SRIKANTH BHATTIPROLU
TAP : A TOOL FOR EVALUATING PROCESSOR ASSIGNMENTS
IN TASK AND DATA PARALLEL PROGRAMS
(Under the direction of DAVID LOWENTHAL)

A parallel program is usually written using either data parallelism or task parallelism. With data parallelism, each processor executes the same code but operates on different data, whereas in task parallelism the program is divided into tasks, and each task is executed on a separate processor. Task and data parallelism have complementary strengths that are appropriate for different problems. However, some programs can benefit from their integration.

One key problem with an integrated approach is the *processor assignment problem*, which is how to assign the different processors to tasks. Determining a good processor assignment is complex and often requires significant experimentation. We have developed a tool called TAP for executing integrated parallel programs with user defined processor assignments. This thesis describes the user interface and implementation details of TAP. We also discuss how the framework was used to study the behavior of PHOENIX, a scientific stellar atmospheric code.

INDEX WORDS: Task parallelism, Data parallelism, Processor assignment, Theses (academic)

TAP : A TOOL FOR EVALUATING PROCESSOR ASSIGNMENTS IN TASK AND DATA PARALLEL PROGRAMS

by

SRIKANTH BHATTIPROLU

B.E., Osmania University, India, 1998

A Thesis Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2001

© 2001

Srikanth Bhattiprolu

All Rights Reserved

TAP : A TOOL FOR EVALUATING PROCESSOR ASSIGNMENTS IN TASK AND DATA PARALLEL PROGRAMS

by

SRIKANTH BHATTIPROLU

Approved:

Major Professor: David Lowenthal

Committee: Eileen T. Kraemer
Surendar Chandra

Electronic Version Approved:

Gordhan L. Patel
Dean of the Graduate School
The University of Georgia
August 2001

ACKNOWLEDGMENTS

I would like to thank Dr. David Lowenthal for the immense help he has provided me in completing this work. I would also like to thank Dr. Eileen Kraemer and Dr. Surendar Chandra for consenting to serve on my committee.

I would like to thank my family who have to stood by me whenever I have needed them the most.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS iv

LIST OF FIGURES vi

CHAPTER

1 INTRODUCTION 1

2 OVERVIEW 4

2.1 TASK AND DATA PARALLELISM 4

2.2 PROCESSOR ASSIGNMENT 7

3 IMPLEMENTATION 12

3.1 EMULATOR FRAMEWORK 12

3.2 SINGLE GRAPH IMPLEMENTATION 13

3.3 MULTIPLE GRAPH IMPLEMENTATION 18

4 PERFORMANCE 26

4.1 OUT OF CORE TESTS 26

4.2 PHOENIX 32

4.3 DISCUSSION 41

5 RELATED WORK 42

6 CONCLUSIONS AND FUTURE WORK 45

BIBLIOGRAPHY 47

List of Figures

2.1 Sample Task Graph. A task is eligible to run when all of its predecessors have completed. 6

2.2 Sample code for a task in Figure 2.1. 7

2.3 Code for each processor using data parallelism. The pieces of X, Y and Z are independent for each processor. 7

2.4 Example pseudo code for tasks shown in Figure 2.1. Tasks T3, T4 and T5 work on different data sets. T4 does not have any predecessor and hence does not have a *getInput* function. 9

2.5 Sample code for a task on a single processor with data parallelism for the task graph shown in Figure 2.1. Each processor works on a piece of data of all tasks. 10

2.6 Task subgraph. Modified task graph with tasks T3, T4 and T5 of Figure 2.1 being considered for performance study. 11

3.1 Code describing the mapping between task list in the file and the tasks in the program. 14

3.2 List of tasks on a processor. Linked list used to store the list of tasks on a processor. The values given are for task T3 whose taskId is 3, is executed on 3 processors, processor 0 being chosen as the group leader. The task has no previous tasks and has a successor task T5. 16

3.3 Modified user code for a task. 17

3.4 Task Graph With Replication 18

3.5 Example code for instances of T3 shown in Figure 2.6. Note that T3 and T3' read the same data set. 19

3.6 Example describing blocking factor applied to task graph shown in Figure 3.4. 21

3.7 Code describing the mapping between task list in the file and the tasks in the program. 22

3.8 Task execution order for the tasks in Figure 3.4 with a processor assignment in Table 3.4. 23

3.9 List of tasks on a processor. Modified task list structure for handling replicas. The values shown are for executing the T3" and T3"', which are instances of T3, on 3 processors. 24

4.1 Task graph for testing the performance of a parallel program when it becomes out of core. 27

4.2 Sample data parallel code used for testing out-of-core behavior. 27

4.3 Replicated tasks graph. T1, T1' etc are instances of T1 and T2, T2' etc are instances of T2. Instances of a particular task work on the same data set. 30

4.4 Synthetic code executed by each replica in Figure 4.3. Each task does some simple array addition. All arrays are one dimensional. 30

4.5 PHOENIX task graph 33

CHAPTER 1

INTRODUCTION

Many applications such as weather prediction and aerodynamics are computationally intensive and require vast amounts of processing power. For example, to calculate a 24 hour weather forecast for the UK requires about 10^{12} operations to be performed. This takes about 2.7 hours on a Cray-1 (capable of 10^8 operations per second). For giving accurate long range forecasts more powerful processors are needed. Though the processing power of the machines has been increasing every year, it will still not be able to meet the requirements of some applications. One solution is to use *parallelism*, which means simultaneously running the application on many processors and reduce the computation time. This requires new algorithms and program structures to perform operations simultaneously, because most existing algorithms are specialized for a single processor. *Concurrency*, ability to execute instructions simultaneously on different processors, becomes a fundamental requirement for parallel algorithms.

The first step in developing these algorithms is to develop a parallel machine model, similar to the existing single machine model comprising of a CPU and memory [6]. Different models had been proposed for the parallel machine, the significant ones being the Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD). In SIMD machines, all processors execute the same instruction stream on a different piece of data. MIMD machines execute a separate stream of instructions on their local data.

Once the model is chosen, communication channels between different processors need to be set up for exchange of data. The communication channels are usually implemented

by using *shared memory* or *message passing*. In the shared memory approach there is global address space, which is accessible to all the processors; in message passing the memory is distributed and the processors are connected by a network for exchanging messages. In addition to the extreme cases of shared memory and distributed memory there are possibilities for hybrid designs that combine features of both.

A parallel computation consists of one or more tasks. Tasks can be mapped to physical processors in various ways; the mapping employed does not affect the semantics of a program. In particular, multiple tasks can be mapped to a single processor. The mapping of the tasks to processors requires a careful design based on the computation and communication involved in the program. There are two approaches for how the mapping to processors is done. In the first one, we write a single program and execute it on many processors, with each processor working on a piece of data. This approach is called data parallelism or SPMD (Single Program Multiple Data). The second idea is to write multiple programs, where each one is responsible for a special task. These different programs are then executed in parallel. We also call this approach task parallelism or MPMD (Multiple Program Multiple Data).

Task and data parallelism are often considered mutually exclusive approaches to parallel programming. For many applications, achieving good performance for a parallel program requires exploiting *both* data parallelism as well as task parallelism. Depending on the size of the input data set and the number of processors, the application is divided into a set of data parallel tasks, which can be executed concurrently on.

This project focuses on integrated task and data parallel programs. They consist of a collection of tasks, some of which are independent; additionally, some tasks are themselves parallel. A good *processor assignment*, which is a mapping of each task onto a set of processors, is crucial for parallel program performance. The goal of this project is to explore effective processor assignments in such programs, taking into account computation time, the memory hierarchy, and communication constraints. We developed a simu-

lation environment for testing different processor assignments for a parallel program. In this thesis we describe how the framework can be utilized for studying the behavior of synthetic parallel programs and also present results to show the behavior of a scientific application called PHOENIX. The latter program is *out of core*, meaning that its data set does not fit in the aggregate memory of all processors.

In the following sections we describe the TAP (Tool for Assigning Processors) which is used for testing different processor assignments on parallel programs. The paper is organized as follows. Chapter 2 gives the fundamental motivation for developing the emulator and introduces an example parallel program, which is re-used throughout the paper to illustrate how the emulator can be used. Chapter 3 gives the user interface and the implementation details for using the emulator. Chapter 4 gives results that are obtained by using the emulator program on the example program and a scientific application PHOENIX. For the example program, we got up to 60% improvement in performance by using integrated approach when compared to data parallelism. Using our horizontal execution approach we got upto seven-fold improvement testing a synthetic PHOENIX application. Chapter 5 describes the related work, and we conclude and discuss future work in Chapter 6.

Chapter 2

Overview

This section discusses how TAP can be used for testing different processor assignments in parallel programming design. Section 2.1 explains task and data parallelism. Section 2.2 explains the application of the emulator.

2.1 Task and Data Parallelism

One key part of parallel program design is the partitioning stage as was explained by Foster [6], where the execution is divided into smaller parts to be executed on different processors. A small portion of the program which can be independently executed on a processor is called a *task*. A parallel program is usually partitioned into tasks to expose opportunities for parallel execution. Hence, the focus is on defining a large number of small tasks in order to yield what is termed a fine-grain decomposition of a problem. In later design stages, evaluation of communication requirements, the target architecture, or software engineering issues may lead us to forego opportunities for parallel execution identified at this stage. We then revisit the original partition and agglomerate tasks to increase their granularity. A good partition divides into small pieces both the computation associated with a problem and the data on which this computation operates. When the focus is on the data associated with a problem, the technique is termed domain decomposition. The alternative approach, where the focus is on the computation to be performed, is termed functional decomposition.

