clustering using the Dominant Sequence Algorithm (DSC).
2. Merge the u clusters into p completely connected virtual processors if $u > p$.
3. Map the p virtual processors into p physical processors.
4. Order the execution of tasks in each processor.

This approach has similarities to Sarkar's two step method. There is however a major difference. The algorithms used here are faster in terms of complexity. This is because we would like to test the multistep method on real applications and parallel architectures and higher complexity algorithms offer very little performance gains especially for coarse grain parallelism.

The DSC clustering algorithm:

Sarkar's clustering algorithm has a complexity of $O(e(v+e))$. Furthermore, zeroing the highest communication edges is not the best approach since this edge might not belong in the DS and as a result the parallel time cannot be reduced. In [YG91, GY92] we have proposed a new clustering algorithm called the DSC algorithm which has been shown to outperform other algorithms from the literature, both in terms of complexity and parallel time. The DSC algorithm is based on the following heuristic:

- The parallel time is determined by the DS. Therefore if we want to reduce it we must zero at least one edge in the DS.
- A DS zeroing based algorithm could zero one or more edges in DS at a time. This zeroing can be done incrementally in a sequence of steps.
- A zeroing should be accepted if the parallel time reduces from one step to the next.

We show how DSC works for the example of Fig. 0.11(a). Fig. 0.15(a) is the initial clustering. The DS is shown in thick arrows. There are two dominant sequences in Fig. 0.15(a) with $PT = 10$. In the first step, the edge (n_1, n_3) in one DS is zeroed as shown in Fig. 0.15(b). The new DS is $< n_1, n_4, n_6 >$ and $PT = 10$. This zeroing is accepted since PT does not increase. In the second step (n_1, n_4) is zeroed and n_4 is added as the last task of cluster $\{n_1, n_3\}$, which results in two new DS $< n_1, n_3, n_4, n_6 >$ and $< n_5, n_7 >$ shown in Fig. 0.15(c) with $PT = 7$ and this zeroing is also accepted. In the third step (n_4, n_6) is zeroed as shown in Fig. 0.15(d) and this zeroing is accepted since $PT = 7$ determined by DS $< n_5, n_7 >$. Next (n_5, n_7) is zeroed and the PT is reduced to 5. Finally (n_2, n_4) and (n_4, n_7) cannot be zeroed because zeroing them will increase the parallel time. Thus three clusters are produced.

Notice that in the third step shown in Fig. 0.15(c) an ordering algorithm is needed to order the tasks in the nonlinear cluster and then the parallel time must be computed to get the new DS. One of the key ideas in the DSC algorithm is that it computes the

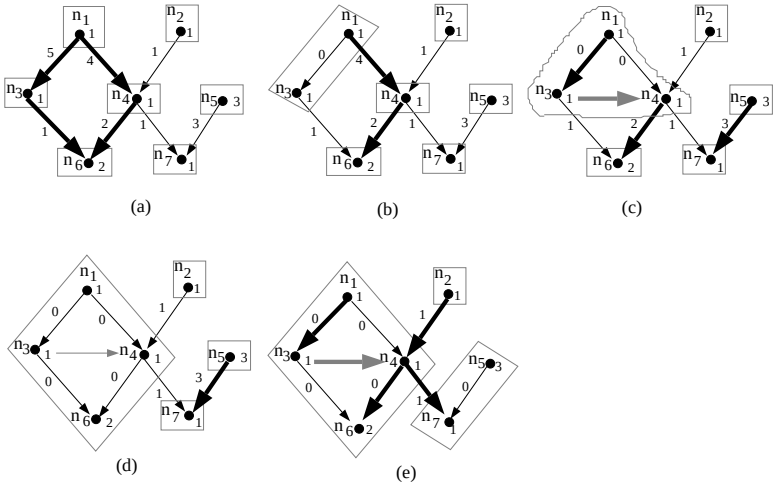


Figure 0.15 The clustering refinements in DSC.

schedule and parallel time incrementally from one step to the next in $O(\log v)$ time. Thus the total complexity is $O((v+e)\log v)$. If the parallel time was not computed incrementally, then the total cost would be greater than $O(v^2)$ which will not be practical for large graphs. More details can be found in [YG91].

The NP-completeness of clustering for parallel time minimization has been shown by Sarkar [Sar89], Chretienne [Ch89a] and Papadimitriou and Yannakakis [PY90]. Chretienne [Ch89b] shows that the problems of scheduling a join, fork DAG or coarse grain tree DAG are solvable in a polynomial time, but the complexity jumps to NP-complete for scheduling fine-grain tree DAGs and a DAG structure obtained by concatenating a fork and a join together. In [YG91], we show that DSC performs well for general DAGs by examining a set of randomly generated DAGs but also produces the following optimal solutions.

