

Exploiting Task-Level Parallelism Automatically Using pTask

by

Sum Huynh

A thesis submitted in conformity with the requirements
for the Degree of Master of Applied Science in the
Department of Electrical and Computer Engineering,
University of Toronto

© Copyright by Sum Huynh (1996)

Exploiting Task-Level Parallelism Automatically Using pTask

Sum Huynh

Degree of Master of Applied Science (1996)

Department of Electrical and Computer Engineering

University of Toronto

Abstract

Existing systems supporting task-level parallelism often involve the process of task synchronization and/or task creation. Task synchronization requires specification of dependencies or dataflow constraints among tasks, or data usage information of tasks. This thesis describes a system called pTask which automatically detects and exploits task-level parallelism in sequential array-based C programs. The system is composed of two components: a compile-time analysis module and a run-time system. The input to the compile-time module is a sequential C program in which the programmer annotates procedures to be asynchronously invoked as tasks. The output is a parallel program containing constructs for the creation and coordination of parallel tasks. pTask utilizes current compile technology to statically extract data usage information of tasks, and uses this information at run-time to dynamically detect and enforce data dependencies among tasks in order to exploit parallelism. A prototype of pTask has been implemented on a KSR1 multiprocessor. Experimental results show the system to be efficient and effective for many applications.

Acknowledgments

First of all, I would like to express my thankfulness to my supervisor, Professor Tarek Abdelrahman, for having introduced me to task-level parallelism and allowed me to explore and experiment; yet was always ready to offer guidance and advices. I learned so much from him.

To the people who helped this research work, I sincerely thank Dennis Gannon and associates from Indiana University for having created Sigma II; and John Ross from University of Toronto for taking care of the KSR machine and having responded (ever so friendly) to my seemingly absurd requests.

Financially, I am very grateful for the support from the Natural Sciences and Research Engineering Council of Canada.

Special thanks to CC, J Pang for their encouragement and support; to NVT for the great lodging, the fancy cuisine, and of course the charming company.

Finally, I would like to dedicate this thesis to my parents and siblings in appreciation for their patience throughout the last two years.

Table of Contents

1	Introduction	1
1.1	Motivation	1
1.2	Thesis Contribution	2
1.3	Project Overview	3
1.4	Thesis Organization	3
2	Background	5
2.1	Data Dependence	5
2.2	Loop-level Parallelism	7
2.3	Task-level Parallelism	9
2.4	Related Work	11
2.4.1	SCHEDULE	12
2.4.2	TDFL and LGDF2	12
2.4.3	COOL	13
2.4.4	Jade	15
2.4.5	Other Languages and Systems	16
2.5	Summary	18
3	pTask Overview	19
3.1	Design Considerations	19
3.1.1	Programming Paradigm	19
3.1.2	Task Definition	20
3.1.3	Task Communication	21
3.1.4	Task Synchronization	21
3.1.5	Hierarchical Concurrency	22

3.2	System Overview	23
3.2.1	Program Execution	24
3.2.2	Task Communication	24
3.2.3	Task Synchronization.	24
3.2.4	A pTask Program	25
3.2.5	System Components	26
3.3	Comparison to Related Systems	26
3.4	Limitations	27
3.5	Summary	27
4	The Compile-Time Module	29
4.1	The Basic Structure	29
4.2	Data Access Summaries	30
4.2.1	Data Access Regions	30
4.2.2	Regions of C Statements	32
4.2.3	Region Annotation.	35
4.3	Task-Procedure Conversions.	37
4.3.1	Handling Task-Procedure Parameters	37
4.3.2	Propagating Task Identifiers	38
4.3.3	Task-Procedure Invocation Conversion.	38
4.4	Synchronization Code Insertion	38
4.4.1	Task-Procedure Entry Synchronization	39
4.4.2	Procedure-Body Synchronization	42
4.4.3	Program Block Exit-Points Synchronization.	45
4.4.4	Program-Exit Synchronization	49
4.5	An Example	50
4.6	Summary	51
5	The Run-Time System	53
5.1	The Basic Structure	53
5.2	Task Synchronization	55
5.2.1	Region-lists	56
5.2.2	Establishing Serial Data Access Order	59
5.2.3	Enforcing The Synchronization Rule	61
5.3	Task Phases	61
5.3.1	Task Creation.	62
5.3.2	Task Execution	62
5.3.3	Task Resumption	64

5.4	Scheduling Tasks for Data Locality	64
5.4.1	Accessed Region-lists	65
5.4.2	Measuring Task-Worker Data Affinity	66
5.5	Summary	68
6	Experimental Results	69
6.1	The KSR1 Multiprocessor.	69
6.2	Performance Metrics	71
6.3	Matrix Multiplication	73
6.4	Search	74
6.5	Multidimensional Polynomial Interpolation (MPI)	75
6.6	Narrowband Tracking Radar.	77
6.7	Mergesort	81
6.8	Quicksort	85
6.9	TSP	88
6.10	Scheduling Tasks for Data Locality	95
6.11	Summary	96
7	Conclusions	99
7.1	Thesis Summary	99
7.2	Future Work	100
	Appendix A: pTask Manual	103
	Appendix B: Sigma II Limitations	107
	Bibliography	109

List of Figures

Figure 2-1:	An example of parallel loops	8
Figure 2-2:	Execution graph for program in Figure 2-1	9
Figure 2-3:	An example task graph	10
Figure 2-4:	A COOL matrix class	14
Figure 2-5:	A Jade matrix class	16
Figure 3-1:	A pTask program	25
Figure 3-2:	Compilation sequence in pTask.	26
Figure 4-1:	Data access regions.	31
Figure 4-2:	Region union and intersection	33
Figure 4-3:	An example of region annotation	36
Figure 4-4:	Task-procedure entry synchronization example	40
Figure 4-5:	Incremental task-procedure entry synchronization.	40
Figure 4-6:	Procedure-body synchronization example	42
Figure 4-7:	Sequential and asynchronous statements	43
Figure 4-8:	Procedure-body synchronization approaches for program in Figure 4-7	44
Figure 4-9:	Procedure-body synchronization for program in Figure 4-7	45
Figure 4-10:	Program block exit-points synchronization.	46
Figure 4-11:	Program block exit synchronization for program in Figure 4-10	49
Figure 4-12:	Example input program	51
Figure 4-13:	pTask-generated program for program in Figure 4-12	52
Figure 5-1:	The run-time system	54

Figure 5-2:	Serial execution order of tasks	56
Figure 5-3:	Region reservation for tasks in Figure 5-2	58
Figure 5-4:	Procedure-body synchronization	63
Figure 5-5:	Algorithm finding worker having highest data affinity for a region	67
Figure 6-1:	The KSR multiprocessor.	70
Figure 6-2:	Matrix multiplication	73
Figure 6-3:	Performance of matrix multiplication (512x512 matrices).	74
Figure 6-4:	Performance of Search	75
Figure 6-5:	Multidimensional polynomial interpolation	76
Figure 6-6:	Task graph for MPI.	76
Figure 6-7:	Performance of MPI (m = 128, n = 256)	77
Figure 6-8:	Narrowband tracking radar.	78
Figure 6-9:	Task graph for radar (first 2 iterations)	79
Figure 6-10:	Performance of narrowband tracking radar	80
Figure 6-11:	A sequential mergesort	81
Figure 6-12:	Task graph for mergesort	82
Figure 6-13:	The pTask mergesort	83
Figure 6-14:	Performance of mergesort (1M integers)	83
Figure 6-15:	A sequential quicksort	85
Figure 6-16:	Task graph for quicksort.	86
Figure 6-17:	The pTask quicksort	87
Figure 6-18:	Performance of quicksort (1 M integers)	87
Figure 6-19:	Path tree for 5 cities	89
Figure 6-20:	A sequential TSP	90
Figure 6-21:	Task graph for TSP for 5 cities	90
Figure 6-22:	The pTask TSP	91
Figure 6-24:	Performance of TSP (13 cities)	92
Figure 6-23:	Task graph for pTask TSP for 5 cities	93

List of Tables

Table 6-1:	Memory access times for KSR system	70
Table 6-2:	Improvements from schedule tasks for data locality	95
Table 6-3:	Summary of programming effort with pTask.	97

Chapter 1

Introduction

1.1 Motivation

Parallel programming is more difficult and error-prone than sequential programming. Much complexity results from the need to control and coordinate the interactions among concurrent processes. Efforts are required to ensure that parallel programs produce correct results. A working parallel program is not always a correct program! Furthermore, a correct parallel program does not necessarily result in good performance. To exhibit good performance, a parallel program must have low overhead, load balance and good data locality. In general, parallel programs are difficult to maintain and port to different platforms because they implement complex parallel algorithms, and contain platform-specific optimized code.

The unwelcomed challenges of parallel programming have fueled research in the area of restructuring and parallelizing compilers. These compilers automatically detect parallelism in sequential programs and restructure them into parallel programs. The majority of parallelizing compilers [6] [15] [21] have focused on parallelism within loops, where parallelism results from executing independent iterations of a loop in parallel. This

type of parallelism is commonly referred to as *loop-level* parallelism. Although these compilers eliminate the need for parallel programming and are generally effective, recent studies [10] [25] have shown that they have limitations, and that loop-level parallelism alone may not fully utilize the resources of parallel machines.

A more general source of parallelism is task-level parallelism in which a task is some unit of computation such as an arithmetic operation, a program block, or a procedure invocation. In this case, parallelism results from executing independent tasks in parallel. Some applications are more naturally expressed as a collection of related tasks [23] [25]. Furthermore, for large applications, it is necessary to exploit both loop-level and task-level parallelisms [25]. However, unlike parallelizing compilers, existing systems that support task-level parallelism demand programming effort, which ranges from having to manually create and synchronize tasks to having to program in different languages and paradigms. There is a need for systems that can automatically exploit task-level parallelism from sequential programs.

1.2 Thesis Contribution

This thesis develops a system called *pTask* which automatically detects and exploits task-level parallelism in sequential array-based C programs. The input to *pTask* is a sequential program in which the programmer intends procedures to be invoked asynchronously as tasks. The output of *pTask*, referred to as the *pTask-generated* program, is a parallel program that contains constructs for the creation and coordination of tasks. Explicit synchronization is inserted into the *pTask-generated* program because tasks communicate through procedure parameters and shared variables. The system automatically coordinates tasks to exploit parallelism while maintaining the program's sequential semantics. The use of *pTask* to parallelize a number of applications results in good performance improvements. The system is limited by available compiler analysis, by

dependences that exist in the sequential programs, and by the choice of procedure invocations as units of parallelism.

1.3 Project Overview

pTask consists of two main components: a compile-time analysis module and a run-time system. Given a sequential array-based C program with task declarations, the compile-time module determines the data accesses by tasks, converts procedure calls to task invocations, and inserts appropriate calls to the run-time system to synchronize tasks. The run-time system dynamically creates, schedules, executes, and synchronizes tasks. The system targets shared-memory multiprocessors. A (Kendall Square Research) KSR1 [17] serves as the development and testing platform. To ensure good performance, the run-time system schedules tasks for load balance and data locality. The system is used to parallelize a number of applications, and results are compared to those of manually parallelized programs.

1.4 Thesis Organization

The thesis is organized as follows. Chapter 2 provides a background on loop-level and task-level parallelism, and also describes related work. Chapter 3 outlines the design of pTask and gives an overview of the system. Chapters 4 and 5 detail the implementation of the system by respectively describing the compile-time module and the run-time system. Chapter 6 presents and analyzes the results of using pTask to parallelize a number of applications. Finally, Chapter 7 summarizes, gives conclusions and suggests some future work.

Chapter 2

Background

This chapter gives a background on data dependence, and on loop-level and task-level parallelism. The chapter also presents and discusses related work for task-level parallelism.

2.1 Data Dependence

This section briefly discusses the different types of data dependence. The subject is well covered in the literature [1][12][20] and hence, will only be reviewed here.

Data dependence relations are used to determine when two operations, statements, or iterations of a loop can be executed in parallel. There are four basic types of data dependence:

1. *Flow Dependence*: (also known as *true dependence*) exists between two statements if the data written by one statement is later read by the other statement.

For example, in the program segment

```
S1:   z = x + y
S2:   c = z * 2
```

there is a flow dependence between $S1$ and $S2$, denoted $S1 \delta^f S2$, or simply $S1 \delta S2$, because $S1$ writes variable z and $S2$ reads z .

2. *Anti Dependence*: exists between two statements if the data read by one statement is later written by the other statement. For example, in the program segment

```
S1:  z = x + y
S2:  x = c * 2
```

there is an anti dependence between $S1$ and $S2$, denoted $S1 \delta^a S2$, caused by variable x .

3. *Output Dependence*: exists between two statements if the data written by one statement is later written by the other statement. For example, in the program segment

```
S1:  z = x + y
S2:  z = c * 2
```

there is an output dependence between $S1$ and $S2$, denoted $S1 \delta^o S2$, because of variable z .

4. *Input Dependence*: exists between two statements if the data read by one statement is later read by the other statement. For example, in the program segment

```
S1:  z = x + y
S2:  c = x * 2
```

there is an input dependence between $S1$ and $S2$, denoted $S1 \delta^i S2$, caused by variable x .

Anti, output and input dependences can be eliminated by variable renaming [20]. The only real dependence is flow dependence.

In a loop, dependence can exist between instances of statements since two instances of statements can access the same element of an array variable. If the instances of the statements having dependences belong to the same loop iteration then the dependence is referred as a *loop-independent* dependence, and the instances can be executed concurrently without synchronization. For example, in the program segment

```
for i = 2 to N-1
S1:    a[i] = b[i] + c[i]
S2:    d[i] = a[i] + 1
endfor
```

the dependences among the instances of S1 and S2 are loop-independent; therefore, the iterations of this loop can be executed concurrently without synchronization. In such cases, the loop is referred as a *parallel loop*. On the other hand, if the instances belong to different loop iterations, then the dependence is referred as a *loop-carried* dependence, and the instances cannot be executed concurrently without synchronization. For example, in the program segment

```
for i = 2 to N-1
S1:    a[i] = b[i] + c[i]
S2:    d[i] = a[i-1] + 1
endfor
```

there is a loop-carried flow dependence between the instance of S1 in an iteration j and the instance of S2 in iteration $j + 1$ where $2 \leq j \leq N - 2$.