Data parallelism uses domain decomposition for achieving parallelism. In this approach to problem partitioning, we seek first to decompose the data associated with a problem. If possible, we divide these data into small pieces of approximately equal size. Next, we partition the computation that is to be performed, typically by associating each operation with the data on which it operates. This partitioning yields a number of tasks, each comprising some data and a set of operations on that data. An operation may require data from several tasks. In this case, communication is required to move data between tasks.

Figure 2.1 gives an example of a parallel program divided into different tasks. The order of execution of tasks depends on the dependence graph. An example code fragment is shown in Figure 2.2. Table 2.1 gives the processor assignment with 4 processors. The data and computation are distributed equally among the processors for achieving good performance. Figure 2.3 gives the modified code for each processor. Data parallelism gives good performance when the computation is regular but has the disadvantage that it does not scale well beyond a certain point.

Tasks	*Processor*
T1, T2, T3, T4	1
T1, T2, T3, T4	2
T1, T2, T3, T4	3
T1, T2, T3, T4	4

Table 2.1: Processor assignment with data parallelism. Each processor works on all the tasks.

Task parallelism is achieved by functional decomposition. This approach represents a different and complementary way of thinking about problems. In this approach, the initial focus is on the computation that is to be performed rather than on the data manipulated by the computation. If we are successful in dividing this computation into disjoint tasks, we proceed to examine the data requirements of these tasks. These data requirements may be disjoint, in which case the partition is complete. In the case of task parallelism

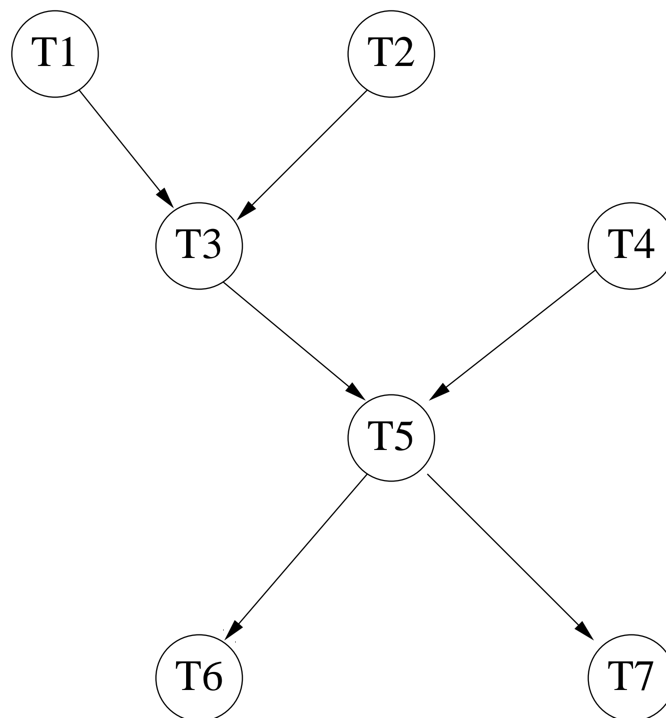


Figure 2.1: Sample Task Graph. A task is eligible to run when all of its predecessors have completed.

the tasks are assigned to run on different processors. For the task dependency graph in Figure 2.1, a sample processor assignment is given in Table 2.2. Pure task parallelism has the disadvantage that it cannot exploit the data parallelism involved in each task.

Integrating task and data parallelism may provide better performance in certain cases. The program is divided into tasks using functional decomposition and a set of processors are assigned to each task. Data parallelism in each task can be applied for domain decomposition. The integrated approach tries to get the best of both the domain and functional decomposition. A lot of research is being done to effectively use both the forms of decomposition to get the proper assignment of different number of processors for the

```
T()
{
 // F() works on X, Y and produces Z
 Z = F ( X, Y );
}
```

Figure 2.2: Sample code for a task in Figure 2.1.

```
T()
{
 // F() works on X', Y' and pro-
duces Z'
 // X', Y', Z' are pieces of X, Y and Z
 Z' = F ( X', Y' );
}
```

Figure 2.3: Code for each processor using data parallelism. The pieces of X, Y and Z are independent for each processor.

given parallel program. This project addresses these class of programs and develops a testing framework to find an effective processor assignment.

2.2 Processor Assignment

Processor assignment in a task and data parallel program requires complex analysis and requires test runs to get the optimal assignment. This could involve considerable time spent for getting an efficient assignment. In the case of large parallel programs whose execution may take a few days, this may not be feasible. Figure 2.1 gives a task graph for a parallel program with complex task dependencies. The processor assignment in these type of complex graphs becomes quite difficult because of the different constraints

Tasks	*Processor*
T1, T3	1
T2	2
T4, T7	3
T5, T6	4

Table 2.2: Sample processor assignment for task parallelism

involved. The computation involved in the task, the amount of memory utilized by the task, and its dependencies with other tasks influence the assignment. Determining an optimal or even effective assignment becomes quite difficult.

The project addresses this issue by developing a testing framework used for running a parallel program with different processor assignments. TAP is an emulator tool for running a part of the application or even a scaled-down version of the actual parallel program. The results can be extrapolated to identify effective processor assignments.

For example, consider a parallel program with dependency graph in Figure 2.1, whose tasks T3, T4, T5 have the computation shown in Figure 2.4. The *getInput* and *enable-Output* depend on the task dependencies of each task. For example, T3 receives input from tasks T1, T2 before it starts executing. T4 has no predecessor, as can be seen from the task dependency graph in Figure 2.1. The task T5 collects the results from T3, T4 for its computation. In many cases, these tasks utilize large amounts of memory to store arrays. Each of these tasks are data parallel, which make them good candidates for domain decomposition. Figure 2.5 gives the modified code for the computation involved in each processor.

The rest of the tasks work on smaller data sets and are less computationally intensive. Hence the performance of tasks T3, T4, T5 determine the overall performance of the

```
T3()
{
 getInput(T1, T2);

 XC = F ( XA, XB );

 enableOutput(T5);
}
T4()
{
 XF = F ( XD, XE  );

 enableOutput(T5);
}
T5()
{
 getInput(T3, T4);

 XG = F ( XC, XF );

 enableOutput(T6, T7);
}
```

Figure 2.4: Example pseudo code for tasks shown in Figure 2.1. Tasks T3, T4 and T5 work on different data sets. T4 does not have any predecessor and hence does not have a *getInput* function.

application. So it may be useful to study the performance of these tasks independently. For the dependency graph in Figure 2.1, if we are interested in seeing the performance of tasks T3, T4, T5, we can use the emulator to run the program with these tasks, resulting in a new task graph as represented in Figure 2.6. Different processor assignments can be tested on these tasks to get a good performance estimate. The results can be used for determining a processor assignment for the original application.

```
compute(start, end)
{
 for i = start to end
   Z = F ( X[start:end], Y[start:end]);
}
```

Figure 2.5: Sample code for a task on a single processor with data parallelism for the task graph shown in Figure 2.1. Each processor works on a piece of data of all tasks.

In the case when the emulator is used to run a few tasks instead of the whole program, the user needs to modify the application such that the the tasks can run without having to execute other predecessor tasks. An example of this might be a case where the task T3 requires the result of the execution of tasks T1 and T2. This situation needs to be simulated by the user so that T3, T4 and T5 can run independently.

The amount of data involved in each task can be scaled down to make it suitable to run the program on a particular machine. Scientific parallel programs often work on gigabytes or terabytes of data and may be using hundreds of machines during their final runs. But it may be a good idea to scale down the application to run on a few workstations to get initial results. For the example shown above, the tasks T3, T4, T5, can be modified to work on smaller array sizes suitable for the test environment.

In the next section we explain how TAP can be used for testing different processor assignments. We will discuss the user interface and the implementation details of the emulator for running parallel programs.

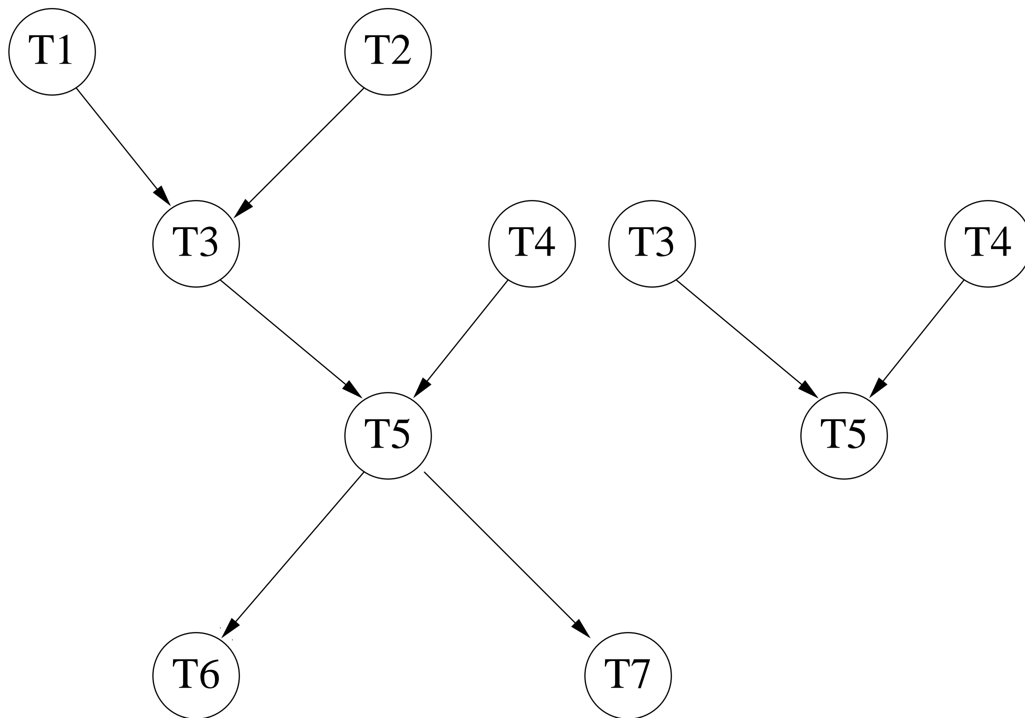


Figure 2.6: Task subgraph. Modified task graph with tasks T3, T4 and T5 of Figure 2.1 being considered for performance study.