Theorem 5 *DSC is optimal for fork, join and coarse grain tree DAGs.*

Cluster merging:

The cost of the Sarkar cluster merging and scheduling algorithm is $O(pv(v+e))$ which is time-consuming for a large graph. PYRROS uses a variation of *work profiling method* suggested by George et. al. [Geo86] for cluster merging. This method is simple and has been shown to work well in practice, e.g. Saad [Saa86], Geist and Heath [GH86], Ortega [Ort88], Gerasoulis and Nelken [GN89]. The complexity of this algorithm is $O(u\log u+v)$, which is less than $O(v\log v)$.

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.

Let us consider an example. For the GE U-DAG in Fig. 0.10 there are $(n-1)$ clusters $M_2, M_3, \cdots, M_n$. We have that

$$LM_j = \sum_{i=1}^{j}(n-i)\omega \approx nj - \frac{j^2}{2}.$$

These clusters can be approxmiately load balanced by using the wrap or reflection mapping, $VP(j) = (j-2) \bmod p$, Geist and Heath [GH86].

For the example in Fig. 0.14(a) with 3 clusters and 2 processors, the result of merging is two clusters shown in Fig. 0.14(b).

Physical mapping:

We now have p virtual processors (or clusters) and p physical processors. Since the physical processors are not completely connected, we must take the processor distance into account. Determining the optimum mapping of the virtual to physical processors is a very difficult problem since it can be instantiated as a Graph isomorphism problem.

Let us define $TC_{i,j}$ to be the total communication, which is the summation of costs of all edges between virtual processor i and j. $CC = \{TC_{i,j} | TC_{i,j} \neq 0\}$ and $m = |CC|$. In general we expect that $m << e$.

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):

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

Fig. 0.16 is an example of physical mapping for a T-DAG. A clustering for this DAG is shown in Fig. 0.16(a). The total communication between 4 virtual processors (clusters) is shown in Fig. 0.16(b). In Fig. 0.16(c) we show one physical mapping to a 4-node hypercube with $F(CC) = 24$ and another mapping is shown in (d) with $F(CC) = 21$.

Currently we use a heuristic algorithm due to Bokhari [Bok90]. This algorithm starts from an initial assignment, then performs a series of pairwise interchanges so that the $F(CC)$ reduces monotonically as shown in the example above.

Task ordering:

Once the physical mapping has been decided then a task ordering is needed to define the scheduling. Since we no longer move tasks between processors the communication cost between tasks becomes deterministic. We show how important task ordering is via an example. The processor assignment along with communication and computation weights are shown in Fig. 0.17(a). In Fig. 0.17(b) we show one ordering with $PT = 12$ and in (c) another ordering in which the parallel time increases to $PT = 15$.

Finding a task ordering that minimizes the parallel time is NP-hard [HVV92]. We have proposed a modification to the CP heuristic for the ordering problem in Yang and Gerasoulis [YG95]. This heuristic, Ready Critical Path (RCP), costs $O(v \log v + e)$

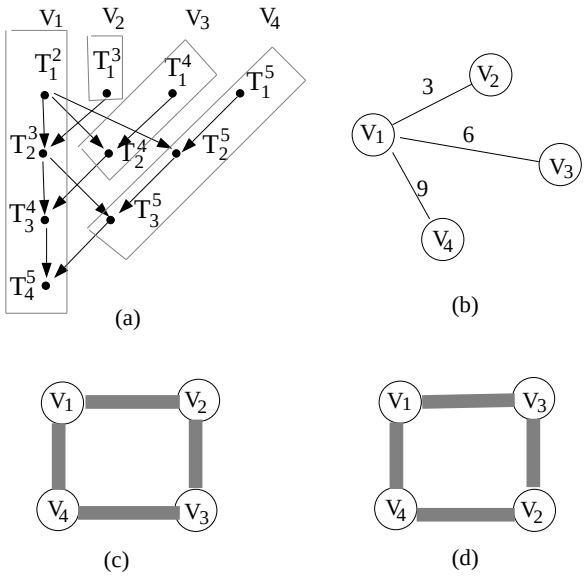


Figure 0.16 An example of physical mapping. Each nonzero edge cost is 3 time units. (a) A T-DAG linear clustering (b) Virtual cluster graph (c) A mapping to a hypercube (d) A better mapping.

and is described below:

1. Adjust the communication edges of the DAG based on the processor assignment and physical distance.
2. Determine a global priority list based on the highest level first principle. The level computation includes both communication and computation cost in a path.
3. In addition to the global priority list each processor maintains a priority list of *ready* tasks for each processor. The ready task with the highest priority is executed as soon as this processor becomes free.

Let us consider the processor assignment in Fig. 0.17(a). The level priorities of tasks are: $L(n_1) = 12$, $L(n_2) = 7$, $L(n_3) = 1$, $L(n_4) = 1$, $L(n_5) = 2$, $L(n_6) = 2$. The priority list is $\{n_1, n_2, n_5, n_6, n_3, n_4\}$. Initially, n_1 is ready and is scheduled on the first processor. At time 5, n_2 and n_5 are ready in the second processor and n_2 is scheduled because of higher priority. The case is similar in the third processor for scheduling n_3 and n_6. The resulting schedule is shown in Fig. 0.17(b) and its parallel time is $PT = 12$.