2.2 Loop-level Parallelism

The majority of parallelizing compilers exploit loop-level parallelism where parallelism results from executing independent iterations of a loop nest. Loop-level parallelism is also known as *data parallelism* because parallelism is achieved by having processors concurrently perform the same operation across a set of data.

```
S1:  ...
L1:  for i = 1 to N
      a[i] = i
    endfor
S2:  ...
L2:  for j = 1 to M
      b[j] = a[j]
    endfor
```

Figure 2-1: An example of parallel loops

Loop-level parallelism is typically implemented on shared memory systems using a fork-join model. Figure 2-1 shows a program segment with two parallel loops. The program would execute as follows. A single thread¹, the *main* or *master* thread, executes statement S1. Once loop L1 is reached, N-1 worker threads are created, and together with the main thread, each independently executes an iteration of loop L1. At the end of loop L1, the main thread waits until the worker threads complete their executions, before it proceeds to execute statement S2. Similarly, once loop L2 is reached, M-1 worker threads are created to execute the iterations of the loop in parallel. Figure 2-2 depicts the execution of this program with an execution graph where nodes represent the execution of the threads and edges represent dependencies among the threads of executions.

On most parallel processors, more than one iteration is typically assigned to a thread since one iteration per thread may lead to unacceptable overhead. The assignment of iterations to threads is known as *scheduling* the loop iterations. Scheduling may be done statically at compile-time or dynamically at run-time [16] [21].

Loop-level parallelism has been the main focus of parallelizing compiler research [6][15][21]; however, it is not without limitations [10]. First, a parallelizing compiler must be able to statically determine if a loop is parallelizable. In some cases, because of the way in which the loop is coded, the compiler must be conservative, i.e., assume dependence

1. We use the word thread generically, referring to a light-weight process that runs concurrently with other threads in a single address space.

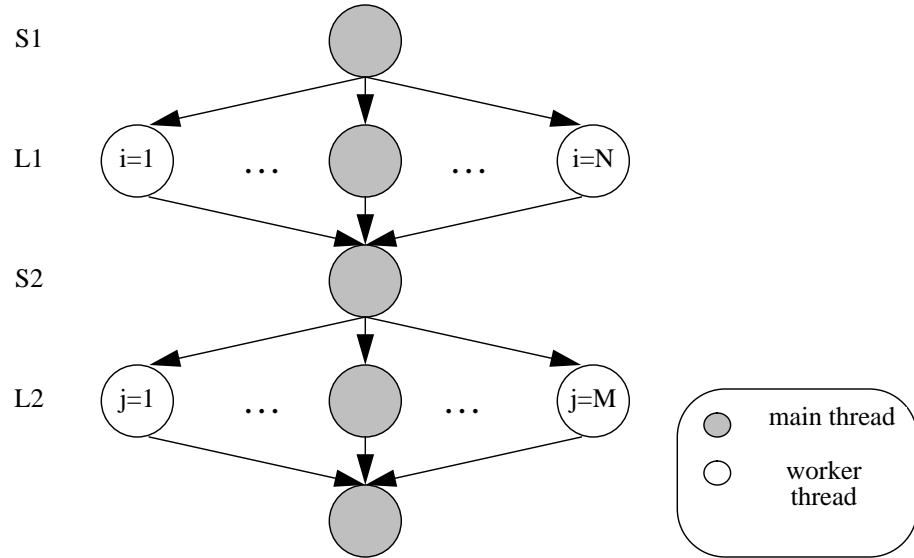


Figure 2-2: Execution graph for program in Figure 2-1

exists when unable to prove otherwise; even though such dependence may not exist. Second, the presence of procedure invocations in a loop often causes the analysis to be conservative. Thus, parallelism may not be fully exploited in modular programs. Third, data parallelism is useful when the size of the data can be scaled to fit the size of the parallel machine. A recent study [25] indicates that for some classes of applications, the physical constraints of the problem often make it impossible to arbitrarily increase the size of the data set, and data parallelism alone does not fully utilize the resources of parallel machines. Moreover, with data parallelism, communication and synchronization overhead grow with machine size; making it less profitable to apply more processors to a single data-parallel computation [4].

2.3 Task-level Parallelism

An alternate view of parallelism is among a collection of cooperating *tasks*. A task is a unit of computation which can be as fine-grain as an arithmetic operation or as coarse-grain as a procedure invocation that executes thousands of instructions. Parallelism stems from executing independent tasks concurrently. In contrast to data parallelism,

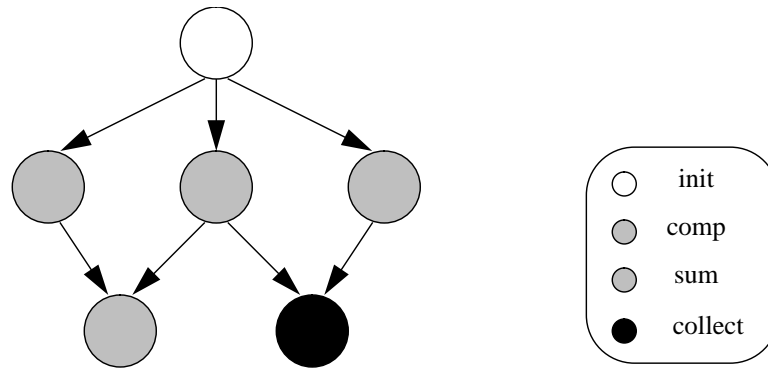


Figure 2-3: An example task graph

concurrently executing tasks are not limited to performing the same set of operations. This thesis is only concerned with coarse-grain tasks, i.e., tasks that execute at least thousands of machine instructions.

The notion of data dependence can be extended to tasks. In systems where tasks are synchronized according to the dataflow constraints among tasks, a task B is said to be dependent on a task A if task A produces some data value required by task B. Hence, task B cannot start execution until task A has produced the data. In systems where tasks are synchronized according a specific order such as the sequential data access order of tasks, then a task B is said to be dependent on a task A if task A precedes task B in execution, and either A or B writes to the same data. In this case, task B cannot start execution until task A has finished accessing the data. In both cases, we refer task A as the *prerequisite* task and task B as the *dependent* task. The dependencies among tasks in a program can be represented by a *task graph*, which is a directed acyclic graph whose nodes represent tasks and edges represent the dependences among tasks. A prerequisite task and a dependent task are at the source and the sink of a dependence edge, respectively. In general, a task can execute only after all its prerequisite tasks have executed.

As an example, Figure 2-3 illustrates the task graph of a program in which a task corresponds to a procedure invocation. The program would execute as follows. Task *init*

executes first since it has no prerequisite tasks. Once task `init` completes, the three `comp` tasks execute in parallel. Task `sum` starts as soon as the first two `comp` tasks complete, and the task `collect` executes as soon as the last two `comp` tasks complete. In this example, parallelism results from executing the three `comp` tasks in parallel, and executing tasks `sum` and `collect` concurrently.

Task-level parallelism is more general and more flexible than loop-level parallelism. For instance, tasks performing the same operations across a set of data can be executed concurrently, much in the same way as data parallelism. In addition, tasks performing different operations on different sets of data can also be executed concurrently.

However, the flexible nature of task-level parallelism leads to some disadvantages. Instead of building parallelizing tools focusing on a specific programming construct such as a loop, various languages and systems have been created to support different flavors of task-level parallelism. These systems vary according to their definitions of a task; to the time tasks are created; to the mechanisms used to support task communication and synchronization; as well as to the programming languages and paradigms used to express parallelism. Consequently, to exploit task-level parallelism, a programmer must first choose a system, and either extensively modify sequential programs or rewrite them in the programming language or paradigm required by the system.

2.4 Related Work

This section describes some systems that support task-level parallelism, focusing on the method of expressing parallelism and the programming effort. In particular, it describes SCHEDULE, TDFL, LGDF2, COOL, and Jade. Other systems such as Hypertool, PYRROS, Fortran M, and Fx are also discussed.

2.4.1 SCHEDULE

SCHEDULE [9] is a Fortran library package used to express and enforce inter-task dependencies in a parallel program. A task in SCHEDULE is a subroutine call. For each task, the programmer explicitly specifies its dependence relationships by supplying a unique identifier for the task, the number of prerequisite tasks, and the number of dependent tasks and their identifiers. In effect, the programmer manually generates task graphs.

Given a task graph, SCHEDULE run-time system executes tasks, obeying the specified dependence constraints. Tasks having no prerequisite tasks are executed first. Once a task finishes, the number of prerequisite tasks in each of its dependent tasks is decremented by one. A task is ready for execution when the number of its prerequisite tasks becomes zero. The process repeats until all tasks are executed.

SCHEDULE has little run-time overhead in the process of task synchronization since dependencies among the tasks are explicitly specified by the programmer. The system, however, has several disadvantages. First, to generate a task graph, the programmer must be able to describe the algorithm in a form of a graph, which is not always deducible from a sequential program or algorithm. Second, the programmer must manually specify the dependencies among tasks, which is an error-prone process, especially during the maintenance phase. Third, SCHEDULE's task graphs are static since the dependencies among tasks cannot be changed and new tasks cannot be added to the task graph once the program starts to execute. Consequently, the programmer must also know the structure of the computation a priori in order to build the task graph.

2.4.2 TDFL and LGDF2

TDFL (Task-Level Dataflow Language) [26] and LGDF2 (Large Grain Data Flow) [8] are parallel programming languages that allow programmers to express concurrency and synchronization with dataflow graphs in which nodes represent tasks and edges

represent the flow of data among the nodes. In these languages, a task is a procedure written in conventional language such as C, Fortran or Pascal. Given a dataflow graph, the system executes tasks according to the dataflow constraints. Tasks having no prerequisite tasks are executed first. A task is ready for execution as soon as all its data become available.

TDFL and LGDF2 have some advantages over SCHEDULE. First, they only require specification of dataflow constraints among tasks, rather than explicit inter-task dependencies. Second, they provide graphical interfaces to simplify the specification of dataflow graphs. Third, they provide programming constructs to support dynamic task creation. However, these systems still burden the programmer with the casting of programs and algorithms into dataflow graphs.

2.4.3 COOL

Concurrent Object-Oriented Language (COOL) [5] is derived from C++ by adding constructs to specify concurrency. A task in COOL is a procedure that is declared as *parallel*. When a parallel procedure is invoked, a task instance is dynamically created. COOL automatically executes tasks that operate on different objects in parallel. Parallelism can also be exploited within an object by declaring procedures (methods) in the class of the object as *parallel*. Hence, there can be concurrent tasks operating on different objects, as well as on a single object. To synchronize tasks within an object, the programmer must identify the procedures that require exclusive access to the object, and declare such procedures as *mutex*. A mutex procedure cannot execute until all other procedures executing on the same object finish; and no procedure can access the object until a mutex procedure finishes.

To further exploit concurrency within an object, the programmer can compose the object from smaller objects. For example, a matrix object can be composed of either row or column objects – depending how the matrix is accessed in the program. Figure 2-4

```
class row{
    int vals[MAXCOLS];
public:
    void read (); // non-exclusive method
    mutex write (); // exclusive method
};
class matrix{
    row rows[MAXROWS];
public:
    parallel read_row (int index) { // parallel method
        rows[index].read (); };
    parallel write_row (int index) { // parallel method
        rows[index].write ();};
};
```

Figure 2-4: A COOL matrix class

shows a COOL matrix class composed of row objects. Tasks operating on different matrices may proceed concurrently. Tasks accessing (reading or writing) different rows of a matrix may proceed concurrently, and likewise for tasks reading the same row. This type of concurrency, where parallelism is exploit at different levels of data accesses, is known as *hierarchical concurrency* [24].

COOL has a number of advantages over SCHEDULE, TDFL and LGDF2. First, it provides the programmer a much more familiar programming paradigm, despite that it is a concurrent language. In addition, it provides simple programming constructs such as `parallel` and `mutex` for the creation and synchronization of tasks. Furthermore, there is no need to express algorithms or programs as task graphs or dataflow graphs.

COOL, however, has two disadvantages with regard to *physically* composing objects from smaller objects in order to exploit hierarchical concurrency. First, access to a composed object can be restrictive. For example, the row-wise parallel matrix in Figure 2-4 cannot be concurrently accessed column-wise. Second, if the object is a vector of small base objects such as integers or characters, then it is more efficient to exploit concurrency

among tasks that access disjoint parts of the vector using COOL's low-level synchronization primitives, than to compose the vector object from these small base objects. This, however, demands similar effort to parallel programming using explicit synchronization primitives.

2.4.4 Jade

Jade [18][24] is a parallel programming language based on C++ that allows a programmer to express dynamic coarse-grain parallelism. A task in Jade is a program block that is annotated by the programmer with data usage information (*side-effects*) of that block. To declare a program block as a task, the programmer precedes the block with code that describes the data being accessed in the block; such code is referred to as side-effect specification. The programmer builds shared data objects using a system synchronization type called *token*, and then uses these tokens to specify the side-effects on the shared data. For example, if a program is to exploit task-level parallelism along the columns of a matrix, the programmer must build a shared matrix class which has a token for each column, and specify the side-effects on the columns using the corresponding tokens.

Figure 2-5 shows a sample Jade program segment in which tasks access a matrix column-wise. The program executes as follows. For each iteration of the `for` loop in procedure `update`, a task is created to write column `i` of the shared matrix `M`. By executing the code in the `withth` construct, the system obtains the side-effects of the tasks and dynamically synchronizes tasks based on their side-effects. In this example, tasks are executed in parallel since their side-effects are not in conflict with one another.

As a system supporting task-level parallelism, Jade possesses many desirable features. It provides the programmer with a familiar sequential programming platform. Similar to COOL, it provides simple means for the specification and synchronization tasks, and there is no need to express algorithms or programs as task graphs or dataflow

```

class SharedColumnMatrix // user defined class
{
    token _token; // synchronize among matrices
    token _column_token[N]; // synchronize within a matrix
    double elements[N][N];
public:
    void read_column (int i) { _column_token[i].rd() };
    void write_column (int i) { _column_token[i].wr() };
};

update (SharedColumnMatrix *M) {
    for (int i = 0; i < N; i++){
        // side-effect specification
        withth { M->write_column (i); }
        do (M, i) // task parameters
        { // code to write column }
    }
}

```

Figure 2-5: A Jade matrix class

graphs. Jade also aims to exploit concurrency among objects as well as within an object. However, unlike COOL in which the programmer must physically compose objects, a Jade programmer only has to use tokens to *logically* partition the objects. Consequently, access to a Jade object is not restricted as that to a COOL object. The disadvantage, however, is that the number of tokens may be large for large data. For example, to exploit parallelism among tasks accessing disjoint parts of a vector, a Jade programmer must associate a token with each element of the vector and specify the side-effects on every element being accessed; whereas in COOL such overhead can be avoided with the use of low-level synchronization primitives.