CHAPTER 3

IMPLEMENTATION

The emulator is a testing framework used for performance analysis of a parallel program that contains both task and data parallelism and specifically examines how different processor assignments affect the performance of a parallel program. In the following sections we will describe what the emulator does, its user interface and the implementation details.

3.1 EMULATOR FRAMEWORK

The emulator can be used for running a scaled-down version of a parallel program with different processor assignments. This may not be able to give an exact performance picture of an application, but gives good estimates of the behavior of the application. The results can be used to determine the best possible configuration of the available processors for running a parallel program efficiently.

The emulator requires the following as the input from the user.

1. *Task List* - the list of tasks in the program.

2. *Task Dependencies* - the order of execution of the tasks.

3. *Processor Configuration* - list of processors assigned for each task.

From the information provided, the emulator runs the tasks in the order specified by the user on the processors assigned to that task. The emulator can be used for running programs with a single graph as in Figure 2.1. It can also be used when a single graph is replicated as in PHOENIX, explained in Section 3.3.

3.2 SINGLE GRAPH IMPLEMENTATION

The emulator runs the parallel program with a user specified processor assignment. The different steps involved for using the emulator and modifying the parallel program are summarized below.

1. Identify the tasks in the parallel program.

2. Identify the portion of the parallel program code to be tested with the emulator.

3. Scale down the parallel program for the test environment if required.

4. Specify the task dependencies and the processor assignment.

5. Modify the parallel program to run along with the emulator. This requires an understanding of the emulator code.

The first three steps are explained in Section 3.2.1, while the last two are covered in Section 3.2.2.

3.2.1 USER INTERFACE

The emulator program requires very few modifications for running parallel programs. With a good understanding of the emulator code, it can be modified quite easily to suit most applications with regular execution paths. The following sections describe how the user needs to provide information to the emulator about the task dependencies and the processor assignment.

TASK LIST AND DEPENDENCIES

Its the job of the user to identify the tasks and provide a mapping between the task list in the file and the tasks in the program. The mapping needs to be coded into the emulator program as in the Figure 3.1 for tasks T3, T4, T5 shown in Figure 2.6.

```
taskId = 3; // Used for assigning Task ID's
taskVector[taskId++] = T3();
taskVector[taskId++] = T4();
taskVector[taskId++] = T5();
```

Figure 3.1: Code describing the mapping between task list in the file and the tasks in the program.

The *taskVector* is an array of function pointers for each task in the program. The emulator uses the *taskId* provided from the input task dependency file to execute a particular task. As we can see from the code above, the tasks T3, T4, T5 have taskId's 3, 4, 5 respectively. The task dependencies are provided to the emulator through a specification file. A sample format of the specification file for the modified task graph of Figure 2.6 is given in Table 3.1. The main purpose of the task graph is to provide run-time information about the order of execution of tasks to the emulator.

TaskCode	*NumPrev*	*PrevTasks*	*NumNext*	*NextTasks*
3	0		1	5
4	0		1	5
5	2	3 4	0	

Table 3.1: Task Dependency Graph Representation in TAP. *TaskCode* identifies the task, *NumPrev* is the number of previous tasks, and *PrevTasks* gives the list of the previous tasks. *NumNext* is the number of tasks after the particular task, and *NextTasks* gives the list of next tasks.

Processor Assignment

The different processor assignments to each of the tasks needs to be given to the emulator program from a file in the format specified in Table 3.2. The processor assignment shown

in Table 3.2 executes all the tasks on a single processor. The example shown is a simple one, but complex processor assignments, based on the computation and communication involved in each task can also be specified, see Table 3.3.

TaskCode	*NumProcs*	*ProcList*
3	1	0
4	1	0
5	1	0

Table 3.2: Sample processor assignment in TAP. *TaskCode* identifies the task, and *NumProcs* is the number of processors to be run on. *ProcList* gives the ranks for processors used; MPI assigns ranks to the processors that are used for running the program.

TaskCode	*NumProcs*	ProcList
3	3	0 1 2
4	1	3
5	2	4 5

Table 3.3: Sample processor assignment in TAP for tasks T3, T4 and T5 in Figure 2.6.

3.2.2 Implementation

The task graph and the processor assignment is stored in a data structure in the emulator code and is replicated among processors running the parallel programming code. The data structure is shown in Figure 3.2. It is initialized based on the task dependencies and processor assignment provided by the user through the specification files. The emulator uses the information to schedule the tasks on different processors. As explained in the Section 3.2.1, the *taskId*, which identifies the task, is used to index into the array of function pointers to execute the task.

The emulator executes the tasks depending on the task dependencies. For the specification in Figure 2.6, T3 and T4 are executed before T5. Since there is no dependency between T3 and T4, they are executed in parallel. If they are both running on the same processor, either ordering is legal. For the processor assignment in Table 3.2, T3 runs before T4.

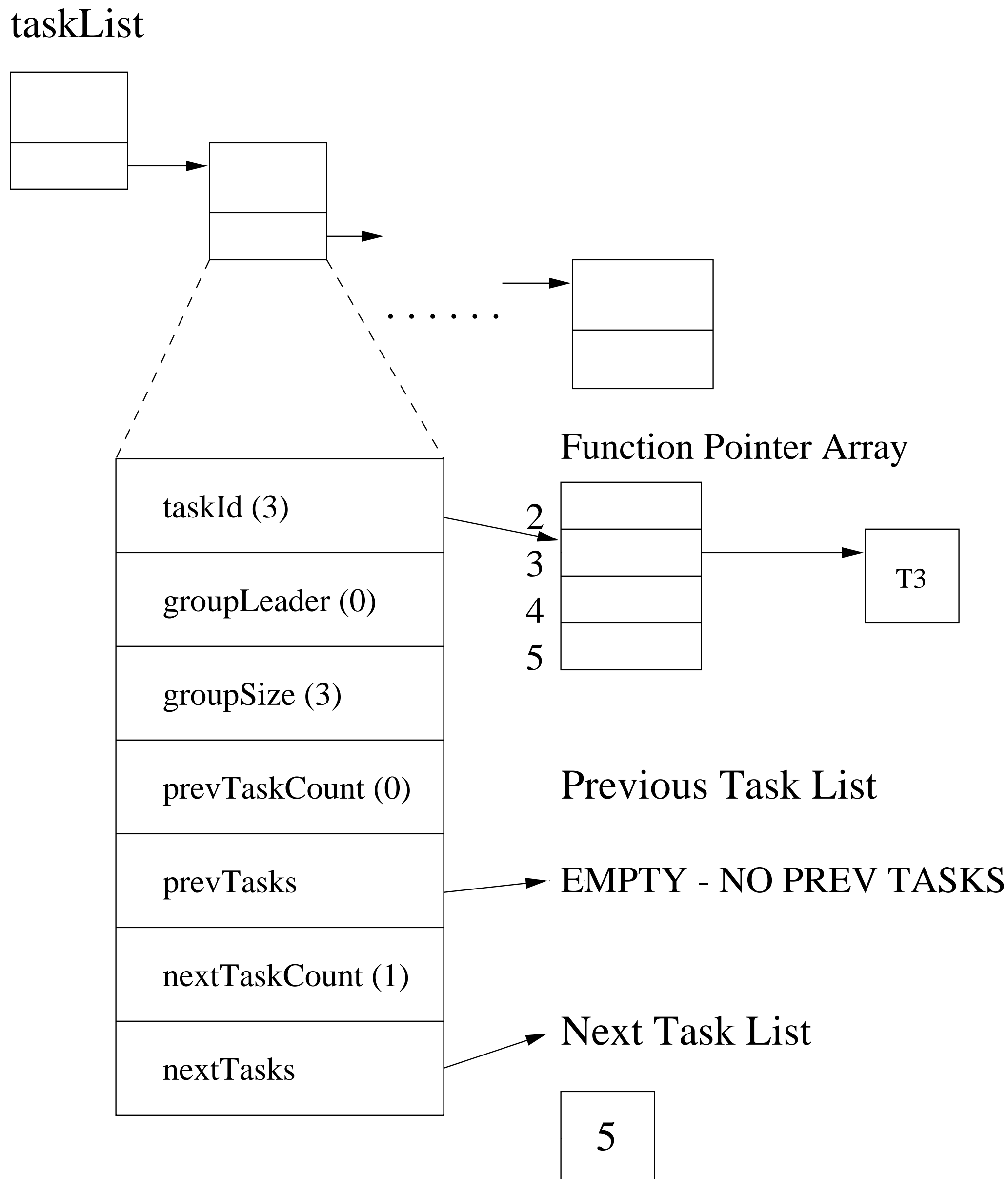


Figure 3.2: List of tasks on a processor. Linked list used to store the list of tasks on a processor. The values given are for task T3 whose taskId is 3, is executed on 3 processors, processor 0 being chosen as the group leader. The task has no previous tasks and has a successor task T5.