The task ordering problem is NP-hard even for chains of tasks [HVV92], however, in [YG95] we prove that fork and join DAGs are tractable.

Theorem 6 *RCP is optimal for fork and join DAGs.*

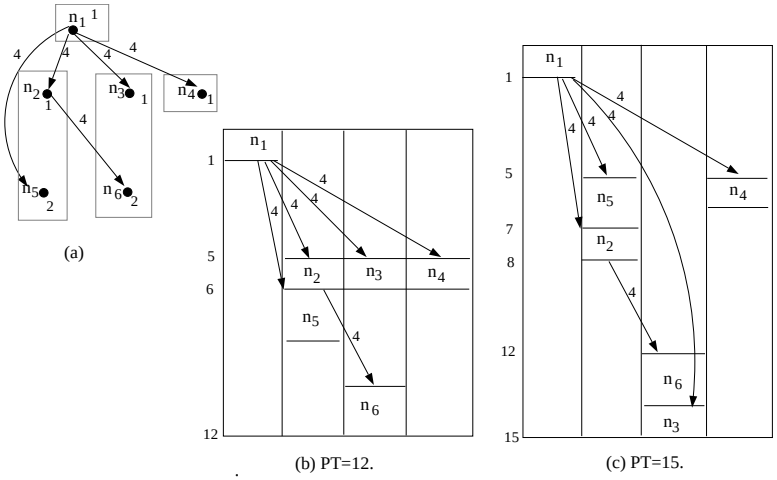


Figure 0.17 (a) A physical mapping of a DAG (b) The RCP ordering (c) Another ordering.

0.4.3 Load balancing vs. Sarkar's cluster merging algorithms

As we discussed above PYRROS uses a simple heuristic based on load balancing for merging clusters. This heuristic uses only the cluster load information and completely ignores task precedences and inter-cluster communication. It is of interest to see how such a simple heuristic will perform vs. a more sophisticated, but more expensive in terms of complexity, heuristic such as Sarkar's cluster merging algorithm. To make a fair comparison, we use the same clustering algorithm for both cases, the DSC algorithm. We next merge the clusters using: (1) the load balancing heuristic (2) Sarkar's merging algorithm. We assume a clique architecture to avoid any mapping effects and use the RCP ordering in both cases to order tasks.

We generate randomly 100 DAGs and weights as follows: The number of tasks and edges are randomly generated and then assign randomly computation and communication weights. The size of the graphs varies from a minimum average of 143 nodes and 264 edges to a maximum average of 354 nodes and 2620 edges. In our experiments, the number of processors is chosen based on the widths of the graphs. The *width* and *depth* of graphs vary from 8 to 20 and thus we choose $p = 2, 4, 8$. Also to see the performance for both fine and coarse grain graphs we vary the granularity by varying the ratio of average computation over communication weights from 0.1 to 10.

Fig. 0.18 shows that the average improvement ratio (1-T(Sarkar)/T(Load balancing), where T() is the parallel time) of Sarkar's algorithm over the load balancing heuristic is between 10% to 35%. When the width of the graph is small compared to the number of processors, e.g. $p = 8$, Sarkar's algorithm is better than load balancing by about 30%. On the other hand, when the width is much larger than the number of processors then the performance differences are getting smaller, especially for coarse grain graphs, e.g. for $p = 2$ the improvement ratio reduces to about 10 % for coarse grain graphs. Intuitively, this is expected since each processor is assigned a larger number of tasks when the width to processor ratio increases and the RCP ordering heuristic can better overlap the computation and communication.

With respect to the execution time of the heuristics, for a Sun Sparcstation computer

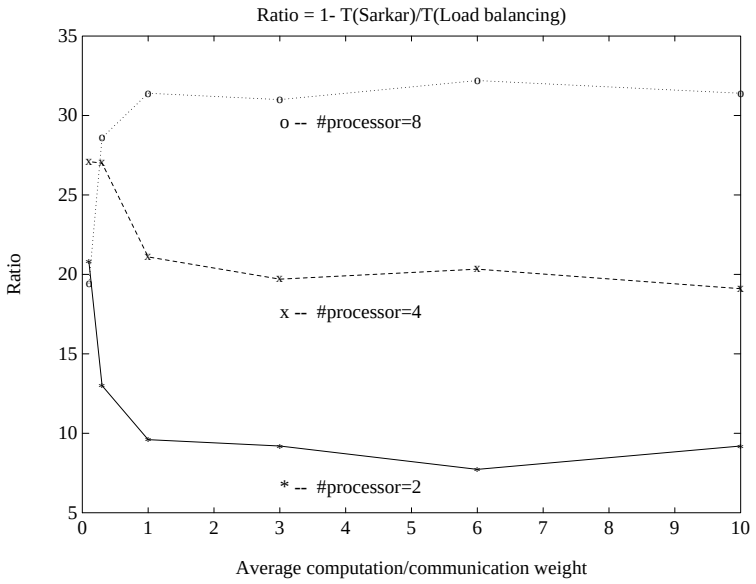


Figure 0.18 The performance of Sarkar's merging algorithm vs. load balancing algorithm. The graph width and depth are between 8 and 20.