2.4.5 Other Languages and Systems

Hypertool [27] is a programming tool for scheduling and synchronizing a fixed number of tasks on hypercube machines. The input to Hypertool is a sequential program containing a series of procedure invocations, each of which represents a task. Data

dependencies among tasks are determined statically based on data flow analysis on the variables passed to the procedure invocations: if a variable is passed to two procedure invocations, a data dependence exists between the two corresponding tasks. The system schedules, maps, and synchronizes tasks based on results from the dependence analysis. In order to perform scheduling, the programmer must supply estimates of tasks' execution times. The dependence analysis in Hypertool is static and hence the task graph is static. The input program is restricted to procedure invocations. Nested procedures are not allowed.

PYRROS [28] is a parallel programming tool for scheduling static task graphs and generating code for message passing multiprocessors. PYRROS allows programmers to define task graphs in which a task is a program block. For each task, the programmer specifies task execution time, the communication (sending and receiving) between the task and other tasks, and communication costs. The system generates the output program and handles scheduling of tasks. A disadvantage with this model is that task execution times and communication costs are not usually available to the programmer.

Fortran M [4] is a language that comprises a small set of Fortran extensions. A Fortran M programmer uses these extensions to define tasks which are similar to procedures, to specify that a set of tasks is to be executed concurrently, and to define communication among tasks. Task synchronization is implicit since tasks communicate by sending and receiving input and output data, respectively, i.e., by message passing.

Fx [13] is a compiler that allows programmers to specify task-level parallelism in a data-parallel language based on High Performance Fortran. An Fx programmer has three responsibilities: (1) to define tasks in what is referred to as *parallel sections*, (2) to specify side-effects of tasks using an input/output directive, and (3) to map tasks onto processors using a mapping directive. The compiler maps tasks onto processors and generates communication to maintain data consistency. Fx restricts the code inside a parallel section to contain only loops and subroutine calls. Nesting of parallel sections is not allowed.

2.5 Summary

This chapter provided a background on loop-level and task-level parallelism. It also discussed some advantages of task-level parallelism over loop-level parallelism. However, with the existing systems, programming efforts are required to obtain task-level parallelism.

Chapter 3

pTask Overview

This chapter presents the design of pTask and an overview of the system from the user's perspective. The implementation details of the system are presented in the subsequent two chapters. The manual page for the system appears in Appendix A.

3.1 Design Considerations

A number of issues must be addressed in designing a system to support task-level parallelism. We identify the following issues: the *programming paradigm*, the *definition of a task*, the *mechanisms for inter-task communication and synchronization*, and the *exploitation of hierarchical concurrency*.

3.1.1 Programming Paradigm

Task-level parallelism can be expressed in many programming paradigms. Examples are task graphs (SCHEDULE), dataflow graphs (TDFL, LGDF2), concurrent objects (COOL), and sequential programming with language extensions (Jade). Graph-based paradigms are suitable for algorithms and applications that can be easily described by graphs, such as signal processing applications. However, a programmer is required to

cast algorithms into graphs, and is likely to be involved in task creation and synchronization – a complex and error-prone process. Concurrent languages burden the programmer with new programming languages and paradigms. We believe that sequential programming paradigm is more familiar and more natural to the programmer. Consequently, pTask is designed to support task-level parallelism within a sequential programming paradigm.

3.1.2 Task Definition

The issues concerning task definition are: what constitutes a task, and whether tasks are created statically or dynamically. To amortize the overhead required to exploit parallelism, most systems require tasks to be coarse-grain. Hence, in a sequential program, a task may correspond a program block or a procedure invocation. With pTask, we elect to define a task as a procedure invocation since this is likely to ensure coarse-grain tasks and this also promotes modular software design.

Tasks can be either static or dynamic. In systems supporting static tasks, the number of tasks does not change once the program executes. Static tasks are suitable for graph-based systems. The advantage with static task graph is that, with additional information such as the expected execution time of each task, the system can statically determine a scheduling policy that would result in good performance [19][28]. The disadvantage is that it restricts the kind of programs that can be represented to programs in which the number of tasks must be known in advance.

In systems supporting dynamic tasks, tasks are created dynamically as the program executes, and tasks can in turn create more tasks, referred to as *subtasks* or *child tasks*. Although these systems must dynamically schedule tasks in order to obtain good performance, and introduce additional run-time overhead in the process, they can better adjust to variable run-time conditions.

Since a task in $pTask$ corresponds to a procedure invocation and since a procedure may invoke other procedures, this implies a task may create subtasks during execution. Hence, tasks in $pTask$ are dynamic.

3.1.3 Task Communication

A system must also provide efficient task communication mechanisms. In systems targeting distributed-memory machines, tasks communicate by message passing. In systems targeting shared-memory machines, tasks communicate through shared variables. For $pTask$, a shared-memory model is assumed for parallel execution; therefore, tasks communicate by passing procedure parameters and by accessing (reading/writing) shared variables. The shared memory permits efficient sharing of data by passing addresses of data as parameters to tasks. Accesses to shared data by a task are referred as the *side-effects* of the task.

3.1.4 Task Synchronization

A system must synchronize tasks to enforce inter-task dependence constraints. Because tasks in $pTask$ are dynamic, the dependence relationships among tasks are also dynamic, and in general, not known until run-time. Hence, $pTask$ must be able to exploit concurrency that can be determined only as the program executes. We refer to such concurrency as *dynamic concurrency*.

There are various approaches which a system can use to determine dependencies among tasks. The first approach is to use compiler analysis [6][21] to statically determine the data dependencies among tasks, and let the system enforce the dependencies at run-time. This approach requires no effort from the programmer and is effective for loop-level parallelism; however, the presence of procedure invocations in modular programs often results conservative analysis [10]. In addition, such static dependence analysis fails to exploit dynamic concurrency.

The second approach is for the programmer to specify dependence (SCHEDULE) or dataflow constraints (LGDF2, [19][28]) among tasks, and let the system enforce these constraints at run-time. This approach allows the system to efficiently synchronize tasks since relationships among them are specified. The problem is that for each task, the programmer must consider all possible interactions between the task and all other tasks. As a result, programs using this approach can be difficult to code and maintain.

The third approach is for the programmer to specify side-effects of tasks, and let the system detect and enforce inter-task dependencies at run-time (COOL, Jade). Although, this approach incurs additional run-time overhead since for each task, the system must establish relationships between the task and other tasks, it reduces the burden on the programmer, who now only has to focus on the side-effects of each task instead of among the tasks.

pTask takes the last approach a step further by using available compiler technology to automatically obtain tasks' side-effects on the shared variables; and thereby effectively freeing the programmer from having to deal with task synchronization. In other words, *pTask* statically obtains tasks' side-effects and dynamically synchronizes tasks based on their side-effects in order to exploit dynamic concurrency. Chapter 4 discusses the side-effects in *pTask* in details.

3.1.5 Hierarchical Concurrency

In a system where concurrency is achieved by executing tasks accessing different data concurrently, one goal is to exploit parallelism at all levels of data access. For example, tasks accessing different matrices may be executed concurrently; similarly for tasks accessing different rows or columns of a matrix; and likewise for tasks accessing disjoint parts of a row or column. As discussed in Chapter 2, to exploit such nested parallelism or hierarchical concurrency, some systems require the data to be physically or logically structured in order to expose the different parts of the data that can be

concurrently accessed by tasks. For example, if tasks were to access rows of a matrix concurrently, a COOL matrix object must be composed of row objects and a Jade matrix must associate a token with each of the row. However, as discussed in Chapter 2, such structuring of the data can restrict parallelism or can be inefficient.

Because pTask synchronizes tasks based on their side-effects on the shared variables, it is possible for the system to decide if, for example, a task is accessing a matrix array, a row or a column of a matrix, or part of a row or column; and to exploit concurrency accordingly. Thus, by synchronizing tasks based on their actual side-effects, pTask does not require the programmer to structure the data for it to extract parallelism from all levels of data access.

3.2 System Overview

pTask is a compiler and a run-time system that automatically exploits task-level parallelism in sequential array-based C programs. To specify a procedure to be invoked asynchronously, the programmer annotates the procedure declaration as follows:

```
/*$ann [TASK()] */
void f1 (...) {
    ...
}
```

We refer such procedure as a *task-procedure*, and the program containing task-procedures as a pTask program. A task-procedure is called in the same way as a normal procedure, with the exception that its invocation results in a *task*. To keep the system simple, pTask requires task-procedures to be declared with `void` type. In general, the programmer should choose procedures with large-grain size to declare as task-procedures. We contend that little effort is required to select appropriate procedures.

3.2.1 Program Execution

pTask executes a C program as follows. The `main` routine, which is considered as the *main task*, starts executing serially. For each task-procedure invocation, instead of calling the procedure, the system creates a task, which is an independent thread of control that executes the body of the task-procedure asynchronously. A task may run on a separate processor, concurrently with the task that created it and with other tasks in the system. A task may in turn create child tasks by invoking task-procedures. In general, the program can be viewed as a set of tasks executing concurrently, with each task sequentially executing the body of the associated task-procedure, and creates more tasks whenever it invokes task-procedures. A task terminates when it reaches the end of the task-procedure. The program terminates when all tasks terminate.

3.2.2 Task Communication

Tasks in pTask communicate in two ways. First, the actual parameters of a task-procedure invocation become input to the new task, making it possible for a (parent) task to pass values to its child tasks. Second, tasks may communicate by accessing (reading/writing) data in the shared memory. In a C program, global variables are accessible by all procedures, and thus are shared by all tasks. In addition to global variables, tasks may also share automatic variables or dynamically allocated data since a procedure may pass references or pointers to these data to other procedures. In general, tasks share data if they access the same data at some address in shared memory.

3.2.3 Task Synchronization

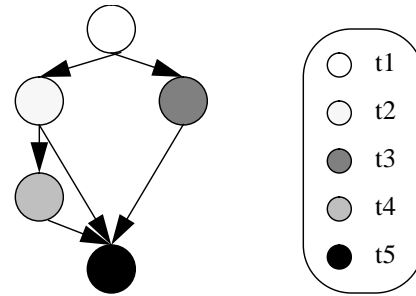
Synchronization of tasks in pTask is necessary to preserve program correctness. pTask preserves a program's sequential semantics by maintaining serial execution order for tasks performing conflicting operations on the same data. For example, tasks that write the same shared data must be executed sequentially in the program's serial execution order. The system also preserves serial execution order between a task that writes a data

```

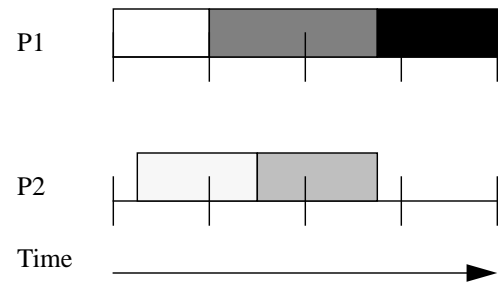
int total = 0;
void t1 () {
    int a[100];
    t2 (a); t3 (); t5 (a);
}
void t2 (int* p) {
    ... = total;
    t4 (p);
}
void t3 () {
    ... = total;
}
void t4 (int* p) {
    p[i] = ...; /*0<=i<=99*/
}
void t5 (int* p) {
    total = ...;
    p[i] = ...; /*0<=i<=99*/
}

```

(a) Program



(b) Task graph



(c) Program Execution

Figure 3-1: A pTask program

and another task that reads the same data. Of course, tasks accessing different data or reading the same data may be executed concurrently.

3.2.4 A pTask Program

Assuming all procedures in Figure 3-1 (a) are declared as task-procedures, the figure shows a pTask program in which tasks invoke subtasks, and tasks access both global and local automatic variables. Figure 3-1 (b) shows the program's task graph. The task graph indicates that tasks `t2` and `t3` may be executed concurrently since they read the same variable (`total`). Task `t4` is created within task `t2`, and may be executed concurrently with task `t2`. Task `t5` may only be executed after tasks `t2`, `t3` and `t4` terminate since it writes `total` which is read by tasks `t2` and `t3`. It also writes `a` which is

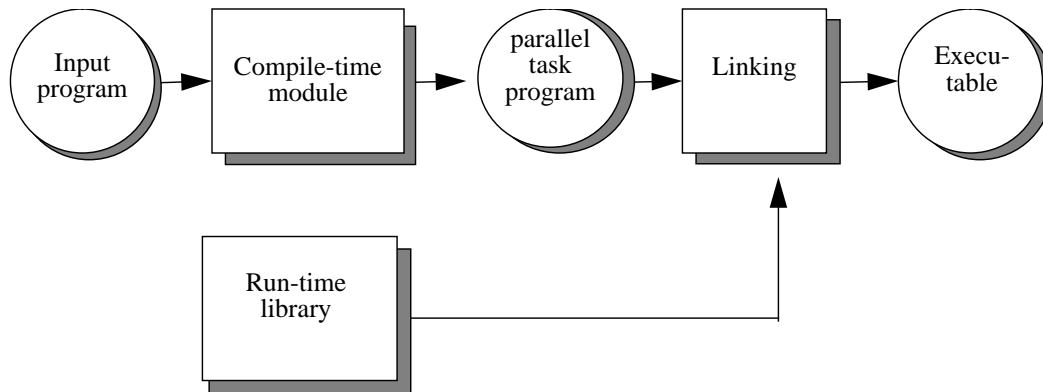


Figure 3-2: Compilation sequence in pTask.

written by task τ_4 . Figure 3-1 (c) depicts a possible program execution on two processors. In this example, the execution times of the tasks are arbitrarily chosen.

3.2.5 System Components

pTask is composed of two main components: a compile-time module and a run-time system. The compile-time module compiles the input sequential program into a parallel task program which contains calls to the run-time library to create and synchronize tasks. The parallel task program is then linked with the run-time library to produce the final executable. This sequence of compilation is shown in Figure 3-2. The run-time system carries out the instructions inserted by the compile-time module. In particular, it creates, schedules, executes, and synchronizes tasks. The compile-time module and the run-time system are described in details in Chapter 4 and 5, respectively.

3.3 Comparison to Related Systems

pTask is designed to automatically exploit task-level parallelism from sequential programs. The fact that task-level parallelism is extracted from sequential programs distinguishes pTask from systems such as SCHEDULE, TDFL, LGDF2 and COOL, in which parallelism is extracted from task graphs, dataflow graphs, or concurrent programs. In this respect, pTask is similar to Jade since both exploit task-level parallelism from sequential programs. However, in Jade, programmers specify side-effects in terms of

tokens, or logical representations of shared data, while in `pTask`, a compiler analysis is used to automatically extracted side-effects which are actual accesses to shared data. This automatic extraction of side-effects in `pTask` is limited (at present) to array-based applications. In Jade, programmers can always specify side-effects even for non array-based applications.