When a task is scheduled on a list of processors, a processor is designated as the group leader. The tasks communicate with each other for signaling the start and end of a particular task through the group leaders. The sample pseudo code in Figure 3.3 explains how the user code is modified so that it can run along with the emulator for the dependency graph of Figure 2.6 and processor configuration in Table 3.3.

```
T()
{
  if(myrank == group leader) {
    for each t = predecessor task {
      G = group leader of t;
      receive(G, "go");
    }
    distribute data to group;
  } else {
    receive data from group leader;
  }

  set start and end;

  compute(start, end); // data parallelism (See 'Overview' Chapter)

  if(myrank == groupleader) {
    for each processor in group {
      receive results;
    }

    for each t = successor task {
      G = groupleader of t;
      send(G, "go");
    }
  } else {
    send results to groupleader;
  }
}
```

Figure 3.3: Modified user code for a task.

The emulator provides to the parallel program the global data structures that were allocated from the user input. The processor list for each task is used in distributing the data and computation among the processors assigned to the task. The group leader for each task takes care of distributing data at synchronization points and also communicates with leaders of other tasks. As we can see from the pseudo code specified above, the number of changes is relatively small. The modifications for the tasks are similar irrespective of what the task does.

3.3 Multiple Graph Implementation

The basic emulator framework explained above can be used only for running parallel programs with a single task graph as in Figure 2.6. More features were added to the emulator to run parallel programs with a task graph as in Figure 3.4. For the example parallel program chosen (PHOENIX explained in Section 4.2 has similar characteristic) with task dependency shown in Figure 3.4, the tasks T3', T3", and T3"' are related to T3 because they work on the same arrays A, B and C but perform different computation. The tasks T3', T3" and T3"' are called *instances* of T3 because they are independent but read the same data set. Similarly T4', T4", T4"' and T5', T5", T5"' are instances of T4 and T5 respectively. A sample computation code for instances of T3 is given in Figure 3.5.

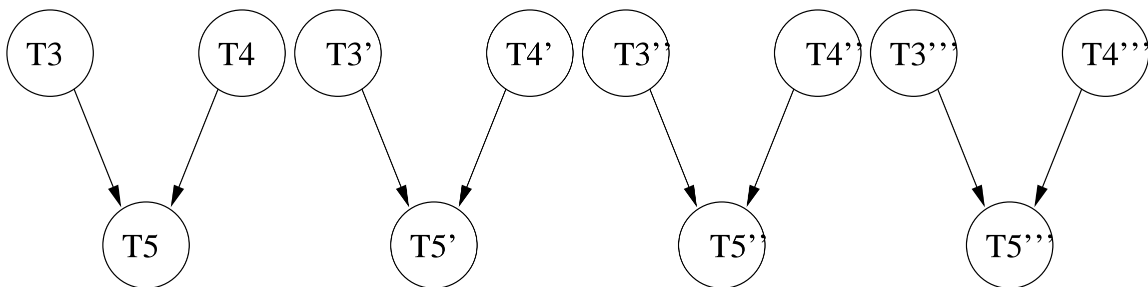


Figure 3.4: Task Graph With Replication

In the task graph given in Figure 3.4, it may be useful to divide the replicas among the processors depending on data locality of the tasks in each replica. The emulator can

```
computeT3(start, end)
{
 input A[start:end], B[start:end];
 output C[start:end];

 getInput(T1, T2);

 C = F ( A, B );

 enableOutput(T5);
}
computeT3'(start, end)
{
 input A[start:end], B[start:end];
 output C[start:end];

 getInput(T1, T2);

 C = F' ( A, B );

 enableOutput(T5);
}
```

Figure 3.5: Example code for instances of T3 shown in Figure 2.6. Note that T3 and T3' read the same data set.

be used to run the tasks based on these characteristics. Two factors need to be taken into consideration in this case.

1. For the example chosen, tasks T3, T3', T3", T3"' work on the same data set, and executing those tasks serially can exploit data locality. This will be useful if the amount of data involved in each task is large. Executing the tasks T3, T4, T5 in the

serial order might result in the data used by T3 being evicted by the time the task T3' starts executing, resulting in disk reads.

2. However, instances of T3 may produce large amounts of data for instances of T5. Executing all instances of T3 might evict the data produced and hence results in more disk accesses while executing T5.

We define a *blocking factor* to denote the order of execution of tasks in parallel programs with replicas as in Figure 3.4. A blocking factor of b implies that we execute b number of instances before executing another set of tasks. For the example given in Figure 3.4, if we are to execute all T3's, then all T4's and then T5's, the blocking factor is 4. Executing the tasks T3, T4, T5, T3', T4', T5', etc., in that order is blocking factor of 1. The Figure 3.6 displays different blocking factors for the task graph in Figure 3.4 and also gives a possible task execution order. As we can see from the figure the blocking factor determines the point at which the dotted line is placed in the figure. The dotted line is like a *fence*, all the tasks on the left of it need to be executed before crossing it.

We extended emulator for running parallel programs with replicas with different possible blocking factors values. In the following sections, we describe how the user interface and the implementation is extended for running this class of programs.

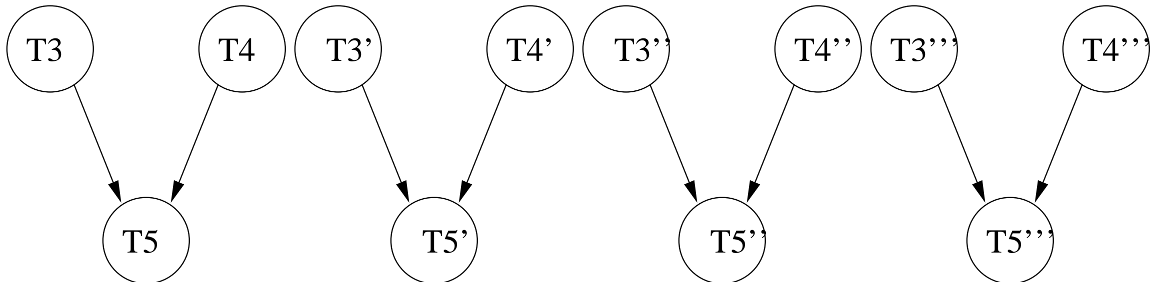
3.3.1 User Interface

The user interface specified in Section 3.2.1 is modified to represent problems similar to the ones described in Figure 3.4.

Task list and dependencies

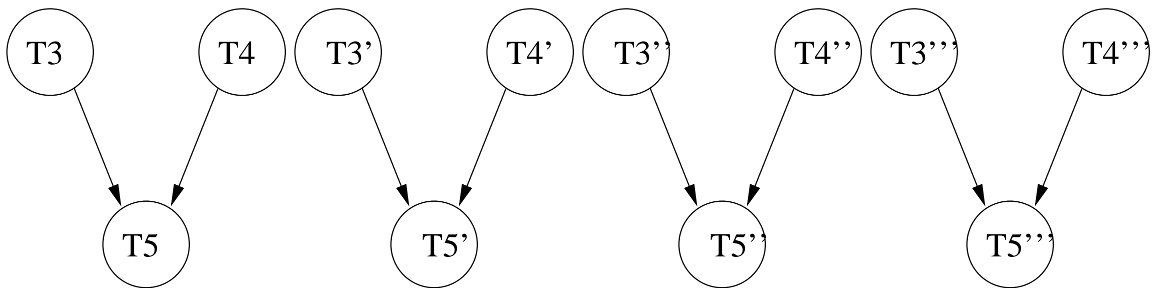
The task list and dependencies remain the same as in the basic emulator set up. The tasks T3 and the related tasks which work on the same data set are given the same taskId. They are distinguished by the replica number as shown in Figure 3.7.

Blocking Factor = 1



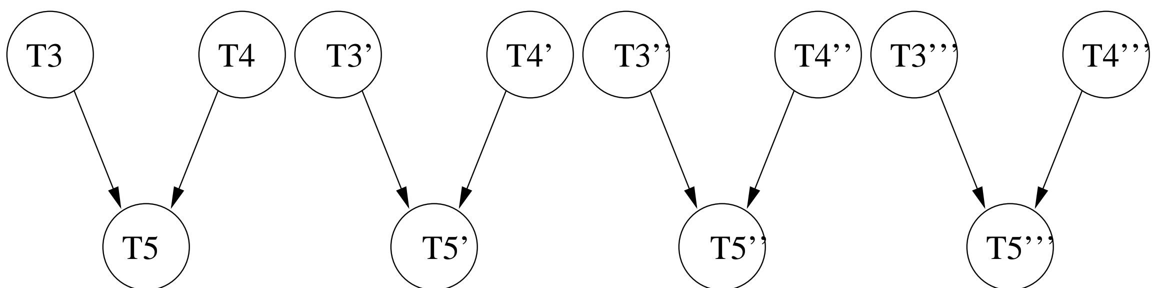
Task Execution order : T3, T4, T5, T3’, T4’, T5’, T3’’, T4’’, T5’’, T3’’’, T4’’’, T5’’’

Blocking Factor = 2



Task Execution order : T3, T3’, T4, T4’, T5, T5’, T3’’, T3’’’, T4’’, T4’’’, T5’’, T5’’’

Blocking Factor = 4



Task Execution order : T3, T3’, T3’’, T3’’’, T4, T4’, T4’’, T4’’’,T5, T5’, T5’’, T5’’’

Figure 3.6: Example describing blocking factor applied to task graph shown in Figure 3.4.

```
replicaNum = 0;

taskId = 3;
taskVector[replicaNum][taskId] = T3();
taskVector[replicaNum][taskId++] = T4();
taskVector[replicaNum][taskId++] = T5();

taskId = 3;
replicaNum++;
taskVector[replicaNum][taskId] = T3'();
taskVector[replicaNum][taskId++] = T4'();
taskVector[replicaNum][taskId++] = T5'();
```

Figure 3.7: Code describing the mapping between task list in the file and the tasks in the program.