the load balancing heuristic takes about 0.1 seconds to produce a solution for graphs with average $v = 200$ and $e = 400$, while Sarkar's algorithm takes about 40 seconds. When we double the graph size, the load balancing heuristic takes 0.2 seconds while Sarkar's needs 160 seconds. For the above graphs and p, the time spent for each graph varied from 0.05 to 0.3 seconds for the load balancing heuristic and from 9.8 seconds to 725 seconds for Sarkar's. On the average, the load balancing heuristic was 1000 times faster than Sarkar's for those cases.

To verify our conclusions we increased the width of graphs from 8-20 to 30-40 but then reduced the depth of graphs between 5-8 to keep the number of tasks sufficiently small for the complexity of Sarkar's algorithm. The results are shown in Fig. 0.19 and are consistent with our previous conclusions. The performance of Sarkar's algorithm becomes better as the number of processors increases from $p = 2$ to $p = 16$ but then the performance reverses for $p = 32$, as expected, since p approaches the width of the graph.

Our experiments show that on average the performance of the load balancing algorithm is within 75% of Sarkar's algorithm for those random graphs. This is very encouraging for the widely used load balancing heuristic. However, more experiments are needed to verify this result.

0.5 The PYRROS software tool

The input of PYRROS is a weighted task graph and the associated sequential C or Fortran code. The output is a static schedule and parallel code for a given architecture. The function modules of PYRROS are shown in Fig. 0.20. The current

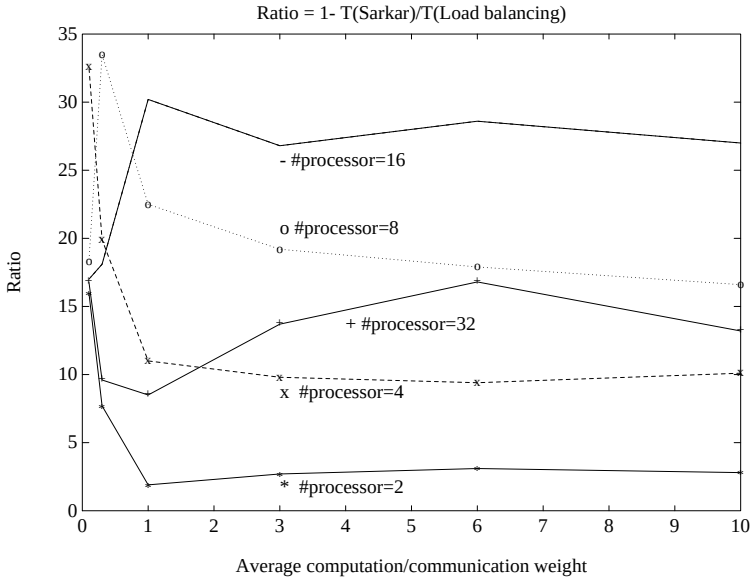


Figure 0.19 The performance of two merging algorithms for the graphs with width between 30 and 40 and depth between 5 and 8.

PYRROS tool has the following components: a task graph language with an interface to C and Fortran, allowing users to define partitioned programs and data; a scheduling system for clustering the graph, load balancing and physical mapping, and communication/computation ordering; a graphic displayer for displaying task graphs and scheduling results; a code generator that inserts synchronization primitives and performs code optimization for nCUBE-I, nCUBE-II and INTEL iPSC/860 hypercube machines. A more detailed description of PYRROS is given in [YG92].

There are several other systems related to PYRROS. PARAFRASE-2 [Pol90] by Polychronopoulos et. al., is a parallelizing compiler system that performs dependence analysis, partitioning and dynamic scheduling on shared memory machines. SCHEDULER by Dongarra and Sorensen [DS87] uses centralized dynamic scheduling for a shared memory machine. KALI by Koelbel and Mehrota [KM90] addresses code generation and is currently targeted at DOALL parallelism. Kennedy's group [HKT91] is also working on code generation for FORTRAN D for distributed-memory machines. PARTI by Saltz's group [Sal90] focuses on run-time DOALL parallelism with irregular distribution of data and optimizes performance by precomputing data accessing patterns. HYPERTOOL by Wu and Gajski [WG88] and TASKGRAPHER by El-Rewini and Lewis [EL90] use the same task model as PYRROS. The time complexity of these two systems is over $O(v^2)$.