3.4 Limitations

`pTask` is not without limitations. First, the system depends on the ability of the compile-time analysis to obtain the side-effects of tasks. In the case where the analysis fails to obtain the side-effects, `pTask` allows the programmer to specify the side-effects, as will be described in Chapter 4. Second, since the performance of a `pTask`-generated program depends on the size of the tasks, `pTask` relies on the programmer to choose appropriate large-grain procedures to declare as task-procedures.

3.5 Summary

This chapter presented the design, an overview, as well as the limitations of `pTask`. The input to `pTask` is a sequential array-based C program in which the programmer annotates procedures to be invoked asynchronously as task-procedures. The system consists of two components: a compile-time module and a run-time system. The compile-time module compiles the input program into a parallel task program which contains constructs for task creation and synchronization. The run-time system dynamically creates, schedules, executes, and synchronizes tasks to maintain the sequential execution semantics and to exploit concurrency.

Chapter 4

The Compile-Time Module

This chapter describes the structure of the compile-time module and the steps it takes to compile a pTask program. The chapter is organized as follows. The basic structure of the module is presented in Section 4.1. Task side-effects and their representation are described in Section 4.2. Task creation and synchronization are detailed in Section 4.3 and 4.4, respectively. The chapter concludes with an example that illustrates the operations of the compile-time module.

4.1 The Basic Structure

The compile-time module compiles a pTask program into an equivalent parallel program which contains calls to the run-time library to create and synchronize tasks. The compile-time module is built around Sigma II [11], a software system for building source-to-source program restructurers and performance analysis tools. Sigma II is used in pTask for four purposes. First, it is used to parse the input program into an abstract syntax tree from which the compile-time module extracts information. Second, it is used to generate code for the output program. Third, it provides a facility for defining and parsing annotations using comments in the source code. We use this facility to allow the

annotations for task-procedures. Finally and most importantly, Sigma II facilitates the computing of tasks' side-effects by summarizing data accesses of procedure invocations, as well as of other statements in the program.

The compile-time module performs three main functions. First, it converts task-procedure invocations into task creations. This step enables the program to create tasks. Second, it obtains from Sigma II data access summaries for the statements in the program. Finally, it inserts code to build data structures to describe the data accesses and to direct the run-time system to synchronize tasks.

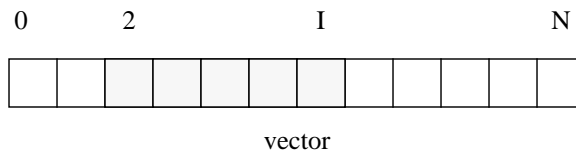
4.2 Data Access Summaries

The compile-time module uses Sigma II to obtain data access summaries for the statements in the input program. This section describes data access summaries and how they are represented by Sigma II.

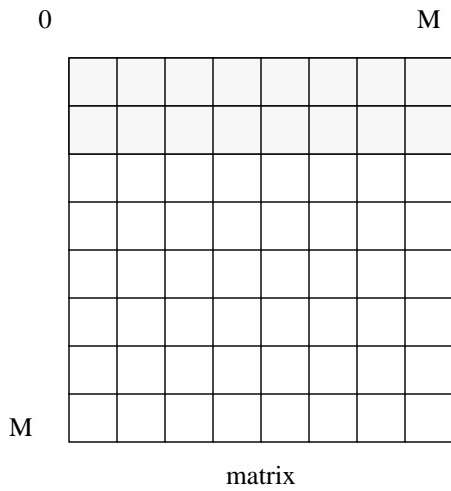
4.2.1 Data Access Regions

Sigma II summarizes the array accesses of a statement in the form of *access regions*. An access region describes the range of elements accessed in an array and the type of access, which is either read or write. A range of accesses is defined for each dimension of the array by the lowest and highest indices of the elements accessed in that dimension. Depending on how the array is accessed, these indices may be constants, variables, or expressions. The region of an array accessed is the cross product of the range in each dimension of the array. Hence, the region is the smallest rectangular space enclosing all elements accessed. Figure 4-1 shows sample elements accessed in a number of arrays and the corresponding regions.

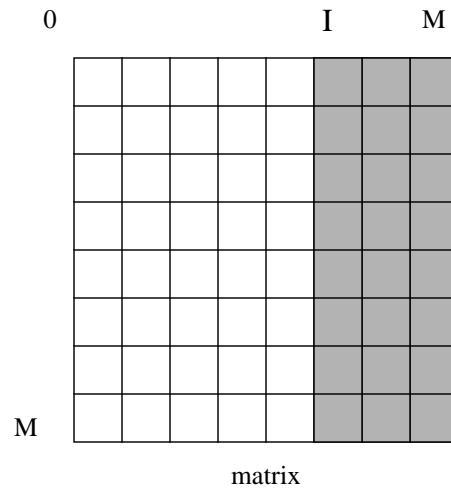
An advantage of representing data accesses using regions is that the notion of a region is simple, and there exist efficient algorithms for region *union* and *intersection*



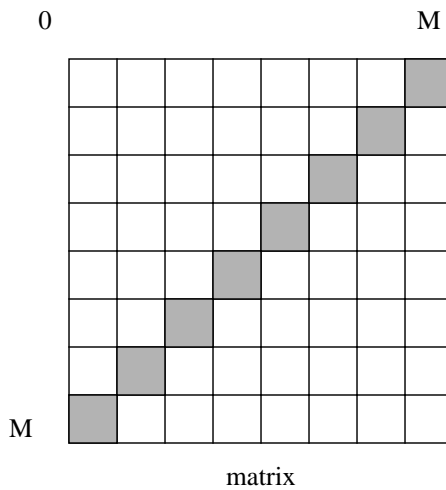
(a) Region = read vector [2:I]



(b) Region = read matrix[0:1][0:M]



(c) Region = write matrix[0:M][I:M]



(d) Region = write matrix[0:M][0:M]

Figure 4-1: Data access regions

operations [14]¹. The union of two or more regions is the smallest region that encloses these regions. The union of two regions is *accurate* if every array element of the resulting region is in either input regions. An accurate union is shown in Figure 4-2 (a). The union of two regions is *inaccurate* if the resulting region contains array elements that are in neither input regions. An inaccurate union is shown in Figure 4-2 (b). The intersection of two or more regions is the subregion in which these regions overlap. Unlike the union operation, region intersection is always accurate. Figure 4-2 (c) shows an example of region intersection.

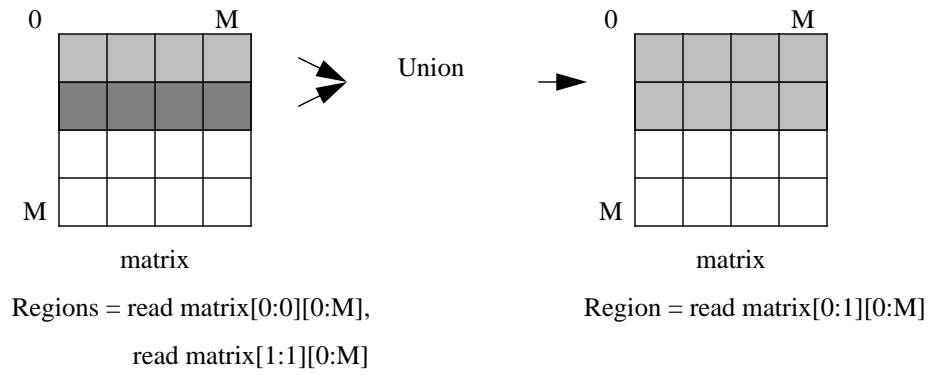
Region union is useful for computing data access summaries for a single statement or a sequence of statements in a program. With Sigma II, if the regions of a statement are adjacent to one another and are of the same access type, then they are unioned into a larger region; otherwise the statement has multiple regions. Region intersection is used to determine if two statements access the same data. Two regions are said to be in *conflict* if their intersection is not empty and one of them has a write access type.

There are, however, disadvantages in representing data accesses using regions. In general, regions cannot be used to describe accesses to dynamic data structures such as linked lists and trees. Furthermore, in Sigma II, the notation for regions is too simple to accurately represent complex data accesses. For instance, the diagonal access in Figure 4-1 (d) is represented by Sigma II as `matrix[0:M][0:M]`; a more accurate representation should be `matrix[i][i] (i=0:M)`.

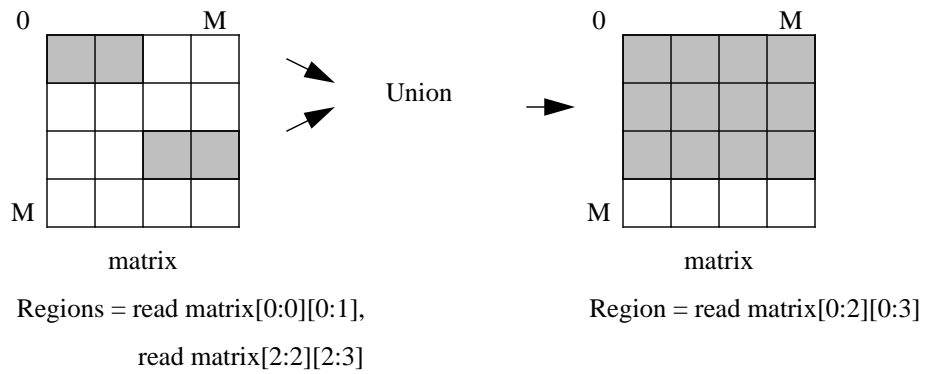
4.2.2 Regions of C Statements

This section defines the regions of various executable statements that commonly appear in C programs.

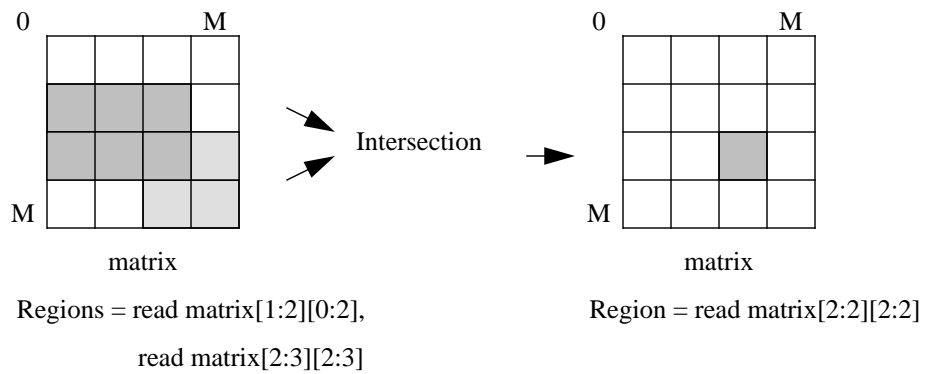
1. The description of these algorithms is beyond the scope of this thesis.



(a) Accurate region union



(b) Inaccurate region union



(c) Region intersection

Figure 4-2: Region union and intersection

Single Statements

The regions of a single statement, such as an assignment, are the data accessed by the statement. These regions are described in terms of the array subscripts in the statement. For example, the regions of the assignment statement `a[i] = b[j+i]` are `write a[i:i]` and `read b[j+i:j+i]`.

Conditional Statements

The regions of a conditional statement consist of the regions of the *condition* and the regions of the statements in the *if* and the *else* clauses. These regions are described in terms of the array subscripts in the statement. For example, the regions of the statement

```
if (a[i] > 0)
    a[i] = b[j+i];
else
    a[i] = c[j+i];
```

are `read a[i:i]`, `write a[i:i]`, `read b[j+i:j+i]`, and `read c[j+i:j+i]`.

Loops

The regions of a loop consist of the regions of the statements in the loop body for all loop iterations. These regions are described in terms of the loop induction variables, the loop bounds, and the array subscripts in the loop. For example, the regions of the loop

```
for (i = 0; i < m; i++) {
    if (a[i] > 0)
        a[i] = b[j+i];
    else
        a[i] = c[j+i];
}
```

are `read a[0:m-1]`, `write a[0:m-1]`, `read b[j:j+m-1]`, and `read c[j:j+m-1]`.

Procedure Invocations

The regions of a procedure invocation consist of the regions of all the statements in the called procedure. If the called procedure contains calls to other procedures then the regions of its invocation also include those of the other procedure calls. These regions are described in terms of the actual parameters of the procedure invocation, and/or the global variables accessed by the called procedure. The called procedure may access local variables; however, the regions of its invocations only involve the variables visible at the call sites. For example, in the following program segment

```

main () {
    ...
    f1 (array, size);
}
void f1 (int* p, int n) {
    int a[100];
    f2 (p, a, n);
}
void f2 (int* p, int* a, int n) {
    int i;
    for (i = 0; i < n; i++)
        a[i] = p[i];
}

```

the region of the call to procedure `f1` in `main` is `read array[0:size-1]`, and the regions of the call to procedure `f2` in procedure `f1` are `read p[0:n-1]` and `write a[0:n-1]`. The access to the local array `a` within procedure `f1` is not part of the regions of the invocations of `f1` in `main`.

Assuming the invocations of procedures `f1` and `f2` result in tasks, the above example shows that the regions of a parent task (invocation of procedure `f1` in `main`) include those of its subtasks (invocation of procedure `f2` within procedure `f1`), except for those regions accessed by the subtasks (task `f2`) which are of local variables (array `a`) within the body of the parent task (procedure `f1`).

4.2.3 Region Annotation

In addition to automatically obtaining data accesses, `pTask` also allows the programmer to manually specify regions of statements using annotations. Region

```

void f1 (int* p, int n) {
    a[100];
    /*$ann [REGION (r (p[0:n-1]), w (a[0:n-1]))] */
    f2 (p, a, n);
}
void f2 (int* p, int* a, int n){
    int i;
    for (i = 0; i < n; i++)
        a[i] = p[i];
}

main () {
    ...
    /*$ann [REGION (r (array[0:size-1]))] */
    f1 (array, size);
}

```

Figure 4-3: An example of region annotation

annotation is useful when the program has to call procedures of which the source codes are not available for analysis, or when Sigma II fails to provide data accesses due to, for example, recursive calls². To specify the regions of a statement, the programmer precedes the statement with a *region-annotation* comment which has the following syntax³:

```

region-annotation ::= /*$ann annotation-string */
annotation-string ::= [REGION (access [, access])]
access ::= access-type (access-var [expr:expr] [[expr:expr]])
access-type ::= w | r
access-var ::= C-variable
expr ::= C-expression

```

Where C-variable and C-expression are valid C variables and expressions, respectively. The regions of the procedure invocations in the previous example can be annotated as shown in Figure 4-3.