PROCESSOR ASSIGNMENT

The specification file has an extra field to specify whether to execute the tasks in serial order or to execute the instances. The blocking factor is given as a command line option to the program. Depending on its value, the emulator determines the fences in the task graph as explained in Section 3.3. Table 3.4 gives an example of how the processor assignment for a task graph given in Figure 3.4, is specified.

Mode	*TaskCode*	*NumProcs*	ProcList
A	3	2	0 1
D	4	2	2 3
D	5	3	1 2 3

Table 3.4: Processor Assignment for task graph in Figure 3.4. *Mode* is used for determining the order of tasks, *TaskCode* identifies the task and *NumProcs* is the number of processors to be run on. *ProcList* gives ranks for processors used, MPI takes care of assigning ranks to the processors that are used for running the program.

The *Mode* column is used to determine the order of execution of tasks between two fences in the task graph. There are two different modes as explained below.

1. If the Mode is 'A', all the instances between the fences with the given taskId are executed. Assuming that the blocking factor is 2, for the example in Table 3.4, tasks with *taskId*=0 and replica number between 0 and 1 are executed first. Hence, tasks T3 and T3' are executed before executing T4.

2. If the Mode is 'D', all the tasks are executed in serial order. For the example in Table 3.4, the last 2 entries have a 'D' for *Mode*. So tasks with ID's 1 and 2, and replica numbers between 0 and 1, are executed serially, yielding an execution order of T4, T5, T4' and T5'.

The order of execution of tasks based on the task graph of Figure 3.4, processor assignment of Table 3.4 and a blocking factor of 2 is given in Figure 3.8. The new set up can be used for running a program that does not have any replicas by just setting the start and ending replica number as 0 and Mode as 'D'. The approach can be used for any regular graph as in Figure 3.4.

T3	T3'	T4	T5	T4'	T5'	T3"	T3"'	T4"	T5"	T4"'	T5"'

Figure 3.8: Task execution order for the tasks in Figure 3.4 with a processor assignment in Table 3.4.

3.3.2 Implementation Details

The user must hardcode all the tasks with the replica number and the taskId as shown in Section 3.3.1. The structure specified in Figure 3.2, is modified to Figure 3.9, to store information about the replicas. Three extra fields, *direction*, *startRepl* and *endRepl* are added to the data structure. Based on the value of blocking factor, *startRepl* and *endRepl* are set to the start and the ending replica numbers between two fences.

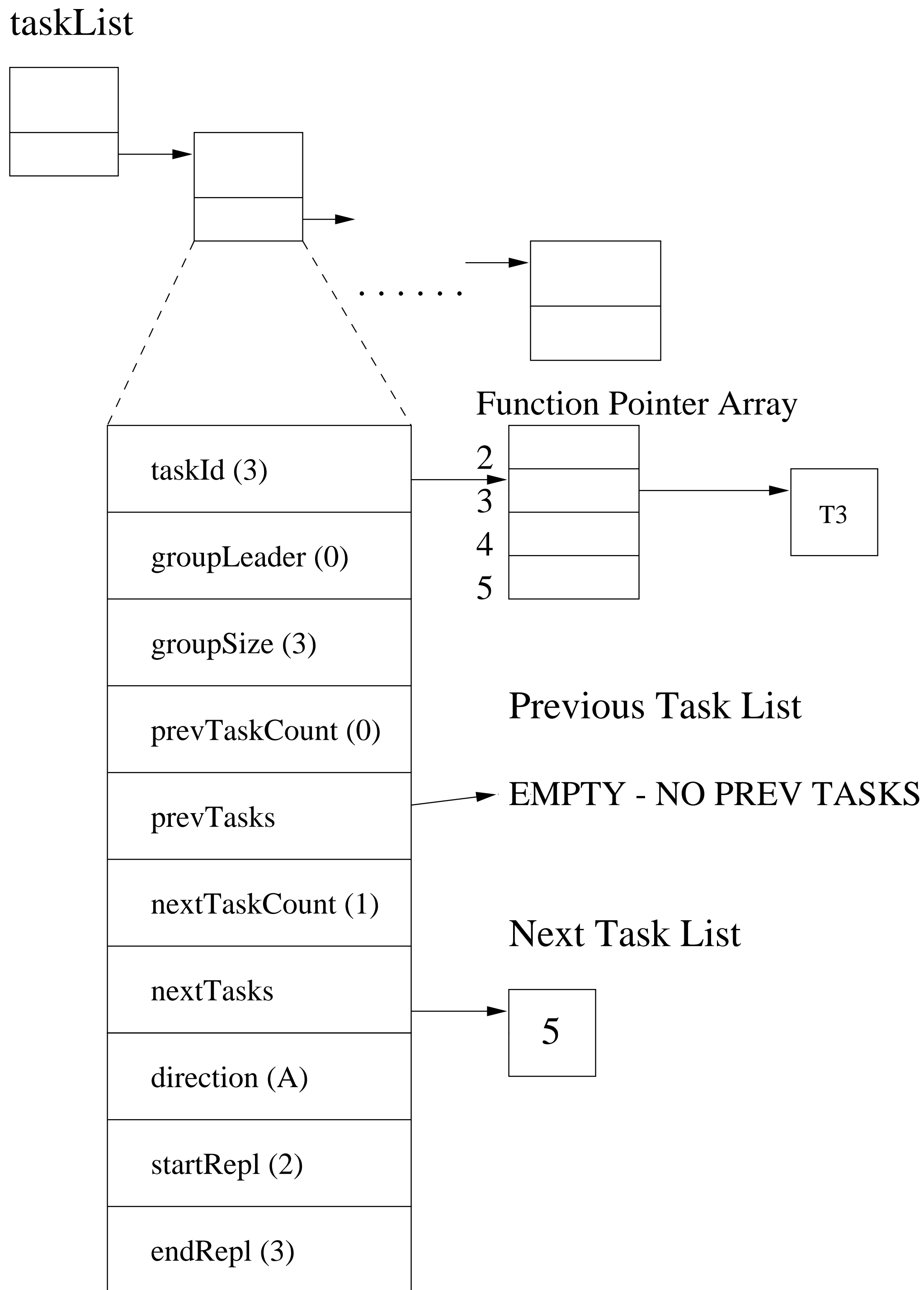


Figure 3.9: List of tasks on a processor. Modified task list structure for handling replicas. The values shown are for executing the T3" and T3"', which are instances of T3, on 3 processors.

The linked list specifies the order in which the tasks are executed. The emulator traverses this linked list, executing the tasks. If it encounters a task with a *direction* parameter of 'D', it executes all subsequent tasks until either an 'A' in encountered or the last replica is executed. The direction of 'A' signifies an execution of the instances in the list of replicas specified by the *startRepl* and *endRepl*.

For example, for the processor assignment in Table 3.4 for the task graph in Figure 3.6 and a blocking factor of 2, a compressed version of the linked list is given in Table 3.5. The first element in the linked list is task T3 and has the direction field 'A'. Hence, tasks T3 and T3' are executed first. When the emulator sees a task with the direction field of 'D' for an element, it executes the task instance with the replica number given by *startRepl*, and repeats this for successor elements in the list with the direction field of 'D'. When it finishes executing all instances with replica number of *startRepl*, it repeats the procedure for the remaining values of between *startRepl* and *endRepl*. In the list of Figure 3.5, the emulator executes tasks T4, T5, T4' and T5' in that order. The *startRepl* and *endRepl* values are set by the TAP based on the blocking factor, so the order of execution is strictly enforced.

Mode	TaskCode	NumProcs	ProcList	startRepl	endRepl
A	3	2	0 1	0	1
D	4	2	2 3	0	1
D	5	3	1 2 3	0	1

Table 3.5: Task list generated by TAP. Linked list specifies the order of execution of the tasks.

CHAPTER 4

PERFORMANCE

This section presents experimental results using TAP for finding effective processor assignments in parallel programs. In the first section we present the results of testing the performance of out-of-core programs, where the data used by the program does not fit in memory, with the task graph given in Figure 2.6. The next section discusses how TAP was used for studying the behavior of PHOENIX, a scientific parallel program.

The experimental environment used is a network of Sun Solaris Ultra-5 workstations connected by a 100Mbps LAN with 128MB RAM and 330MB swap space on each machine. However the actual amount of memory available to applications is less than 90MB, because the rest is used by the operating system. The only application that was running on the workstations during experiments was the program being tested.

4.1 OUT OF CORE TESTS

The amount of data each task works on influences the performance of the application. If a task becomes out of core, performance tends to degrade significantly. Allocating extra processors to data intensive tasks may result in better performance. In this section we will examine out-of-core task behavior through several experiments.

4.1.1 SINGLE TASK TEST

For this test we parallelized the data parallel code for task T3 of Figure 4.1 (which is the same as Figure 2.6 but reprinted here for convenience) given in Figure 4.2. The program

performs an array addition. This program was tested with different processor assignments while increasing the size of arrays used for each test. The table in Table 4.1 gives the performance results; the first column shows the result of running the program on five processors, and the second column for ten processors.