0.5.1 Task graph language

The PYRROS system uses a simple language for defining task graphs. For example, the program code in Fig. 0.21 is a description of the T-DAG partitioning shown in Fig. 0.3 and 0.4 in terms of PYRROS task graph language. The key words are boldfaced. The

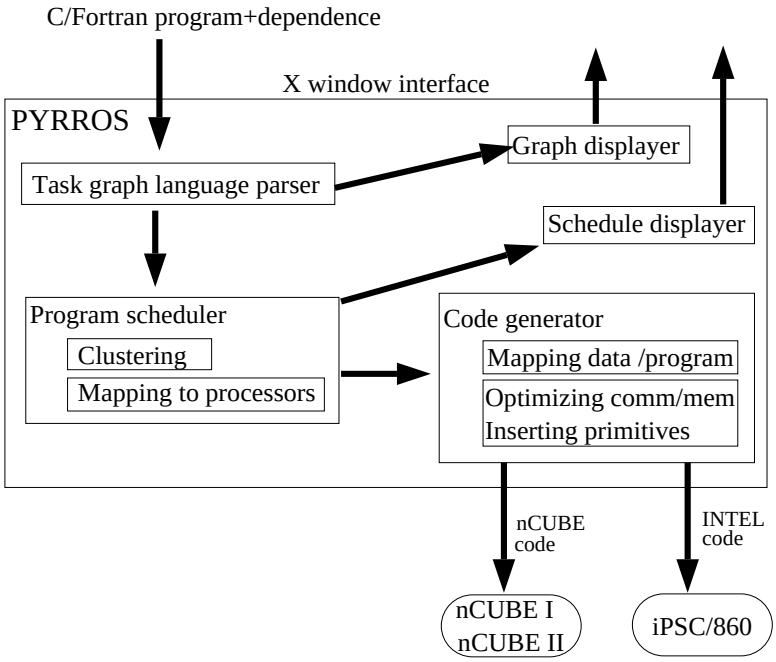


Figure 0.20 The system organization of PYRROS prototype.

semantic of the loop is the same as that in Fig. 0.3. The interior loop body contains the data dependence and weight information for a task T_k^j along with the specification of task computation. Task T_k^j reads column k and j if $k > 1$. The *c_update* is an external C function which defines the updating of column j using column k for task T_k^j corresponding to the interior loop in the GE program in Fig. 0.3. After the *c_update* is executed, then if $k < n$ this task writes column j to be used by T_{k+1}^j and also performs a broadcast if $k = j - 1$.

PYRROS will read this program and perform lexical and semantic analysis to generate an internal representation of the DAG. Then using the X-window DAG displayer we can verify whether the definition of the task graph is correct.

0.5.2 A demonstration of PYRROS usage

In this section we demonstrate one usage of PYRROS. For GE T-DAG, we choose $\alpha = 10, \beta = \omega = 1, n = 5$ and PYRROS displays the dependence graph in the screen as shown in the left part of Fig. 0.22. Task $T(1,2)$ corresponds to task T_1^2 in the T-DAG of Fig. 0.4 and has an internal task number 1 written to its right. The edges of the DAG show the columns sent from one task to the successors.

As we mentioned above when a program is manually written for a library such as LINPACK, the clustering must be given in advance. Let us assume that the widely used natural linear clustering M_j defined previously is used. This implies that $M_1 = \{T(1,2)\} = \{T1\}$, $M_2 = \{T(1,3), T(2,3)\} = \{T2, T5\}$ and so on. At this point the user, executing the program with natural linear clustering, cannot determine how many processors to choose so that the parallel time is minimized. If he chooses $p = 4$,

```
struct Dataitem column[n];
task T[k][j]{
   int b;
   set_weight(n-k);
   if(k>1){
      read(&column[k], (n-k+1)*ELESIZE);
      read(&column[j], (n-k+1)*ELESIZE);
   }
   c_update(&column[k], &column[j],k,j);
   if(k<n-1){
      if(k!=j-1)
         write(&column[j], T[k+1][j], (n-k)*ELESIZE);
      else for(b=j+1; b<=n; b=b+1)
         write(&column[j],T[k+1][b], (n-k)*ELESIZE);
   }
}
dag Tdag{
   int k, j;
   for(k=1; k<n; k=k+1)
      for(j=k+1; j<=n; j=j+1)
         eval_task(T[k][j] );
}
```

Figure 0.21 PYRROS task specification for the T-DAG.