2. A list of Sigma II limitations appears in Appendix B.

3. Backus-Naur Form (BNF)

4.3 Task-Procedure Conversions

This section describes the second step of the compilation process in which the compile-time module converts task-procedure invocations into calls to the run-time system to create tasks. We refer these calls as *create-task* calls. This step involves handling task-procedure parameters, propagating task identifiers, and finally, converting task-procedure invocations into task creations.

4.3.1 Handling Task-Procedure Parameters

The run-time system executes tasks by invoking the appropriate task-procedures. One problem, however, is that task-procedures may have different number of parameters, which are of different data types. For the run-time system to invoke task-procedures correctly, it must keep track of the number and the types of the parameters of each task-procedure. This would complicate the implementation of the run-time system.

We choose to simplify the run-time system by having the compile-time module modify the program so that task-procedures accept a fixed number of known-type parameters. For each task-procedure, the compile-time module first creates a *task-context* which is a data structure containing all formal parameters of the task-procedure. Second, it precedes each task-procedure invocation with code to create a corresponding task-context and fill this task-context with the actual parameters of the invocation. Once the actual parameters are contained within a task-context, they can be conveniently passed to the create-task call in the form of the task-context. The compile-time module modifies each task-procedure header to accept its corresponding task-context as the only formal argument, and inserts code at the beginning of the task-procedure to extract the formal parameters from the task-context.

The above modifications simplify the run-time system because now the create-task call takes a task-context instead of actual parameters, and the run-time system can invoke task-procedures uniformly, passing task-contexts as formal parameters.

4.3.2 Propagating Task Identifiers

For the purpose of task synchronization, the run-time system must maintain parent-child relationships among tasks. When a task is created, the run-time system assigns it a unique identifier, referred to as the *task id*. When the task is executed, the run-time system passes this id to the task-procedure. When the task creates a child task, its id becomes the child task's parent task id. To propagate task ids, the compile-time module modifies headers of all task-procedures to accept the task id, in addition to the task-context, and it also passes task ids to create-task calls.

A task may create child tasks directly within the associated task-procedure or indirectly as a result of calling other procedures. In either case, its id must be passed to its child tasks. To support this, the compile-time module adds task id as an additional parameter to headers and invocations of *all* non task-procedures in the program. Hence, available within a procedure is the id of the task invoking the procedure.

4.3.3 Task-Procedure Invocation Conversion

Finally, the compile-time module replaces task-procedure invocations with calls to the run-time library to create tasks. Each create-task call contains all necessary information to execute and synchronize the new task. In particular, a create-task call takes as parameters the id of the task which creates the new task, the pointer to the task-procedure, the input to the new task in the form of a task-context, and the regions describing the data accessed by the new task.

4.4 Synchronization Code Insertion

A task can access a region only if the access does not violate the execution semantics of the sequential program. Hence, tasks in ρTask are synchronized according to the following rule: *a task can access a region if the region is not in conflict with regions of*

the tasks that execute and complete earlier (than the task) as procedures in the sequential program.

The run-time system provides a function called `region_check` which, given one or more regions of a task, determines whether the task can access the region(s), and if necessary causes the task to wait until it can access the region(s). The run-time system also provides a function called `region_signal` which, given one or more regions of a task, allows tasks waiting on the region(s) to proceed. The implementation of these functions is described in Chapter 5. Given these functions, the compile-time module obtains regions of all statements in the program, and inserts calls to these functions where synchronization is required. Basically, at each point where synchronization is required, a task would call a `region_check` on the region(s) before accessing the region(s), and calls a `region_signal` after having accessed the region(s).

In a program, synchronization is necessary at four specific locations: at the beginning of task-procedures, within the body of procedures⁴, at exit points of program blocks⁵, and at program exit points⁶. The following sections provide the motivations and the methods for synchronization at these locations.

4.4.1 Task-Procedure Entry Synchronization

A task-procedure invocation results in a task which is an independent thread of execution that executes the body of the task-procedure asynchronously. Synchronization is required at the beginning of the task-procedure to ensure the task accesses its regions only if such accesses do not violate the sequential execution semantics. For example, the program in Figure 4-4 creates a task t_1 , followed by task t_2 . Task t_2 must not proceed to execute the body of task-procedure t_2 until the data it needs is produced by task t_1 .

4. Task-procedures included.

5. In C, a program block begins and ends with `{` and `}`, respectively. Exit points here do not include the system call `exit ()`.

6. Calls to `exit ()`.

```

int sum;
main () {
    t1 ();
    t2 ();
}
/*$ann [TASK()]*
void t1 (int* b) {
    sum = ...;
}
/*$ann [TASK()]*
void t2 () {
    ... = sum;
}

```

Figure 4-4: Task-procedure entry synchronization example

```

/*$ann [TASK()]*
void t1 (int* a, int* b) {
    a[0] = ...;
    ... = b[0];
    b[1] = ...;
    ... = b[0];
    ...
}
void t1 (int* a, int* b) {
    r = all regions of a
    region_check (r);
    a[0] = ...;
    region_signal (r);

    r = all regions of b
    region_check (r);
    ... = b[0];
    b[1] = ...;
    ... = b[0];
    region_signal (r);
    ...
}

```

Figure 4-5: Incremental task-procedure entry synchronization

One way to achieve this synchronization is for the compile-time module to identify the first access to each variable within the task-procedure and precede each first access with a `region_check` call on those regions (of the task) associated with the variable. Similarly, the compile-time module also identifies the last access to each variable within the task-procedure and follows each last access with a `region_signal` call. Thus, a task may execute the body of the associated task-procedure as soon as it is created, but it must check for the regions of each variable before the first access to the variable. Figure 4-5 demonstrates this approach for a simple task-procedure. An advantage with this approach is that it enables the system to exploit more concurrency since regions associated with a

variable are available to waiting tasks as soon as the corresponding prerequisite tasks finish accessing the variable. A disadvantage is that the approach may cause a task to suspend many times during its execution since it may have to wait for regions to be signalled by prerequisite task(s). As will be discussed in the next chapter, task suspension incurs run-time overhead.

An alternate approach is to select and execute tasks that are eligible for execution, i.e., tasks that can access their regions without violating the sequential execution semantics. Given a task, the approach is to have the run-time system invoke `region_check` on the regions of the task to determine whether the task can access its regions. If the task can access its regions then its corresponding task-procedure is invoked; otherwise another task is considered. Hence, with this approach, the `region_check` operations are not inserted in the body of task-procedures, but are executed by the run-time system. One advantage with this approach is that it reduces the number of times a task may have to suspend. Another advantage is that depending on which of the processors the prerequisite tasks were executed, the run-time system may be able to schedule the task so that data locality is enhanced. Chapter 5 details such a scheduling strategy. The disadvantage with this approach is the loss of concurrency since tasks are not executed as soon as possible, and are not allowed to incrementally progress in execution. However, in applications where there are many tasks, it may be more efficient to selectively schedule and execute tasks than to execute and suspend tasks.

Depending on the application, one approach may result in better performance than the other. In the prototype, `region_check` operations are always inserted in the body of the task-procedures, and users may decide on the synchronization approach at run-time using an environment variable called `PTASK_CHECK_VAR`. If this environment variable is set then the first approach is used; otherwise the second approach is used.

```
/*$ann [TASK()]*/           /*$ann [TASK()]*/
void t1 (int* b) {          void t2 (int* b) {
    t2 (b);                b[0] = ...;
    b[0] = ...; /* statement s */ }
}
```

Figure 4-6: Procedure-body synchronization example

4.4.2 Procedure-Body Synchronization

Task-procedure entry synchronization ensures that a task starts to execute or proceeds in execution only if all existing dependence constraints are satisfied. However, an executing task may create child tasks and continue to access data that is also accessed by the child tasks, and hence additional synchronization is required to enforce these new dependences. For example, in the program shown in Figure 4-6, synchronization is required in task-procedure `t1` to ensure that statement `s` is executed only after its child task `t2` has completed; otherwise the dependence between task `t2` and statement `s` is violated. This type of synchronization is required in the body of a procedure that creates tasks, and hence is referred to as procedure-body synchronization.

Within a procedure, the compile-time module identifies two types of executable statements: *sequential* and *asynchronous*. A sequential statement is an executable statement that does not invoke tasks when executed. Examples of sequential statements are simple assignment statements, loops, and procedure calls that do not result in tasks. An asynchronous statement is an executable statement that may invoke tasks when executed. Examples of asynchronous statements are task-procedure invocations, and procedure calls that directly or indirectly invoke task-procedures. The compile-time module distinguishes task-procedure invocations as *direct asynchronous statements* and the rest of the asynchronous statements as *indirect asynchronous statements*. This distinction is necessary because, although both result in tasks, an indirect asynchronous statement may be a call to a procedure that contains both sequential and asynchronous statements. For example, in the program shown in Figure 4-7, the invocation of task-procedure `t2` in

```

/*$ann [TASK()]*
void t2 (int* a) {
    a[0] = ...;      /* sequential */
}
f1 (int* a) {
    t2 (a);          /* direct asynchronous */
}
/*$ann [TASK()]*
void t1 (int* a) {
    f1 (a);          /* indirect asynchronous */
    a[1] = ...;     /* sequential */
}

```

Figure 4-7: Sequential and asynchronous statements

procedure `f1` is a direct asynchronous statement, while the call to procedure `f1` in task-procedure `t1` is an indirect asynchronous statement because `f1` is a procedure that invokes task `t2`.

The compile-time module can provide procedure-body synchronization using either a *barrier-based* approach or a *region-based* approach. In the barrier-based approach, consecutive sequential statements in the body of a procedure are first grouped into blocks. Each *indirect* asynchronous statement is considered a block by itself because such a statement may be a call to a procedure that contains sequential statements. Next, each block is preceded with a *barrier* to cause the task executing the procedure to wait for all subtasks created from within the procedure to terminate. The task continues to execute the blocks only after its subtasks have terminated. Figure 4-8 (a) illustrates the barrier-based approach for the program in Figure 4-7. This approach is simple but it may introduce unnecessary waits since the regions of the subtasks and the regions of the blocks may not be in conflict. In the example, task `t1` can in fact execute the assignment statement concurrently with task `t2` since the region of the assignment statement and the region of task `t2` are not in conflict.

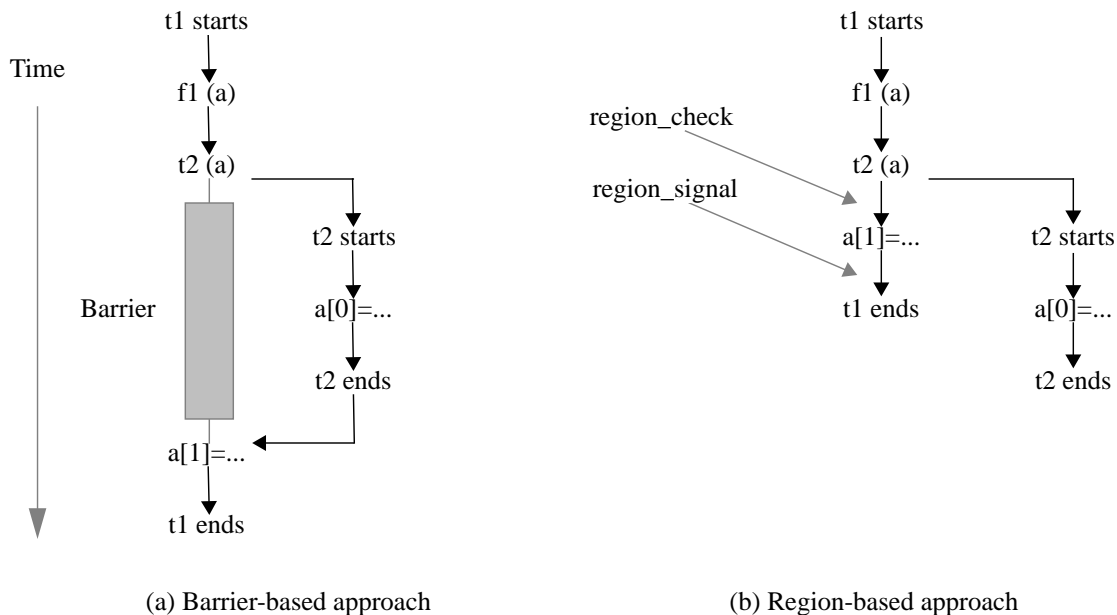


Figure 4-8: Procedure-body synchronization approaches for program in Figure 4-7

In the region-based approach, regions of each sequential and indirect asynchronous statements are obtained, and each statement is surrounded with a `region_check` and a `region_signal` call. Consequently, a task can execute a statement concurrently with its subtasks if the regions of the statement and the regions of the subtasks are not in conflict. Figure 4-8 (b) illustrates this approach for the tasks in Figure 4-7. The figure shows the assignment statement in task-procedure τ_1 being executed concurrently with task τ_2 since their regions are not in conflict. The compile-time module implements the region-based approach since this approach allows more concurrency. Figure 4-9 shows procedure-body synchronization for the program in Figure 4-7.

As a minor optimization, the compile-time module does not surround the statements that come before the first asynchronous statement with synchronization code since these statements are executed before the first subtask in the procedure is created.

A disadvantage with the region-based approach is that if the sequential statements are mostly single statements, rather than calls to large-grain procedures then the run-time

```

/*$ann [TASK()]*
void t2 (int* a) {
    a[0] = ...;          /* sequential */
}
f1 (int* a) {
    t2 (a);              /* direct asynchronous */
}
/*$ann [TASK()]*
void t1 (int* a) {
    f1 (a);              /* indirect asynchronous */
    region = write a[1:1];
    region_check (region);
    a[1] = ...;          /* sequential */
    region_signal (region);
}

```

Figure 4-9: Procedure-body synchronization for program in Figure 4-7

overhead from `region_check` and `region_signal` operations may outweigh gains from concurrency. A possible optimization is to group consecutive sequential statements into blocks, obtain and union the regions of the statements in each block, and surround each block with a `region_check` call and a `region_signal` call on the union regions. However, unless the union operation reduces the number of regions then the overhead would remain the same. On the other hand, if the number of regions is reduced then the result regions may be inaccurate, leading to false dependencies, and hence loss of concurrency. The benefits of such optimization are not clear and it is not pursued further.