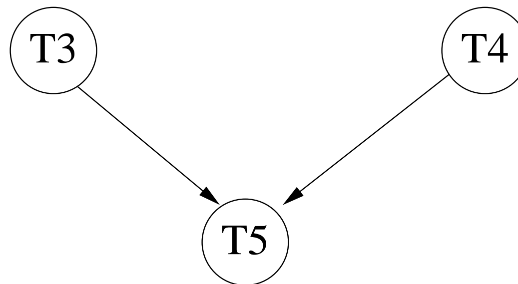


Figure 4.1: Task graph for testing the performance of a parallel program when it becomes out of core.

```
T()
{
 // Simple array addition
 for i = 0 to loopCount
    A = B + C + D + E;
}
```

Figure 4.2: Sample data parallel code used for testing out-of-core behavior.

	Num Procs		
	5	*10*	*Improvement*
Data Size(MB)	*Time(s)*	*Time(s)*	*%*
90	139	84	40
360	562	316	44
440	842	515	39
540	25809	1278	96

Table 4.1: Out Of Core test. Results shown are the execution time running the program on 5 processor and 10 processors. Data Size is the total memory used by all arrays in the program.

As can be seen from the results in Table 4.1, the improvement is about 40% for the first three tests. For the fourth test, the data will not fit in the aggregate memory of five

processors, resulting in poor performance. Doubling the number of processor gives a 20 fold improvement for this test, because the data fits in the aggregate memory of ten processors. Assigning extra processors to a data intensive task like T3 will likely result in better performance. For example, with the initial processor assignment in Table 4.2, if the task T3 becomes out of core, assigning four processors to T3 and just one processor to T4 will likely give better performance.

TaskCode	*NumProcs*	ProcList
3	3	0 1 2
4	2	3 4
5	3	1 2 3

Table 4.2: Sample processor Assignment for the task graph in Figure 4.1

4.1.2 Three task test

As we have mentioned in Section 1, data parallelism does not scale well beyond a certain point, and hence assigning more processors to a certain task may not always improve performance. Consider the task graph in Figure 4.1. Tasks T3, T4 and T5 are run with two different processor configurations shown in Table 4.3 and Table 4.4. Table 4.5 gives the results obtained from the tests.

Task	*Processr List*
T3	0, 1, 2, 3
T4	4, 5, 6, 7
T5	0. 1, 2, 3, 4, 5, 6, 7

Table 4.3: Task and Data parallelism. In this configuration the tasks T3, T4 executed by different set of processors and once those tasks are done, all processors are used for running T5.

The results show that using few processors at smaller data sizes is useful. The communication overhead for a parallel program increases as we increase the number of processors. This compensates for the gain in performance obtained by distributing computation

Task	*Processr List*
T3	0, 1, 2, 3, 4, 5, 6, 7
T4	0, 1, 2, 3, 4, 5, 6, 7
T5	0. 1, 2, 3, 4, 5, 6, 7

Table 4.4: Pure Data parallelism. In this tasks are executed one after the other; each task is executed on all processors.

Configuration	*Task and Data*	*Data*
Data Size(MB)	*Time(s)*	*Time(s)*
8	0.049	0.063
32	0.165	0.171
128	1.424	1.336

Table 4.5: Comparison of configurations in Table 4.3 and Table 4.4 . Data Size is the total memory used for arrays in tasks T3, T4, T5.

among more processors. Hence the processor assignment in Table 4.3 gives better performance, as four processors each are used for tasks T3 and T4.

4.1.3 Replicated tasks

Figure 4.3 gives an example of a task graph with sixteen replicas of tasks T1 and T2, each executing the code shown in Figure 4.4. The program was run on two processors with a blocking factor of 1. The tests were run with two different processor configurations as shown below.

1. *Data parallel* : Both the processors work on each task. This has the advantage of larger aggregate memory but often does not scale due to communication and synchronization overhead.

2. *Task parallel* : Instances of T1 and T2 are run on distinct processors. Each processor works on all the instances of a task and hence fewer processors are required.

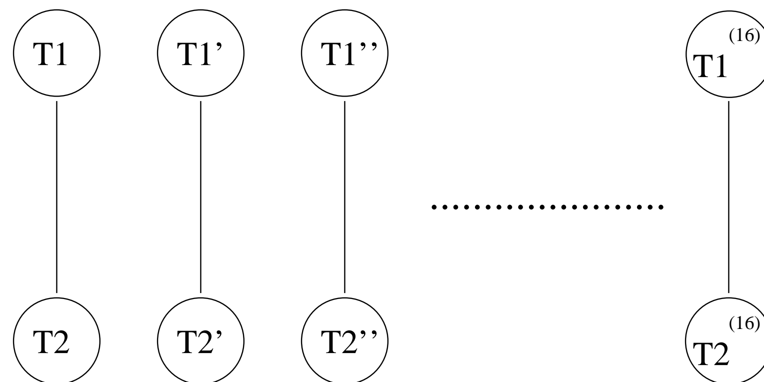


Figure 4.3: Replicated tasks graph. T1, T1' etc are instances of T1 and T2, T2' etc are instances of T2. Instances of a particular task work on the same data set.

```
T1()
{
 A = B + C;
}
T2()
{
 D = E + F;
}
```

Figure 4.4: Synthetic code executed by each replica in Figure 4.3. Each task does some simple array addition. All arrays are one dimensional.

Table 4.6 gives the results of the tests with and without synchronization in each task. Synchronization was added to the original code by adding barriers. Following observations can be made from the results in Table 4.6.

1. *No Synchronization* : When the data size is small there is not much difference in the performance of either approaches. However when the data size is increased running the tasks using task parallelism degrades the performance as the tasks become out

of core. In data parallel approach, both the processors work on a task and hence the aggregate memory fits the data resulting in better performance.

2. *Synchronization* : When the data size is small the performance of the data parallel approach degrades because of the synchronization overhead. The performance of a task parallel program does not get affected much because of lower synchronization overhead. However as the task becomes out of core data parallelism performs better.

	No Synchronization		*Synchronization*	
	Configuration			
	Data parallel	*Task parallel*	*Data parallel*	*Task parallel*
Array Size	*Time(s)*	*Time(s)*	*Time(s)*	*Time(s)*
2000	51.88	48.29	104.72	53.00
2200	59.94	58.44	123.46	63.71
2400	72.20	69.64	138.54	81.43
2600	84.79	87.80	156.62	93.96
2800	106.79	147.39	182.58	541.26
3000	125.89	2563.60	212.04	3234.57
3200	149.64	3546.32	271.34	4085.99
3400	171.11	4237.72	373.31	4983.93
3600	200.59	5412.43	544.74	6245.25

Table 4.6: Performance results for replicated task graph in Figure 4.3. Array size is the number of integers in each of arrays used for T1 and T2. The second and third columns give results of tests done by doing simple array addition with no synchronization; fourth and fifth columns give results with synchronization overhead.

4.2 PHOENIX

PHOENIX is a scientific application used for computing wavelengths of light emitted from stars. The program is written using FORTRAN and MPI [6]. The program has a task dependency graph of Figure 4.5 replicated more than thousand times, with no dependencies among replicas.

Characteristics of the implementation of PHOENIX are specified below.

1. Instances of tasks T1 and T2 read the wavelengths on disk in blocks of size 128KB. The data set read is the same for some instances of a task, and hence executing them serially exploits data locality. For example, the first ten replicas may work on blocks 1-16, then the next ten replicas work on blocks 17-32, and so on. Instances follow a *buffer-based* approach where a memory buffer is used to store the blocks read. The buffer size influences the total memory being used by the application. Increasing the buffer size improves the performance of instances of tasks T1 and

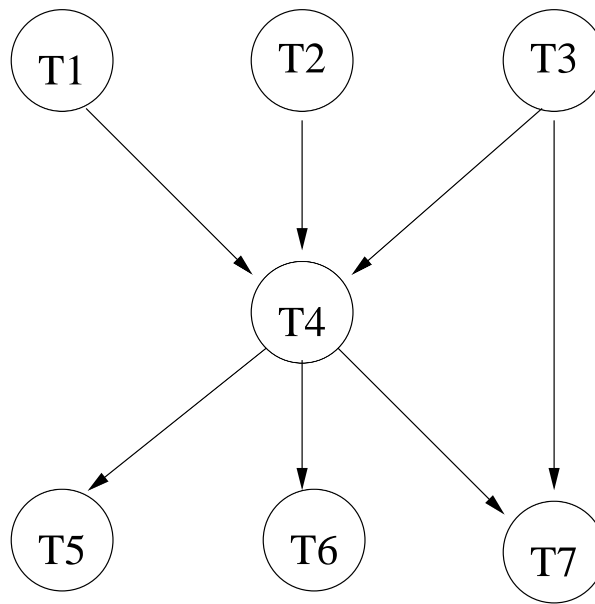


Figure 4.5: PHOENIX task graph

T2, but negatively affects the performance of other tasks. The total amount of data read from the disk by the whole application is in terabytes; about 20-100MB data is read by each replica.

2. Instances of T3 access about 1MB and produce about 20-100MB of data for instances of T7 in each replica. Hence, executing each instance of T7 after a instance of T3 exploits the data locality.

3. Instances of the tasks T1, T2, and T3 each produce 1KB of data for T4-instances. Hence, it is important to make sure that the data required by instances of T4 is not evicted from memory before the instances start execution.

4. Instances of tasks T5 and T6 do not significantly influence the performance of the application.

5. Within a replica, executing different tasks on different processors results in communication overhead because of the data distribution required between tasks in a replica. Hence, the replicas are divided among the processors and the tasks in a replica are run on a single processor.