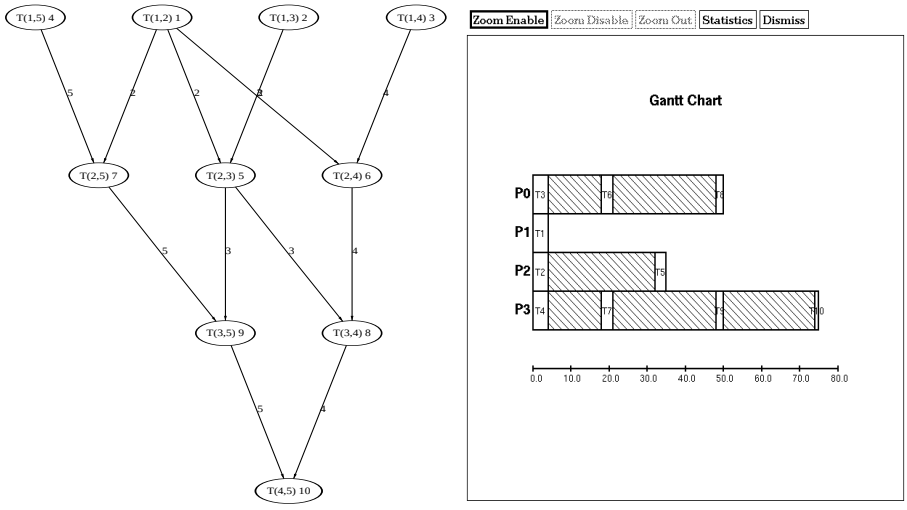


Figure 0.22 The left part is a GE DAG with $n = 5$ displayed in PYRROS X window screen. The right part is a Gantt chart using natural clustering.

because the width of the graph is 4 parallel tasks, the parallel time will be 75 time units shown in the right part of Fig. 0.22 after mapping clusters to processors. The striped lines in this Gantt chart represent communication delay on the hypercube with $p = 4$ processors. The internal numbers of tasks are used in the Gantt chart.

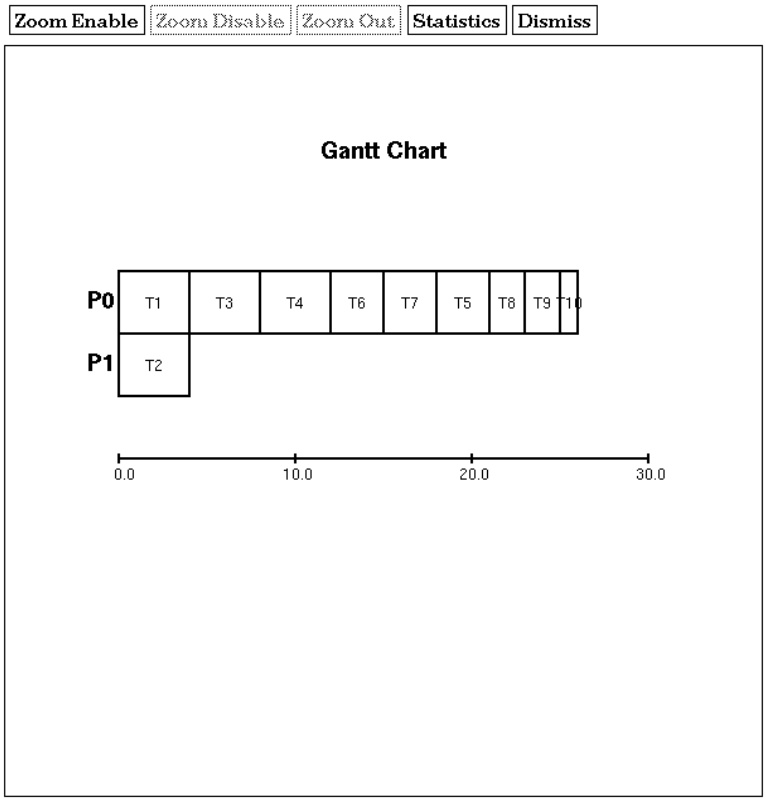


Figure 0.23 The automatic scheduling result by PYRROS.

On the other hand, if the scheduling is determined automatically by PYRROS a better utilization of the architecture and shorter parallel time can be accomplished. In Fig. 0.23 PYRROS using the DSC algorithm determines that $p = 2$ processors are sufficient for scheduling this task graph and the parallel time is reduced to 26 time units. The reason that natural clustering performs poorly here is that the graph is fine grain. Thus PYRROS is useful in determining the number of processors suitable for executing a task graph. This demonstrates one advantage of an automatic scheduling system.

0.5.3 Experiments with PYRROS

We report our experiments on the BLAS-3 GE program in nCUBE-II. The dependence graph is similar to the one in Fig. 0.4 except that tasks operate on submatrices instead of array elements. The hand-written program uses the data column block partitioning

Table 0.1 The speedup ratio of PYRROS over the sequential GE program. The block size is the dimension of a submatrix.

	n=450 block size=5	n=450 block size=10	n=1000 block size=10
p=2	1.97	1.9	1.99
p=4	3.8	3.7	3.9
p=8	7.3	6.9	7.8
p=16	12.8	11.9	14.4
p=32	19.0	12.9	25.7

with cyclic wrap mapping along the gray code of a hypercube following the algorithm of Moler [Mol86] and Saad [Saa86]. Tasks that modify the same column block are mapped in the same processor. The broadcasting uses a function provided by the nCUBE-II library. The extra memory storage optimization for the hand-made program is not used to avoid the management overhead and as a consequence the maximum matrix size that this simple program can handle is $n = 450$.