4.4.3 Program Block Exit-Points Synchronization

Synchronization is required at exit points of a program block⁷ because a block may directly or indirectly create subtasks that access its array local (automatic) variables. For such blocks, the run-time system must ensure that the task executing the block leaves the block only after the subtasks have finished accessing the local variables; otherwise these variables would be de-allocated, causing the subtasks to access invalid data. For example,

7. In C, a program block begins and ends with { and }, respectively.

```

f0() {
    int a[N], b[N], c[N];
    t1 (a);
    f1 (b);
    f2 ();
    t2 (c);
}
/*$ann [TASK()]*
void t1 (int* p) {
    /* accessing p */
}

f1 (int* p) {
    t3 (p);
}
f2 () {
    int d[N];
    t4 (d);
}
/*$ann [TASK()]*
void t2 (int* p) {
    t5 (p);
}

```

Figure 4-10: Program block exit-points synchronization.

Task-procedures t_3 , t_4 and t_5 are identical to task-procedure t_1 .

synchronization is needed to ensure that the task executing procedure f_2 in Figure 4-10 returns only after task t_4 has terminated, because t_4 accesses the local array d .

One way to make certain that tasks always access valid data is to allocate memory space, copy local variables into the allocated space, and have tasks access the copies. Consequently, the executing thread can exit the block, leaving the subtasks with copies of the local variables. This approach, however, results in run-time overhead due to memory allocation and copying, and it consumes more memory. In addition, to de-allocate the memory properly, tasks must be coordinated to have the task that accesses a copy last de-allocate the space for that copy. Thus for each copy, the run-time system may have to maintain the number of tasks accessing the copy, and allow the last of such tasks to de-allocate the space.

An alternative is to provide synchronization to suspend the executing task until the subtasks that access its local variables have completed⁸. The insertion of synchronization code is complicated by the fact that a task may access variables that are automatically

8. In this system, the suspension of a task does not consume processor resource, i.e., it does not prevent another task from being executed on the same processor as that of the suspending task.

allocated elsewhere in the program. In general, a task may access data allocated in a program block of *any* ancestor task. For instance, in Figure 4-10, task t_4 accesses the data allocated in procedure f_2 of its parent, while task t_5 accesses the data allocated in procedure f_0 of its grandparent. The remainder of this section describes how `pTask` provides this type of synchronization.

Let us denote a program block that directly or indirectly creates tasks that access its local variables as a *local-access block*. Our solution is to precede exit points of local-access blocks with code causing execution to wait for all tasks created from within. For instance, in Figure 4-10, the task executing procedure f_0 must wait for tasks t_1 , t_2 and t_3 ; and the task executing procedure f_2 must wait for task t_4 . There is no need for the task executing procedure f_1 to wait for task t_3 since procedure f_1 is not a local-access block. Thus, the compile-time module must insert code to cause the task executing procedure f_0 to wait for task t_4 at the end of the local-access block in procedure f_2 and for tasks t_1 , t_2 and t_3 at the end of the local-access block in procedure f_0 .

In order for a task A to wait for a set of subtasks created from within a particular local-access block, it must know the number of subtasks created in that local-access block. Similarly, in order for these subtasks to signal task A to continue, each of the subtasks must know in which of task A 's local-access blocks it was created. This synchronization is facilitated as follows. Each task maintains an integer called the *block_number* which is initialized to zero at the beginning of the task-procedure. At the top of each local-access block, the compile-time module inserts code to increment the task's *block_number*. Before the exit points of each local-access block, code is inserted to decrement the task's *block_number*.

When a task is created, the current *block_number* of its parent is passed to it. Also the parent keeps count of the number of subtasks created at a particular *block_number*. Consequently, a task knows in which of its parent task's *block_number* it was created, and it also knows the number of subtasks it created in a particular *block_number*. To force a

task to wait for the subtasks created from within a local-access block, the compile-time module simply precedes the decrement of the `block_number` with code to cause the task to wait for the subtasks having the current `block_number`. In the example, tasks τ_1 , τ_2 and τ_3 are within a `block_number`, for example 1, while task τ_4 is in a higher `block_number`, for example 2. Hence, the task executing procedure f_0 can wait for task τ_4 at the end of the local-access block f_2 and for tasks τ_1 , τ_2 and τ_3 at the end of the local-access block f_0 .

The above scheme only ensures that tasks access valid data allocated in a program block of the parent task, but not in other ancestors. Indeed, in the example, the above scheme does not guarantee that task τ_5 will access valid data because procedure f_0 does not wait for task τ_5 . This raises an interesting question: how does procedure f_0 wait for task τ_2 's subtask τ_5 when task τ_2 does not wait for its own subtask? `pTask` handles this problem by treating a task-procedure as a local-access block and preceding the exit points of each task-procedure with code to cause the task to wait for all subtasks created in `block_number` zero. Hence, a task has to wait for its child tasks, but not other descendants. In the above example, procedure f_0 only has to wait for tasks τ_1 , τ_3 and τ_2 ; and task τ_2 has to wait for task τ_5 . It should be noted that for task τ_2 to wait for task τ_5 is unnecessary; however, it is acceptable because task τ_2 has completed its execution; it has signaled all its regions allowing tasks waiting on the regions to continue; and its suspension does not consume a processor resource.

In short, to honor the scope rule, the compile-time module inserts code to record in which programming block of a task its subtasks were created. It then precedes the exit points of each local-access block with code to cause the task executing the block to wait for the subtasks created within the same block. Figure 4-11 highlights the necessary synchronization for the program in Figure 4-10.

```

f0(void* _id){
    int a[N], b[N], c[N];
    block_number_increment (_id);
    create_task (_id, t1,...);
    f1 (_id, b);
    f2 (_id, b);
    create_task (_id, t2,...);
    wait for subtasks created
        in current block_number;
    block_number_decrement (_id);
}

/*$ann [TASK()]*/*
void t1 (void* _id, context) {
    extract p from context;
    /* accessing p */
}

f1 (void* _id, int* p){
    create_task (_id, t3, ...);
}
f2 (void* _id, int* p){
    int d[N];
    block_number_increment (_id);
    create_task (_id, t4,...);
    wait for subtasks created
        in current block_number;
    block_number_decrement (_id);
}

/*$ann [TASK()]*/*
void t2 (void* _id, context) {
    extract p from context;
    block_number_init (_id);
    create_task (_id, t5,...);
    r = all regions of p;
    region_signal (r);
    wait for subtasks created
        in current block_number;
}

```

Figure 4-11: Program block exit synchronization for program in Figure 4-10

4.4.4 Program-Exit Synchronization

A sequential program terminates upon invoking an `exit`. In a `pTask`-generated program, since any task may invoke `exit`, synchronization is required to ensure proper program termination. Regardless which of the tasks invokes `exit`, the parallel task program must terminate. On some parallel systems, if the main thread⁹ invokes `exit` then all threads in the program terminate, but if any other thread invokes `exit` then only that thread terminates. In `pTask`, the task that invokes `exit` (referred as the *exit-task*) must terminate, *and in addition*, states of all other tasks, such as its subtasks, its dependent tasks, and tasks already running in the system, must be determined.

9. The main thread is the one that creates all other threads in a parallel program.

`pTask` handles program-exit synchronization by having the compile-time module replace every `exit` call by a `sys_exit` call to the run-time system. The procedure `sys_exit`:

- Causes the calling task (i.e., the exit-task) to wait for its subtasks. This is necessary because in the sequential execution, these subtasks would have been executed before `exit` is invoked.
- Calls `exit` to terminate the program if the calling task is the main task. Since every task waits for its subtasks and the main task is the ancestor of all tasks, this implies the program terminates when all tasks have terminated. In other words, this results normal program termination.
- Signals the main task to indicate that an `exit` has been invoked, if the calling task is not the main task. The calling task then waits for the program to terminate. Upon receiving the signal, the main task executes a signal handling routine which calls `exit` to terminate the program. Unlike above where the program terminates with all tasks terminated, some running tasks may be prematurely terminated and hence the program may not terminate normally. Such behavior is valid because the prematurely terminated tasks must had been accessing different data from the one was being accessed by the exit-task; whether they get to run to completion or not does not change the intended sequential behavior of the program.

4.5 An Example

This section illustrates the functions of the compile-time module with an example. The input program, shown in Figure 4-12, declares a procedure called `order_key` and a task-procedure called `order`. Procedure `order_key` calls `order` to merge subarrays of array `key`, using array `w` as a temporary buffer. After merging, array `w` is copied back to array `key`. Figure 4-13 shows the output from the compile-time module. In the output

```

order_key (int* key, int n, int k){
    int j, k, m, w[MAXSIZE];
    for (j = 0; j < n-k; j+= 2*k)
        order (key, w, j, j+k, k);
    for (j = 0; j < n; j++)
        key[j] = w[j];
}
/*$ann [TASK()] */
void order (int* a, int* b, int loff, int hoff, int m){
    int i, j, k;
    for (i = j = k = 0; i < m && j < m; k++)
        if (a[i + loff] < a[j + hoff])
            b[k + loff] = a [i + loff], i++;
        else
            b[k + loff] = a[j + hoff], j++;
}

```

Figure 4-12: Example input program

code, task-procedure `order` contains only task-procedure entry synchronization because it consists of solely sequential statements. Procedure `order_key` has other synchronization code because it has data (array `key` and array `w`) accessed by both asynchronous and sequential statements (the first and second loop, respectively), and because it has subtasks accessing a local array variable (array `w`).

4.6 Summary

The compile-time module compiles a `pTask` program into an equivalent parallel task program which contains calls to the run-time library to create and synchronize tasks. Beside having to obtain regions of statements in the program, and to convert task-procedure declarations and invocations, the compile-time module must also insert code to synchronize tasks. Much effort is required to provide proper synchronization in order to preserve the semantics of the sequential program. In addition to automatically obtaining data access regions, the compile-time module also allows programmers to specify regions using region annotations.

```

order_key (void* _id, int* key, int n, int k){
    declarations of context, region1, region2;
    int j, k, m, w[MAXSIZE];
    block_number_increment (_id);
    for (j = 0; j < n-k; j+= 2*k){
        /* order (key, w, j, j+k, k); */
        context = key, w, j, j+k, k;
        region1 = read key [j:j+k-1] and read key[j+k:j+k+k-1]
            and write w[j:j+k+k-1];
        create_task (_id, order, context, region1);
    }
    region2 = read w[0:n-1] and write key[0:n-1];
    region_check (region2);
    for (j = 0; j < n; j++)
        key[j] = w[j];
    region_signal (region2);
    wait for subtasks created in current block_number;
    block_number_decrement (_id);
}

/*$ann [TASK()] */
void order (void* _id, order_context* _context){
    declaration of r;
    int i, j, k;
    extract parameters from _context;
    r = all regions of a; region_check (r);
    r = all regions of b; region_check (r);

    for (i = j = k = 0; i < m && j < m; k++)
        if (a[i + loff] < a[j + hoff])
            b[k + loff] = a[i + loff], i++;
        else
            b[k + loff] = a[j + hoff], j++;

    r = all regions of a; region_signal (r);
    r = all regions of b; region_signal (r);
}

```

Figure 4-13: pTask-generated program for program in Figure 4-12

Chapter 5

The Run-Time System

This chapter presents the design and the implementation of the run-time system. The chapter is organized as follows. Section 5.1 gives an overview of the system by describing its main components, and how tasks are queued and executed. Section 5.2 discusses the mechanisms used to preserve the sequential execution semantics and to exploit concurrency. Section 5.3 details task creation, execution and resumption. Finally, Section 5.4 discusses task scheduling strategies.

5.1 The Basic Structure

The run-time system provides an environment to create, execute and synchronize tasks. It is initialized and taken down by calls inserted by the compile-time module at the entry and exit points of the main procedure, respectively. Hence, the run-time system is started as soon as the parallel task program starts and is taken down when the program terminates. The system consists of a global task queue, and a set of p workers where p is the number of available processors. The global task queue and the workers are created and freed during the system initialization and take-down process, respectively.

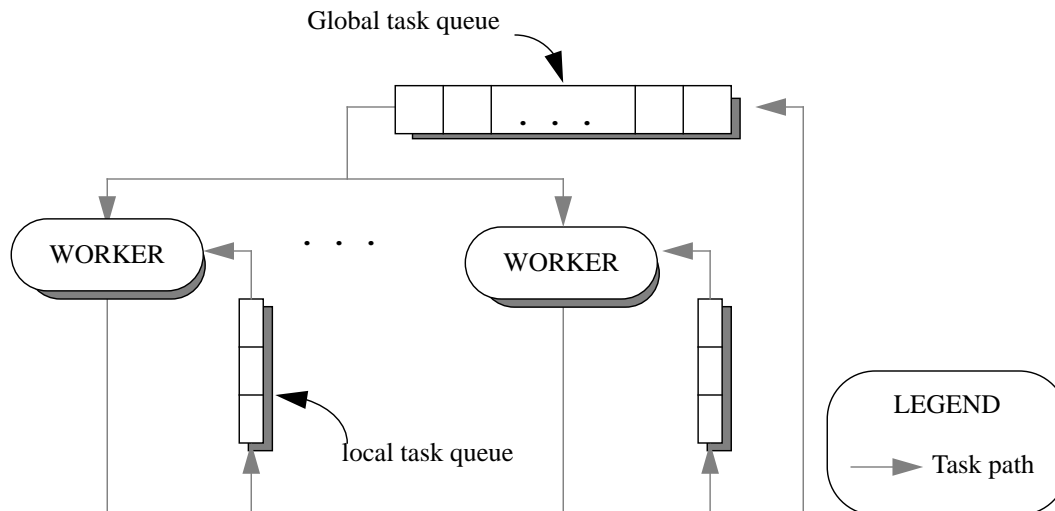


Figure 5-1: The run-time system

An invocation of `create_task` in the parallel task program creates a data structure containing a unique id for the task, the id of the parent task, a pointer to the associated task-procedure, the task-context, and the regions of the task-procedure invocation. An instance of this data structure is herein referred to as a task. Although in subsequent discussions, tasks are said to be passed from one component of the run-time system to another, in the actual implementation, only pointers to the tasks are passed. We explicitly refer to task pointers whenever clarification seems necessary. Once the task is created, it is added to the global task queue. Workers remove tasks from the global task queue and execute tasks.