As we can see from the task characteristics explained above, it is quite complex to determine the behavior of PHOENIX with different processor assignments. The emulator was used for testing a scaled down implementation of PHOENIX code.

4.2.1 HORIZONTAL EXECUTION

In the first set of tests the buffer-based approach was modified to a *horizontal execution* approach to minimize the size of the buffer, which results in better performance. In the buffer-based approach, blocks are read and stored in the buffer. If the buffer is full, a block is evicted by using a standard replacement strategy such as LRU. If the successor replica requires the evicted block it is re-read from the disk. Imagine a scenario where tasks T1 and T1' read 15 consecutive blocks and the buffer is big enough to hold 10 blocks. Sequential access of blocks may result in every block being read from the disk for tasks T1 and T1'.

In the horizontal-execution approach, a block will never have to be read more than once. We achieve this by performing the computation on all the replicas that require the block. This can be done in PHOENIX because within a replica, the wavelengths read are accessed in sequence and only once. Hence our approach is to read a block of data from disk, compute all replicas that work on that block, then read the next block, etc. This approach will decrease the size of buffer used for the tasks to a single block. Also the data

read is more likely to fit in the hardware memory cache of 512KB, speeding up memory access.

The buffer-based and the horizontal-execution approaches were compared by running 80 instances each of T1 and T2 on a single workstation. The code executed by the tasks was modified to do some simple array addition, which was adequate to represent the actual problem. The focus is to observe the performance with different approaches used for accessing blocks. The buffer-based approach was optimized so that each replica works on blocks that are present in the buffer before going to disk to fetch the required blocks. For example, if tasks T1 and T1' work on blocks 0-19 and the buffer can hold 10 blocks, at the completion of T1 blocks 10-19 will be in the buffer. Hence T1' works on those blocks before fetching blocks 0-9. Hence T1 has a *miss rate* of 1.0 and T1' 0.5 for buffer-based approach, where miss rate is defined as the percentage of failures to find the block in the buffer.

The block access patterns for instances of tasks T1 and T2 of each replica influence the performance of both the approaches. The different access patterns used for the performance analysis are given in Table 4.7; *t* denotes the replica number. The first two patterns have a miss rate of 1.0 for both approaches as the block needs to be fetched from the disk every time. In test case 4, the first two replicas work on blocks 0-19, the next two replicas work on blocks 10-29 and so on. For the buffer-based approach, test case 4 gives miss rates of 1.0, 0.5, 1.0, 0.5, etc for the instances of T1 and instances of T2, with a buffer to hold 10 blocks; test case 7 has a miss rate of 0.0 for all the instances except the first one, where all blocks need to be brought into the buffer. The miss rate for other patterns is high with buffer-based approach. Table 4.8 gives the results of both the approaches using a 10-block buffer.

As we can see from the results in Table 4.8, the horizontal execution approach gives better performance with most of the patterns. In the first two tests, it does not outperform block based approach because the miss rate is the same for both approaches. In the last

Test case	*Start block*	*End block*	*Example*
0	t	t	0, 1, 2, ...
1	(t-1)*10	t*10-1	0-9, 10-19, 20-29, ...
2	0	9+t	0-9, 0-10, 0-11, ...
3	0	t*10-1	0-9, 0-19, 0-29 ...
4	((t-1)/2)*10	((t+1)/2)*10-1	0-19, 0-19, 10-29, 10-29, 20-39, 20-39 ...
5	((t-1)/16)*80	((t+15)/16)*80-1	0-79, 0-79, *..16 times*, 80-159, 80-159 ..
6	0	160	0-160, 0-160, 0-160, ...
7	0	9	0-9, 0-9, 0-9, ...

Table 4.7: *Block access patterns.* Different patterns used for comparing the buffer based and horizontal execution approaches. t denotes the replica number.

test case horizontal execution was better because the blocks read fit in hardware memory cache.

The performance of instances of T1 and instances of T2 can be improved by increasing the size of the buffer. However, as previously said, this affects the performance of other tasks. This was tested by seeing the behavior of PHOENIX varying the buffer size. Consider the processor assignment in Table 4.9, for the task graph given in Figure 4.5, where tasks T1-T7 have task codes of 0-6. All the tasks are run on a single processor but the ordering of replicas is determined by the direction specified for each task.

The PHOENIX program was scaled-down as specified in Table 4.10 for running on a single workstation. The focus of this test is to study the behavior of the program with a particular disk and memory access patterns. Hence the PHOENIX code was mimicked by performing simple data manipulation. Three different block access patterns were tested. The patterns were (1) 0-31, 0-31, 0-31, etc, (2) 0-15, 16-31, 32-47 etc and (3) 0-15, 0-15, 16-31, 16-31 etc. Table 4.11 gives the results of increasing the memory cache for F1, F2.

Test case	*Buffer based*	*Horizontal execution*
	Time(s)	*Time(s)*
0	0.87	0.76
1	84.50	83.60
2	57.90	07.60
3	5221.00	107.00
4	58.50	43.70
5	94.60	47.20
6	138.00	12.30
7	0.51	0.25

Table 4.8: Results of comparison of buffer based and horizontal execution approaches.

Mode	*TaskCode*	*NumProcs*	*ProcList*
A	0	1	0
A	1	1	0
D	2	1	0
D	3	1	0
D	4	1	0
D	5	1	0
D	6	1	0

Table 4.9: Processor Assignment for observing the behavior of buffer based approach by increasing the buffer size for T1 and T2

The results show that the performance of the buffer based approach becomes worse by increasing the buffer size with patterns 2 and 3. This is because of the fact that increasing the buffer size decreases the amount of memory available to the other tasks. Because of this the data produced by instances of tasks T1 and T2 for instances of T4 is evicted from memory, the performance of instances of T4 suffers greatly as the data needs to be re-read from the disk. In the horizontal execution approach, we use a single block buffer.

Number Of Replicas	320
Data produced by each instance of T1 and T2	10KB
Data accessed by instances of T3	1MB
Data produced by T3	20MB

Table 4.10: Characteristics of simulated version of PHOENIX.

Pattern	*Buffer based*		*Horizontal execution*
	Time(s)		*Time(s)*
	10	40	1
1	89.70	37.72	12.43
2	50.40	179.40	47.99
3	43.66	95.2	33.22

Table 4.11: Performance when we increase the buffer size for T1 and T2. The results show the time taken for tests with 10-block and 40-block buffer for buffer based approach. We use a single block buffer for horizontal execution.

4.2.2 Blocking factor

In the previous section we saw that using horizontal execution can improve the performance when compared to the original approach. The performance results obtained in the previous sections were with a blocking factor equal to the number of replicas. Varying the blocking factor influences the performance of the parallel program. If instances of tasks T1 and T2, produce large amounts of data a higher blocking factor results in evicting the data produced by these tasks and will degrade the performance of instances of T4. A smaller blocking factor may not be able to take advantage of data locality for instances of T1 (as well as for instances of T2). The emulator can be used for finding a suitable blocking factor to give good performance.

Table 4.13 gives the results of tests done varying the blocking factor for the program with the characteristics given in Table 4.12 and using the horizontal execution approach.

We see from the results in Table 4.13 that there is not a significant difference in performance when the blocking factor is 16 and above. Better performance with a blocking factor of 16 when compared to blocking factor of 1024 is most likely due to hardware memory caching of data produced by the tasks. This behavior is influenced by the block access pattern that was chosen for tasks T1 and T2. With horizontal execution used for a block access pattern of 16 replicas of 0-15, 16 replicas of 16-31 etc, using any blocking factor which is a multiple of 16 exploits the data locality of the task where a block is read at most once. However when the blocking factor is smaller than 16, each block has to be read more than once; hence the performance becomes worse.

Number Of Replicas	1024
Data produced by each instance of T1 and T2	1KB
Block access pattern of T1 and T2	0-15[16], 16-31[16], ..
Data accessed by instances of T3	1MB
Data produced by T3	1MB

Table 4.12: In-core blocking factor test. Amount of data involved in the program used for obtaining a good blocking factor. The value in the brace in the example for block access pattern is the number of consecutive replicas which have the specific pattern specified.

BlockingFactor	*Time(s)*
1024	12.03
512	11.88
256	11.84
128	11.89
64	11.75
32	11.63
16	11.65
8	15.78
4	23.58
2	39.89
1	72.54

Table 4.13: Performance results obtained by varying the blocking factor for an in-core program with characteristics given in Table 4.12.

Larger blocking factors will affect the performance if the data produced by T1 and T2 gets evicted before T4 starts execution. Table 4.15 gives the execution time of instances

of tasks T1, T2 and T4 with different blocking factor values for the PHOENIX code with the characteristics given in Table 4.14 and the processor assignment in Table 4.9.

Number Of Replicas	512
Data produced by each instance of T1 and T2	100KB
Block access pattern of T1 and T2	0-31[32], 32-63[32], ..
Data accessed by instances of T3	1MB
Data produced by T3	10MB

Table 4.14: *Out-of-core blocking factor test.* Amount of data involved in out of core program used for testing the blocking factor.

BlockingFactor	*Time taken(s)*			
	T1	T2	T4	Total
512	211.45	222.88	112.66	556.00
256	223.20	221.35	27.31	496.06
128	223.84	221.31	8.83	475.16
64	224.79	220.87	5.19	461.41
32	224.42	221.64	5.35	459.08
16	**223.60**	**219.93**	**5.20**	**456.12**
8	225.76	222.02	5.19	460.50
4	233.50	228.17	5.20	474.27
2	240.44	233.70	5.19	486.78
1	253.45	250.40	5.20	516.57

Table 4.15: Performance when we vary the blocking factor for an out of core program with characteristics given in Table 4.14. *Total time* includes all tasks, though only 3 are shown.