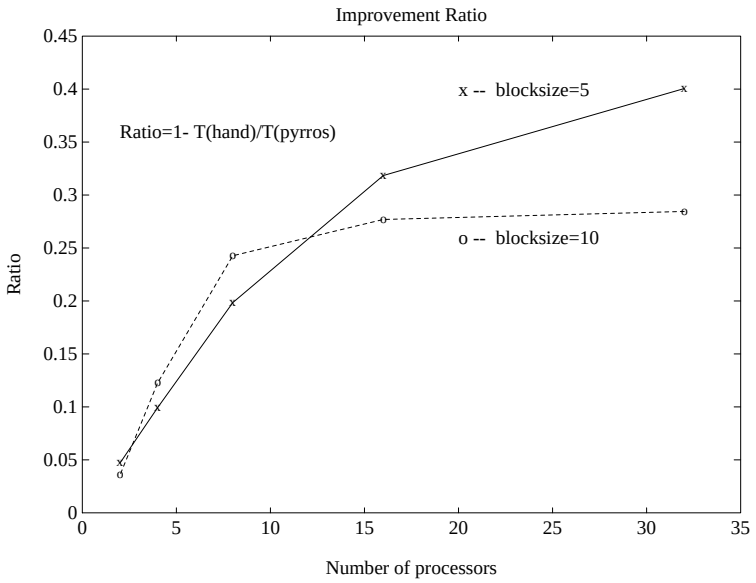


Figure 0.24 The improvement ratio of PYRROS over a hand-written GE program on nCUBE-II.

The performance improvement of PYRROS code over this hand-written program, $Ratio = 1 - PT(hand)/PT(pyrros)$, for block sizes 5 and 10 is shown in Fig. 0.24. We can see the improvement is small for $p = 2$ because each processor has enough work to do. When p increases, the PYRROS optimization plays an important role which results in 5% to 40% improvement. The speedup ratio of PYRROS over the sequential program for matrix size of 450 and 1000 is shown in Table 0.1.

0.6 Conclusions

Scheduling program task graphs is an important optimization technique for scalable MIMD architectures. Our study on the granularity theory shows that scheduling needs to take communication overhead into account especially for message passing architectures. We have described several scheduling heuristic algorithms that attain good performance in solving the scheduling problem. Those scheduling techniques are shown to be practical in PYRROS which integrates scheduling optimization with other compiler techniques to generate efficient parallel code for arbitrary task graphs.

Acknowledgments

We thank Weining Wang for developing the task graph language parser, Milind Deshpande for the X window schedule displayer, Probal Bhattacharjya for the graph generator of sparse matrix solver, and Ye Li for programming the INTEL i860 communication routines.

The work presented here was in part supported by ARPA contract DABT-63-93-C-0064 and by the Office of Naval research under grant N000149310114, and Sophia Shen Fellowship and Excellence Fellowship from Rutgers University, and by startup funds from University of California at Santa Barbara. The content of the information herein does not necessarily reflect the position of the Government and official endorsement should not be inferred.

REFERENCES

[ACD74] T. Adam, K.M. Chandy, and J. R. Dickson, A Comparison of List Schedules for Parallel Processing Systems, *CACM*, 17:12, 1974, pp. 685-690.

[Bok90] S.H. Bokhari, *Assignment Problems in Parallel and Distributed Computing*, Kluwer Academic Publisher, 1990.

[CK88] D. Callahan and K. Kennedy, Compiling Programs for Distributed-memory Multiprocessors, *Journal of Supercomputing*, Vol. 2, 1988, pp. 151-169.

[Ch89a] P. Chretienne, Task Scheduling over Distributed Memory Machines, *Proc. of Inter. Workshop on Parallel and Distributed Algorithms*, North Holland, 1989.

[Ch89b] P. Chretienne, A Polynomial Algorithm to Optimally Schedule Tasks over an ideal Distributed System under Tree-like Precedence Constraints, *European Journal of Operational Research,* 2:43 (1989), pp225-230.

[CD73] E. G. Coffman and P. J. Denning, *Operating Systems Theory*, Prentice Hall, 1973.

[Cos88] M. Cosnard, M. Marrakchi, Y. Robert, and D. Trystram, Parallel Gaussian Elimination on an MIMD Computer, *Parallel Computing*, vol. 6, 1988, pp. 275-296.

[DS87] J. J. Dongarra and D. C. Sorensen, SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs, in *The Characteristics of Parallel Algorithms*, D.B. Gannon, L.H. Jamieson and R.J. Douglass (Eds), MIT Press, 1987, pp363-394.

[Dun91] T. H. Dunigan, Performance of the INTEL iPSC/860 and nCUBE 6400 Hypercube, ORNL/TM-11790, Oak Ridge National Lab., TN, 1991.

[EL90] H. El-Rewini and T. G. Lewis, Scheduling Parallel Program Tasks onto Arbitrary Target Machines, *Journal of Parallel and Distributed Computing*, Vol. 9, 1990, pp. 138-153.

[GH86] Geist, G.A. and Heath,M.T., Matrix Factorization on a Hypercube Multiprocessor, *Hypercube Multiprocessors*, SIAM, 1986, pp. 161-180.