A worker is a thread running on a separate processor. Each worker also has a local task queue which is read by the worker itself and can be written to by other workers. If a worker receives a task that must be executed by another worker, or that can be more efficiently executed by another worker then it adds the task directly to the other worker's local task queue. As will be explained later in this chapter, a suspending task must be resumed on the same worker that was executing it, and it may be more efficient to execute a task on the worker that most recently accessed the regions of the task. Figure 5-1 illustrates the run-time system.

Once a worker is started, it flags that it is idle, and then continuously removes tasks from the global and the local task queues. Higher priority is given to the local task queue since tasks on this queue can only be executed by the worker that owns the queue. Given a task, depending on whether task-procedure entry synchronization is performed by the run-time system or by the `region_check` operations within the task, a worker may or may not have to determine whether the task is ready for execution. Section 5.3.2 details how a worker performs this function. If the task is not ready then the worker becomes idle and fetches another task. If the task is ready then the worker executes the task by invoking the associated task-procedure. Once the execution of the task completes, the worker becomes idle and fetches another task. The worker terminates upon receiving a special task called `exit_task` which is distributed to the worker during the system take-down process.

5.2 Task Synchronization

Tasks are synchronized according to the data regions they access. A task can access a region only if such access does not violate the execution semantics of the sequential program. Hence, as stated in Section 4.4, tasks in pTask are synchronized according to the following rule: *a task can access a region if the region is not in conflict with regions of tasks that execute and complete earlier (than the task) as procedure invocations in the sequential program.* For a given task, we refer to these tasks as its *earlier* tasks, and to the order in which tasks would be executed if they were to be executed sequentially as *task serial execution order*.

Since tasks are dynamically created and concurrently executed, task serial execution order is not necessarily the order in which tasks are created in time. For instance, for the program shown in Figure 5-2, according to the task serial execution order, task t_{1_1} must be executed and completed before task t_2 , although in a parallel execution, task t_2 may be created before task t_1 is executed, and even before task t_{1_1} is created. Consequently, in order to support the above synchronization rule, the run-time

```

/*$ann [TASK()] */
void t1_1 () {
    ...
}
/*$ann [TASK()] */
void t1 () {
    t1_1 ();
}
main () {
    t1 ();
    t2 ();
}

/*$ann [TASK()] */
void t2_1 () {
    ...
}
/*$ann [TASK()] */
void t2 () {
    t2_1 ();
}

```

Figure 5-2: Serial execution order of tasks

system must first establish task serial execution order before it can use the regions accessed by tasks to detect and enforce inter-task data dependencies in order to exploit concurrency. The run-time system establishes task serial execution order by maintaining task serial data access order using the regions supplied by the compile-time module. The subsequent sections explain how the run-time system maintains this order with data structures which we refer to as *region-lists* and how it uses these region-lists to detect and enforce inter-task data dependencies.

5.2.1 Region-lists

Dynamically created and maintained by the run-time system, a *region-list* is a list of all regions of a variable currently being accessed by tasks in the program. Each node on the list contains the descriptor of the region, the id of the task that accesses the region, and a list of pointers to tasks waiting for this region. For simplicity, we refer these nodes as regions on the region-lists. Each region-list is maintained such that the order of the regions on the list reflects the serial order in which tasks access the corresponding variable. That is if tasks t_1 and t_2 access a variable A , and task t_1 accesses A earlier than task t_2 according to the serial execution order, then on the region-list of A , the regions of task t_1 are in *front* of those of task t_2 .

Associated with region-lists are three operations: `region_reserve`, `region_unreserve`, and `region_check`. The first two operations insert and remove regions to and from region-lists, respectively. The third operation determines whether a region on a list is in conflict with regions of the earlier tasks.

Region_reserve

Given a region of a task, this operation atomically inserts the region onto the region-list of the associated variable. The region is inserted such that its position on the list reflects the serial data access order of tasks.

In a sequential program, when a procedure invokes subprocedures, the subprocedures execute one at a time, each to completion, in the order in which they are invoked. If these subprocedures access the same data, then they would access the data in the order in which they are invoked, and a subprocedure always completes its access to the data before returning to the calling procedure. In the framework of `pTask` where a procedure invocation corresponds to a task, this implies sibling tasks access the data in the order in which they are invoked, and a task always completes its access to the data before its parent continues to access the same data. Thus, regions of a task can be inserted onto region-lists relative to the regions of its parent and siblings. More specifically, the regions of a task should be inserted in front of the regions of its parent, and among the regions of its siblings, reflecting the order in which it and its siblings are invoked as procedures in the sequential program. Since a task creates child tasks sequentially, reserving a region of a task essentially amounts to inserting the region *immediately in front* of the regions of its parent — provided the parent task reserves regions of its child tasks in the order in which the child tasks are created, which is the case as will be discussed in Section 5.3.1.

Assuming the tasks in the program in Figure 5-2 access some global array `A`, Figure 5-3 shows the result of following sequence of `region_reserve` operations¹:

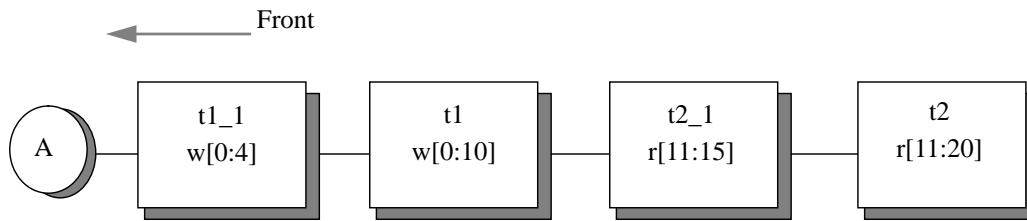


Figure 5-3: Region reservation for tasks in Figure 5-2

1. Reserving the region `write [0:10]` of task t_1 , which is the first child task of the main task. This region is inserted immediately in front of the regions of its parent, which is the main task. The main task is considered to access all regions in the program; however, its regions are not explicitly reserved. Since the main task is ancestor of all tasks, to insert a region in front of the regions of the main task is simply to insert the region at the end of the corresponding list. Hence, this region is inserted at the end of the list.
2. Reserving the region `read [11:20]` of task t_2 , which is the second child task of the main task. Similar to the region of task t_1 , this region is inserted immediately in front of the regions of the main task, which is again at the end of the list.
3. Reserving the region `write [0:4]` of task t_{1_1} , which is a child task of task t_1 . This region is inserted immediately in front of the region of task t_1 because task t_{1_1} is a child task of task t_1 .
4. Reserving the region `read [11:15]` of task t_{2_1} , which is a child task of task t_2 . This region is inserted immediately in front of the region of task t_2 since task t_{2_1} is a child task of task t_2 .

This example shows that although regions are reserved in an order that is different from the order in which tasks are invoked as procedures in the sequential execution, the

1. In the figure, the name of the variable is placed at the front of the list, and each region is represented by a rectangular box which contains the id of the task accessing the region, the ranges of the region, and the type of access (*r* for reading, *w* for writing)

`region_reserve` operation maintains the serial data access order of tasks on region-lists, as long as the parent-child relationship among the tasks is known.

Region_unreserve

Given a region on a region-list, this operation atomically removes the region from the region-list. If there are tasks waiting on this region then `region_unreserve` resumes these tasks. The process of task resumption is explained in Section 5.3.3. The `region_signal` operation in the previous chapter directly translates into this operation.

Region_check

While the previous two operations are used to maintain task serial data access order, this operation is used to detect inter-task data dependencies. Given a region on a region-list, this operation determines whether the region is in conflict with those of the earlier tasks. This operation involves traversing the associated region-list starting from the given region toward the front of the list, checking the region against those on the path until the head of the list is reached or a *conflicting* region is found; two regions are in conflict if their intersection is not empty and at least one has the write access type. In the case of a conflict, `region_check` returns the conflicting region. For example, given the region-list in Figure 5-3, a `region_check` on the region of task `t2` would return no conflicting region, while a `region_check` on the region of task `t1` would return the region of task `t1_1`.

5.2.2 Establishing Serial Data Access Order

In Section 5.2.1, we have shown by means of an example that `region_reserve` operations insert regions onto region-lists according to serial data access order of tasks, regardless of the order in which the respective `region_reserve` operations were performed. In this section, we show that in general for any two tasks `t1` and `t2`, region-lists always reflect the serial order in which these tasks access shared variables. That is, we

show that if task t_1 is earlier than task t_2 , and both access some common variable then, on the region-list of the variable, regions of t_1 are always in front of those of t_2 . We show this on the basis that all tasks in the system originate from the main task. From the description of `region_reserve`, it is easily seen that if tasks t_1 and t_2 have either sibling or parent-child relationship then regions of t_1 would appear in front of those of t_2 . Hence, we consider the case in which t_1 and t_2 are not sibling and have different parents.

Assume tasks t_1 and t_2 have tasks t_{1_p} and t_{2_p} as their respective parents. If task t_1 is earlier than t_2 , then task t_{1_p} is also earlier than task t_{2_p} , as dictated by the order in which the corresponding task-procedure are invoked in the sequential program. The same argument holds for tasks t_{1_p} and t_{2_p} , i.e., the parent of task t_{1_p} is also earlier than the parent of task t_{2_p} . Let us denote by t_{1_e} and t_{2_e} the immediate children of the youngest common ancestor of t_1 and t_2 , such that t_{1_e} and t_{2_e} are ancestors of t_1 and t_2 , respectively. By the same argument above, task t_{1_e} is also earlier than task t_{2_e} . Since t_{1_e} is an earlier sibling of t_{2_e} , regions of t_{1_e} are in front of those of t_{2_e} . In addition, since regions of a child task are always inserted *immediately* in front of those of its parent, regions of all t_{1_e} 's descendants are in front of those of task t_{1_e} , and regions of all t_{2_e} 's descendants are in front of those of task t_{2_e} , behind those of task t_{1_e} . This implies regions of task t_1 are always in front of those of task t_2 .

$p\bar{T}ask$'s approach to ordering regions in serial access order is similar to that of Jade, and indeed to any system that maintains serial execution semantics. The difference between $p\bar{T}ask$'s approach and that of Jade is that region-lists represent accesses to variables, and hence dependence and parallelism must still be extracted by $p\bar{T}ask$'s run-time system. In Jade, it is synchronization objects (tokens) that are maintained in the serial access order, and are simply accessed one at a time in the order of their appearance on the lists.

5.2.3 Enforcing The Synchronization Rule

The run-time system maintains sequential execution semantics. That is, given any two arbitrary tasks t_1 and t_2 such that t_1 is earlier than t_2 , and they access some common data, the run-time system will always detect and enforce the data dependencies that exist between these two tasks. This can be easily seen because:

1. As shown in Section 5.2.2, the `region_reserve` operation will always insert the regions of t_1 ahead of the regions of t_2 on region-lists to indicate task t_1 being earlier than task t_2 , regardless of the order of their creations.
2. Each of t_1 and t_2 must execute a `region_check` before accessing the data and a `region_signal` after accessing the data. The calls to `region_check` and `region_signal` are inserted by the compile-time module as described in Chapter 4. If a region of task t_2 is in conflict with a region of task t_1 then `region_check` returns the conflicting region and task t_2 will be forced to wait until task t_1 issues a `region_signal` on the conflicting region. When `region_check` on the regions of task t_2 returns no conflicting region, then the run-time system allows t_2 to access its regions. In effect, the run-time system enforces the synchronization rule stated at the beginning of this section for any two arbitrary tasks.

It is easy to see that deadlock will not occur in this system because, by means of region-lists and region operations, the system maintains sequential execution order of tasks performing conflicting operations on same data.

5.3 Task Phases

This section describes how the run-time system coordinates tasks by detailing the events in the phases of a task. A task goes through three phases; namely, creation, execution and possibly suspension, and resumption if it has been suspended. The subsequent sections discuss each phase in turn.

5.3.1 Task Creation

A `create_task` call in the parallel program performs three functions. First, it creates a data structure representing the new task. Contained within this structure are the task id, the parent task id, the pointer to the associated task-procedure, the task-context, and the regions of the task-procedure invocation. Next, it establishes the serial data access order for the new task by reserving all regions of the task using the `region_reserve` operation. Finally, it adds the new task to the global task queue.

The fact that a task creates child tasks sequentially and regions of child task are reserved sequentially at task creation time allows region reservation to be carried out efficiently. Each task maintains pointers to the first of its regions on each region-list. Since a task has its parent task id, to reserve a region of a new task is simply to insert the region in front of its parent's first region on the corresponding region-list. Thus, although `region_reserve` is an atomic operation, it is performed efficiently without having to traverse the list.

5.3.2 Task Execution

Section 4.4.1 discussed how users can control task-procedure entry synchronization. If this synchronization is to be performed by the run-time system, then given a task, a worker executes the task when the regions of the task are not in conflict with those of its earlier tasks. To determine whether the task is ready for execution, the worker issues a `region_check` on all of its regions. If the `region_check` returns no conflicting region, the worker executes the task. Otherwise it adds a pointer to the task to the conflicting region's list of waiting tasks, and becomes idle, fetching another task. Section 5.3.3 explains how a waiting task is eventually resumed.

On the other hand, if task-procedure entry synchronization is to be performed by the `region_check` operations inside the task, then the worker executes a task as soon as it receives the task. During the execution of the task, if a `region_check` returns no

```

/*$ann [TASK()]*
void t2 (int* a) {
    a[0] = ...;
}
f1 (int* a) {
    t2 (a);
}
/*$ann [TASK()]*
void t1 (int* a) {
    f1 (a);
    region = write a[1:1];
    region_reserve (region);
    region_check (region);
    a[1] = ...;
    region_unreserve (region);
}

```

Figure 5-4: Procedure-body synchronization

conflicting region, the execution is continued. Otherwise, the worker adds a pointer to the task to the conflicting region's list of waiting tasks, saves the task's execution context, and becomes idle, fetching another task.

As discussed in Section 4.4.2, synchronization also occurs during the execution of a task. The example in Section 4.4.2 (Figure 4-9) is again shown in Figure 5-4, with synchronization code being translated into appropriate region operations². In Figure 5-4, task t_1 can only execute the assignment statement if the region of the statement is not in conflict with those of task t_2 ; otherwise task t_1 must wait until task t_2 completes its access to its regions. As shown in the figure, this synchronization is realized by performing a `region_reserve` followed by a `region_check` on the region of the assignment statement. If the `region_check` returns no conflicting region, then the worker continues task t_1 's execution; otherwise the task is suspended. Section 5.3.3 explains how a suspended task is eventually resumed.

2. `Region_check` is translated into `region_reserve` and `region_check`. `Region_signal` is translated into `region_unreserve`.