The following observations can be made from the results shown in Table 4.15.

1. As we decrease the blocking factor, the time to execute instances of tasks T1 and T2 increases. This is because higher blocking factor values exploit data locality.

2. The performance of instances of T4 improves by decreasing the blocking factor. We can see that at the blocking factor of 512 the time taken to execute instances of T4 is large because the data produced by instances of tasks T1 and T2 for instances of T4 has been evicted before it started its execution.

3. The total time taken is influenced by the previous two factors. The blocking factor values of 16, 32, 64 give a good overall performance in this test case, with 16 the best (the results with blocking factor 16 are shown in boldface in Table 4.15).

4.3 Discussion

The results obtained using TAP on PHOENIX can be used for improving the performance. We saw that horizontal execution results in speed-up for the simulated PHOENIX program. We are expecting similar results in improvement after PHOENIX is modified for implementing the new approach. The results for the blocking factor tests yield further improvement in performance of the simulated program. We see that using a blocking factor value equal to the pattern size results in significant improvement in performance. If time is an important factor for doing simulations a blocking factor equal to pattern size is likely to give better results. The replicas can be divided among different processors based on the blocking factor. If there are enough processors, the replicas between two fences could be run on a single processor.

CHAPTER 5

RELATED WORK

In this section, we discuss the related work done in integrating task and data parallel programs, processor assignment, efficient I/O techniques for out-of-core programs, and data models for improving the performance of out-core programs.

INTEGRATED PARALLEL PROGRAMS

Considerable work has been done in exploiting the advantages of integrating task and data parallelism. For example, integrated parallelism was used in [17, 4] to obtain better speed-ups compared to either pure task parallelism or pure data parallelism [8, 14]. Gross et al. [7] developed a Fortran compiler that produces integrated task and data parallel code. The user needs to identify opportunities for task parallelism, and the compiler takes care of task creation and communication between tasks. The compiler identifies data parallel subroutines and they are used as units for task parallelism. Detailed analysis of the benefits of mixed data and task parallelism are provided by Chakrabarti et al. [4]. They developed a model to estimate the gains of mixed data and task parallelism and show that the integrated approach is better when either communication is slow or the number of processors is large. The language Braid [17] developed by West and Grimshaw integrates task and data parallelism. Braid programming model is a data parallel extension to the Mental Programming Language(MPL), which is an object-oriented task parallel language in C++. Bal and Hassen [8] present a programming model that integrates task and data parallelism using shared objects. The shared objects are used for communicating between

processors, storing shared data and for distribution of work in a data parallel way. Alok et. al. [1] developed a semantic model for communication between two data-parallel tasks. Bal and Haines [2] discuss different approaches as well as languages to integrate task and data parallelism. They discuss the approaches followed and the problems.

PROCESSOR ASSIGNMENT

Subhlok and Vondron [16] developed an algorithm for determining a processor assignment for a chain of tasks that optimizes latency in the presence of throughput constraints. The paper describes a general and realistic model of inter-task communication and addresses the entire problem of mapping, which includes clustering tasks into modules, assignment of processors to modules, and possible replication of modules. Andrei and Cristina [13] describe CPR (Critical Path Reduction), a compile-time heuristic for scheduling dependence graphs of coarse-grain data-parallel tasks. The paper how CPR can be used for scheduling tasks on processors for arbitrary topologies. Ramaswamy et al. [15] used convex programming to find the number of processors each task can be executed on and then scheduling them using a list-based algorithm.

I/O TECHNIQUES FOR OUT-OF-CORE PROGRAMS

Out of core parallel programs have been of interest in the parallel programming circles. Demke's [5] paper discusses a fully automatic technique to prefetch for out-of-core scientific applications. Compiler techniques are used to determine the future access patters for managing data reads. Li et. al. [10] present a framework for synthesizing I/O efficient out-of-core programs for block recursive algorithms, such as Fast Fourier Transform(FFT) and block matrix transposition algorithms. They capture the semantics of block-cyclic data distributions of out-of-core data and also data access patterns. They use that information for determining the data distribution for improving the performance. Jarek and

Foster [11] describe a programming model for a disk-based representation of an array and provide functions for transferring blocks of data between global arrays and disk resident arrays, allowing users to access data located on disk via a simple interface. However, the user has to take care of transferring sections of these disk arrays depending on the available main memory for computation.

DATA MODELS FOR OUT-OF-CORE PROGRAMS

Rajesh et. al. [3] present a data storage model that allows processors independent access to their own data and a corresponding compilation strategy that integrates data-parallel computation with data distribution for out-of-core problems. Kennedy et. al. [12] developed a compiler-based approach where the computations are delayed till the data required becomes available. The compiler techniques developed choreograph I/O of the application based on high-level programmer annotations similar to Fortran D's DECOMPOSITION, ALIGN and DISTRIBUTE statements. Kandemir et. al. [9] propose a data layout optimization for alleviating the problem of excessive I/O calls. The layout uses compiler analysis to obtain high level information about the data access pattern. This information is plugged into the run-time system.

TAP can be used as a test bed for the existing approaches to determine processor assignments. For example, compile time heuristics in CPR can be used for selecting the processor assignments to be tested with TAP. The mapping results obtained for processor assignments can be run with TAP to find estimates for synthetic parallel programs.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

Integrating task and data parallelism is important for practical applications and is often necessary for good performance. Determining processor assignments in these class of problems is complex, and for data intensive problems, executing many sample test runs may not be feasible. TAP is an emulator tool that can be used for running a portion of the parallel program or a scaled-down version of the program to determine good assignments. The paper presents how the emulator can be used for instrumenting parallel programs to run them with different processor assignments. With a good understanding of the emulator code, it can be used for determining the behavior of a parallel program. We showed how the performance of any application degrades if a particular task becomes out of core. Hence, allocating extra processors to data intensive tasks improves the performance. TAP was used to study the behavior of PHOENIX, and we observed that modifying the buffer based approach to horizontal execution approach gave big improvement in performance. The performance was further improved by choosing a proper blocking factor for the replicas in PHOENIX.

As of now, TAP can be used for running parallel program with regular task dependency graphs. In the future it should be possible to support arbitrary graphs for obtaining the behavior of any parallel program. Another future possibility include modifying TAP to support dynamic processor assignment. Finally, the approach currently followed requires user intervention for modifying TAP and the application before doing tests. In the future,

the emulator could be automated to make it easy for the user. It can be modified to instrument the application with the emulator code and running the application with different processor configurations.

Bibliography

[1] Bhaven Avalani, Alok Choudhary, Ian Foster, and Rakesh Krishnaiyer. Integrating task and data parallelism using parallel i/o techniques. Technical report, Syracuse University, Syracuse, NY, 1994.

[2] Henri E. Bal and Matthew Haines. Approaches for integrating task and data parallelism. Technical report, 1998.

[3] Rajesh Bordawekar, Alok Choudhary, Ken Kennedy, Charles Koebel, and Michael Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice or Parallel Programming*, pages 1–10, July 1995.

[4] S. Chakrabarti, J. Demmel, and K. Yelick. Modelling the benefits of mixed data and task parallelism. *SPAA'95*, 1995.

[5] Angela K. Demke. Automatic i/o prefetching for out-of-core applications. Technical report, University of Toronto, 1997.

[6] Ian Foster. In *Designing and building parallel programs*. Addision Wesley, 1995.

[7] T. Gross, D. O'Hallaron, and J. Subhlok. Task parallelism in a High Performance Fortran framework. *IEEE Parallel and Distributed Technology*, 2(3):16–26, 1994.

[8] S. Ben Hassen and H.E. Bal. Integrating Task and Data Parallelism Using Shared Objects. In *10th ACM International Conference on Supercomputing*, Philadelphia, PA, May 1996.

[9] M. Kandemir, A. Choudhary, and J. Ramanujam. Improving locality in out-of-core computations using data layout transformations. Technical report, Syracuse University, 1998.

[10] Zhiyong Li, John H. Reif, and Sandeep K.S. Gupta. Synthesizing efficient out-of-core programs for block recursive algorithms using block-cyclic data distributions. Technical report, Duke University, 1996.

[11] Jarek Nieplocha and Ian Foster. Disk resident arrays: An array-oriented i/o library for out-of-core computations. Technical report, 1996.

[12] Michael Paleczny, Ken Kennedy, and Charles Koebel. Compiler support for out-of-core arrays on data parallel machines. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 110–118, February 1995.

[13] Andrei Radulescu and Cristina Nicolescu. Cpr : Mixed task and data parallel scheduling for distributed systems. Technical report, Delft University of Technology, Delft, Netherlands, 2001.

[14] S. Ramaswamy, S. Sapatnekar, and P.Banerjee. A framework for exploiting task and data parallelism on distributed memory multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, pages 1098–1115, 1997.

[15] Shankar Ramaswamy, Sachin Sapatnekar, and Prithviraj Banerjee. A convex programming approach for exploiting data and functional parallelism on distributed memory multicomputers. In *Proceedings of the 23rd International Conference on Parallel Processing*, pages II:116–125, August 1994.

[16] Jaspal Subhlok and Gary Vondran. Optimal use of mixed task and data parallelism for pipelined computations. *Journal of Parallel and Distributed Computing*, 60(3):295–319, March 2000.

[17] Emily A. West and Andrew S. Grimshaw. Braid: Integrating task and data parallelism. In *Fifth Symposium on the Frontiers of Massively Parallel Computing*, February 1995.