[GN89] A. Gerasoulis and I. Nelken, Static Scheduling for Linear Algebra DAGs, *Proc. of HCCA 4*, 1989, pp. 671-674.

[GY93] A. Gerasoulis and T. Yang, On the Granularity and Clustering of Directed Acyclic Task Graphs, *IEEE Trans. on Parallel and Distributed Systems*, Vol. 4, No. 6, June 1993, pp. 686-701.

[GY92] A. Gerasoulis and T. Yang, A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors', *Journal of Parallel and Distributed Computing,* special issue on scheduling and load balancing, Vol. 16, No. 4, pp. 276-291 (Dec. 1992).

[Geo86] A. George, M.T. Heath, and J. Liu, Parallel Cholesky Factorization on a Shared Memory Processor, *Lin. Algebra Appl.*, Vol. 77, 1986, pp. 165-187.

[GP88] M. Girkar and C. Polychronopoulos Partitioning Programs for Parallel Execution, *Proc. of ACM Inter. Conf. on Supercomputing*, St. Malo, France, 1988.

[HVV92] J.A. Hoogeveen, S.L. Van de Velde, and B. Veltman, Complexity of scheduling multiprocessor tasks with prespecified processor allocations, CWI, Report BS-R9211 June 1992, Netherlands.

[HR88] M.T. Heath and C. H. Romine, Parallel Solution of Triangular Systems on Distributed Memory Multiprocessors, *SIAM J. Sci. Statist. Comput.*, Vol. 9, 1988, pp. 558-588.

[HKT91] S. Hiranandani, K. Kennedy, and C.W. Tseng, Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines, *Proc. of Supercomputing '91*, IEEE, pp. 86-100.

[KM90] C. Koelbel and P. Mehrotra, Supporting Shared Data Structures on Distributed Memory Architectures, *Proc. of ACM SIGPLAN Sympos. on Principles and Practice of Parallel Programming*, 1990, pp. 177-186.

[LR78] J.K. Lenstra and A.H.G. Rinnooy Kan, Complexity of Scheduling under Precedence Constraints, *Operation Research,* 26:1 (1978).

[KB88] S.J. Kim and J.C. Browne, A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures, *Proc. of Inter. Conf. on Parallel Processing*, Vol. 3, 1988, pp. 1-8.

[Kun88] S.Y. Kung, *VLSI Array Processors*, Prentice Hall, 1988.

[Mol86] C.Moler, Matrix Computation on Distributed Memory Multiprocessors, *Hypercube Multiprocessors 1986*, SIAM, pp. 181-195.

[Ort88] J. M. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum (New York), 1988.

[PY90] C. Papadimitriou and M. Yannakakis, Towards on an Architecture-Independent Analysis of Parallel Algorithms, *SIAM J. Comput.*, Vol. 19, 1990, pp. 322-328.

[Pic92] C. Picouleau, New complexity results on the UET-UCT scheduling algorithms, *Proc. of Summer School on Scheduling Theory and its Applications,* Chateau De Bonas, France, 1992, pp. 487-502.

[Pol90] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten, The Structure of Parafrase-2: an Advanced Parallelizing Compiler for C and Fortran, in *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nicolau and D. Padua (Eds.), 1990.

[Sal90] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman, Run-Time Scheduling and Execution of Loops on Message Passing Machines, *Journal of Parallel and Distributed Computing,* Vol. 8, 1990, pp. 303-312.

[Saa86] Y. Saad, Gaussian Elimination on Hypercubes, in *Parallel Algorithms and Architectures,* Cosnard, M. et al. (Eds.), Elsevier Science Publishers, North-Holland, 1986.

[Sar89] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*, MIT Press, 1989.

[Sto87] H. Stone, *High-Performance Computer Architectures*, Addison-Wesley, 1987.

[VRK93] T. Varvarigou, V. Roychowdhury, and T. Kailath, Scheduling In and out Forests in the Presence of communication delays, to appear in *IEEE Trans. on Parallel and Distr. Systems.*, Short version appeared in *Proc. of International Parallel Processing Symp.*, April 1993, CA.

[WG88] M.Y. Wu and D. Gajski, A Programming Aid for Hypercube Architectures, *Journal of Supercomputing*, Vol. 2, 1988, pp. 349-372.

[YG91] T. Yang and A. Gerasoulis, A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors, *Proc. of Supercomputing '91*, IEEE, pp. 633-642. A longer version will appear in *IEEE Trans. on Parallel and Distributed Systems.*

[YG95] T. Yang and A. Gerasoulis, List Scheduling with and without Communication Delay, To appear in *Parallel Computing*, 1995.

[YG92] T. Yang and A. Gerasoulis, PYRROS: Static Task Scheduling and Code Generation for Message-Passing Multiprocessors, *Proc. of 6th ACM Inter. Confer. on Supercomputing,* Washington D.C., 1992, pp. 428-437.