5.3.3 Task Resumption

A task may have to wait for a region of another task or for its child tasks to terminate. Once a task completes its accesses to a set of regions, a `region_unreserve` is called for each of these regions. For each unreserved region, the worker obtains the list of tasks waiting on the region and adds these tasks back to the task queues. A task that is waiting for a region may or may not have started its execution, depending whether task-procedure entry synchronization is performed by the run-time system or by the `region_check` operations inside the task. If the task has not started its execution, the worker would add the task back to the global task queue, and a worker would eventually continue to determine whether the task is ready for execution. If the task has started its execution, it would be added to the local task queue of the same worker who was executing it, and this worker would eventually restore the task's execution context and continue with its execution.

In the case where the task must wait for its child tasks to terminate, the child task which terminates last adds the parent task back to the local task queue of the worker on which the parent task was executing. This worker would eventually resume the parent task.

5.4 Scheduling Tasks for Data Locality

Given a task, the default strategy is for the task to be executed by the first idle worker that becomes available. This scheduling method maintains the load among the processors (workers) fairly balanced and incurs little run-time overhead. However, it does not take into account the fact that on most scalable shared-memory multiprocessors such as the KSR1, the time it takes a processor to access data depends on the location of the data with respect to the processor. The farther away data is from a processor, the longer it takes the processor to access the data. Thus, this default strategy may result in poor data locality.

There are applications in which performance can benefit from a scheduling strategy that assigns tasks to processors such that data locality is enhanced. Such a strategy assigns tasks referencing same data to the same processor, so that the processor can reuse the data brought into the local memory by previous tasks. However, such a scheduling practice often incurs run-time overhead, especially if the system has to measure task-processor data affinity at run-time. In addition, scheduling tasks for data locality is often in conflict with scheduling tasks for load balance since, at the extreme, the best data locality is achieved by scheduling all tasks on one processor. Conversely, in order to obtain good load balance, tasks may have to be distributed among a large set of processors and poor data locality may result.

The run-time system provides an alternate scheduling strategy which exploits data locality without sacrificing load balance. Given a task, this strategy assigns the task to the *idle* worker that has highest data affinity for the task. This scheduling strategy incurs more run-time overhead since task-processor data affinity must be determined at run-time. Users can execute a pTask-generated program with the default strategy which incurs little run-time overhead, or with this alternate strategy which incurs more overhead, but may improve the performance of the application³. The remainder of this section describes this strategy.

5.4.1 Accessed Region-lists

For each region-list, the run-time system maintains a corresponding list of unreserved regions. The `region_unreserve` operation is augmented to append removed regions from a region-list onto the corresponding unreserved list. These unreserved lists are referred to as *accessed-lists* because regions on these lists are regions that have already been accessed by tasks. Each node on the accessed-list has the identifier of the worker which executed the task that accessed the region. For simplicity, we refer these nodes also

3. This scheduling strategy is determined by the environment variable `PTASK_LOCALITY`.

as regions on the accessed-lists. Since regions are appended onto these lists in roughly the order in which they were accessed, regions at the rear of an accessed-list are more recently accessed in comparison to those at the front.

5.4.2 Measuring Task-Worker Data Affinity

Accessed-lists are used to measure task-worker data affinity. Since a task may access many regions that were accessed by tasks executed on different workers, there may be more than one worker having data affinity for the task. Given a task that is ready for execution, for each of the task's regions, the worker determines the set of workers having data affinity for the region and the extent of affinity that each of these workers has for the region. The worker computes data affinity for all regions for each of these workers and assigns the task to the idle worker having the highest data affinity.

The main algorithm for this scheduling strategy is outlined in Figure 5-5. The algorithm determines the worker that has the highest data affinity for a given region. The full strategy is a simple extension of this algorithm. Given a region, a worker locates the *latest copy* of the region by traversing the associated accessed-list from the rear toward the front, intersecting the given region with those on the path. We use the amount of data in the intersection to determine how much of the given region was accessed by a worker. We also use the accessed-regions' access types to determine whether the elements accessed are of latest values. The algorithm takes into account the fact that an accessed-region may only intersect *part* of the given region, and the intersection of the *last* (as positioned on the accessed-list) accessed-region of write access type contains latest elements of the given region. The algorithm stops when the front of the accessed-list is reached, or when an intersection that is the latest copy of the given region is found. The worker that accessed most of the latest copy is the one having the highest data affinity for the given region.

It is important for the run-time system be as efficient as possible in the process finding the worker with highest data affinity. Otherwise the overhead from having to

```

Input: Region R of a task
Output: Id of the worker that accessed most of the latest copy
           of R and the number of elements of R that was accessed
Comment: Each node on accessed-list contains id of worker accessed the
           region in the node
begin
  for each region (called LESSR_R for less recent) from the rear to the
  front of the associated accessed-list
    LESSR_ISIZE = number of elements R and LESSR_R intersect
    if (LESSR_R is the region at the rear)
      LATEST_R = LESSR_R, LATEST_ISIZE = LESSR_ISIZE
    else if (LESSR_ISIZE > LATEST_ISIZE)
      if (LATEST_R is of read type)
        // LESSR_R contains a larger portion of latest copy
        LATEST_R = LESSR_R, LATEST_ISIZE = LESSR_ISIZE
      else // LATEST_R is of write type
        if (WORKERID of LESSR_R == WORKERID of LATEST_R
            or LESSR_ISIZE - LATEST_ISIZE > LATEST_ISIZE)
          // If these regions were accessed by same worker then
          // LESSR_R contains a larger portion of latest copy.
          // If these regions were accessed by different workers
          // then LESSR_R contains thus far the largest portion of
          // latest copy only if it intersects R more than
          // LATEST_R intersects R by the amount of LATEST_R
          LATEST_R = LESSR_R, LATEST_ISIZE = LESSR_ISIZE
        endif
      endif
    endif
  endif
  if (LATEST_ISIZE = size of R)
    // LATEST_R contains all of R
    return WORKERID of LATEST_R, and LATEST_ISIZE
  endif
endfor
return WORKERID of LATEST_R, and LATEST_ISIZE
end

```

Figure 5-5: Algorithm finding worker having highest data affinity for a region

maintain and traverse the accessed-lists would outweigh the benefit from enhancing data locality. The system performs two optimizations. First, since regions are always appended at the rear of accessed-lists, and a list is always traversed starting from the rear (tail), the run-time system allows the lists to be traversed while being updated. In other words, to traverse a list, a worker only has to lock, obtain and unlock the tail, and then traverses the list. While workers are traversing the list, other workers may atomically append nodes to the tail of the list. Second, the search for the worker having the highest data affinity for a region terminates as soon as the latest copy of the region is located. Consequently, a worker does not always have to traverse the whole list to determine task-worker data affinity.

5.5 Summary

The run-time system provides an environment to create, schedule, execute, and synchronize tasks. Its main objective is to coordinate tasks to exploit concurrency and, at the same time, to maintain the sequential execution semantics. It realizes this by first preserving the serial data access order of tasks on region-lists, and detecting the data dependencies among tasks based on the regions they access and on the order in which they access the regions. Regions are also used to determine data affinity for tasks. To achieve good performance, the run-time system balances workload among the processors and provides users the option to schedule tasks for data locality.

Chapter 6

Experimental Results

A prototype of pTask has been implemented on a KSR1 multiprocessor. This chapter presents the experimental results for a number of applications showing that good performance improvements can be obtained by using pTask. For some of the applications, we compare the performance of pTask-generated programs to that of manually-parallelized programs or of manually-optimized programs.

The remainder of this chapter is organized as follows. Section 6.1 gives an overview of the KSR1 multiprocessor. Section 6.2 describes the metrics used to measure the performance. Section 6.3 to Section 6.9 present the experimental results. Section 6.10 discusses the impact of scheduling tasks for data locality. Finally, Section 6.11 summarizes and concludes.

6.1 The KSR1 Multiprocessor

The KSR1 [3][17] is a COMA (Cache Only Memory Architecture) shared-memory multiprocessor consisting of 32 up to 1088 processing modules connected by a

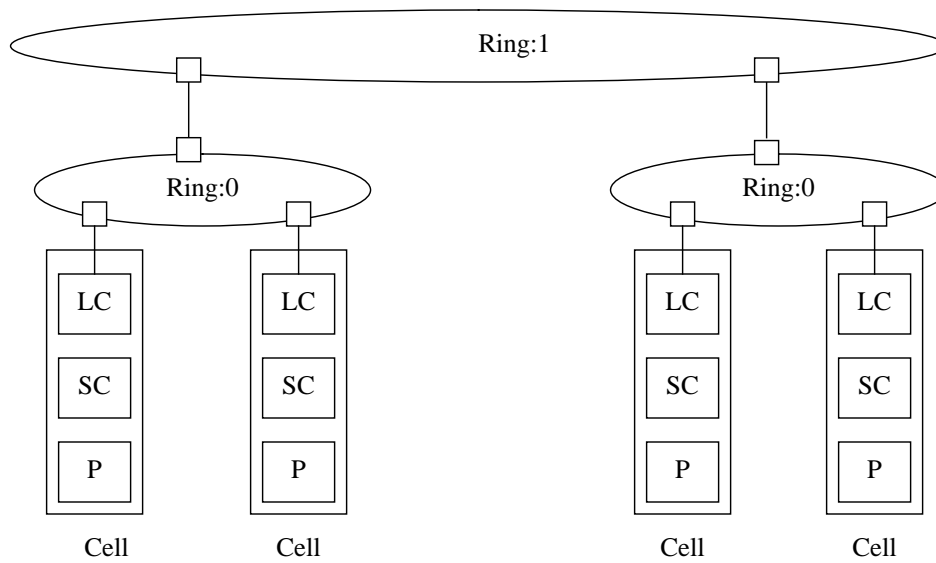


Figure 6-1: The KSR multiprocessor

Data Location	Processor Cycles
Subcache	2
Local cache	20 - 24
Same ring:0	175 - 200
Different ring:0	600

Table 6-1: Memory access times for KSR system

hierarchical ring interconnection network, as shown in Figure 6-1. Thirty two processing modules (referred to as cells) are connected to a ring, referred to as ring:0. Up to 34 ring:0 rings can be connected to a second-level ring, referred to as ring:1. Each cell is a 64-bit processor with 512-Kbyte first level cache called the *subcache* and 32-Mbyte of main memory called the *local cache*. The subcache is divided into 256-Kbyte for instruction and 256-Kbyte for data. The local cache is 16-way set associative, and its 128-byte cache line is referred to as a *subpage*. The physical distribution of the shared memory on the KSR system results in non-uniform memory access times. As shown in Table 6-1, the farther away data from a processor, the longer it takes the processor to access the data. The

KSR1 multiprocessor on which our experiments were conducted consists of a single ring:0 with 32 processors.

6.2 Performance Metrics

The performance of a parallel program is measured using the following metrics:

1. The sequential execution time T_s , which is the execution time of the sequential program on a single processor.
2. The parallel execution time T_p , which is the execution time of the parallel program on p processors.
3. The speedup, which is the ratio of T_s to T_p .
4. The task run-time overhead T_o in the pTask-generated program. For each task, the run-time system maintains a run-time overhead t_o , which is the total time the run-time system spends servicing calls such as `create_task`, `region_reserve`, `region_check`, `region_unreserve`, and `children_wait`. These calls may cause the task to suspend; however, t_o does not include the time the task suspends. The time T_o is the sum of t_o over all tasks.
5. The percentage of overhead δ , which is the percentage of the task run-time overhead T_o to the sequential execution time T_s , i.e., $\delta = T_o / T_s \times 100$.
6. The task computation time T_c in the pTask-generated program. For each task, the run-time system maintains the time t_c the run-time system spends executing the code of the associated task-procedure. It is the time period measured between the invocation and return of the task-procedure, excluding the run-time overhead t_o and the time period in which the task suspends. The time T_c is the sum of t_c over all tasks.

If a pTask-generated program performs same amount of computation as its sequential counterpart, then ideally, T_s should be equal to T_c . However, this is not always the case. On the KSR1, T_s and T_c differ mainly due to:

1. Non-uniformity in memory access times. For instance, if the data is too large to fit in the local memory of one processor, then the sequential program spends more time accessing a portion of the data remotely. In order to reduce this impact on the speedup of the parallel program, we select the data set size so that the data fits within the local memory of a processor. Conversely, if the parallel program exhibits poor data locality then it spends more time accessing the data remotely in the parallel execution.
2. Caching effects. For instance, threads in the parallel program access data smaller in size than the sequential program. The probability of this data fitting and remaining in the cache of the processor executing the thread increases as the number of processors increases. This in turn reduces the number of accesses memory to memory and improves execution time.
3. False sharing, which occurs when processors in the parallel program write the same cache line, but reference different parts of the cache line. In this case, a processor may have to fetch the same cache line on every write because the cache line may have been invalidated by the write of another processor. This leads to what is known as the “ping-pong” effect[2].

The metrics δ , T_s , and T_c can be used to infer whether the performance of a pTask-generated program is affected by the overhead in the run-time system or by data access patterns of tasks. For example, when T_c is much larger than T_s , an analysis of the data access patterns can reveal whether the cause is due to poor data locality and/or false sharing.

```
/* c = a x b. All matrices are of size n x n */
mm (a, b, c, n) {
    for row = 0 to n - 1
        mm_row (a, b, c, n, row)
    endfor
}
mm_row (a, b, c, n, row) {
    for k = 0 to n - 1
        r = a[row][k]
        for j = 0 to n - 1
            c[row][j] = c[row][j] + r * b[k][j]
        endfor
    endfor
}
```

Figure 6-2: Matrix multiplication

Another source of overhead which does not exist in the sequential programs is memory overhead. Each task incurs an overhead of 640 bytes. Each region descriptor occupies 96 bytes. Each node on a region-list is only 32 bytes. The memory space for these objects are allocated as required and de-allocated when no longer needed.

In the following sections, we present the experimental results for seven applications: matrix multiplication, search, multidimensional polynomial interpolation, narrowband tracking radar, mergesort, quicksort, and the traveling salesman problem. We executed the pTask-generated programs without having the `PTASK_LOCALITY` option being activated and both with the `PTASK_CHECK_VAR` option on and off. Better results were obtained with this option being off, and are the ones presented.

6.3 Matrix Multiplication

Matrix Multiplication (MM) calculates the product of two matrices. Figure 6-2 shows a matrix multiply algorithm in which procedure `mm` calls procedure `mm_row` to calculate a row of the product matrix.

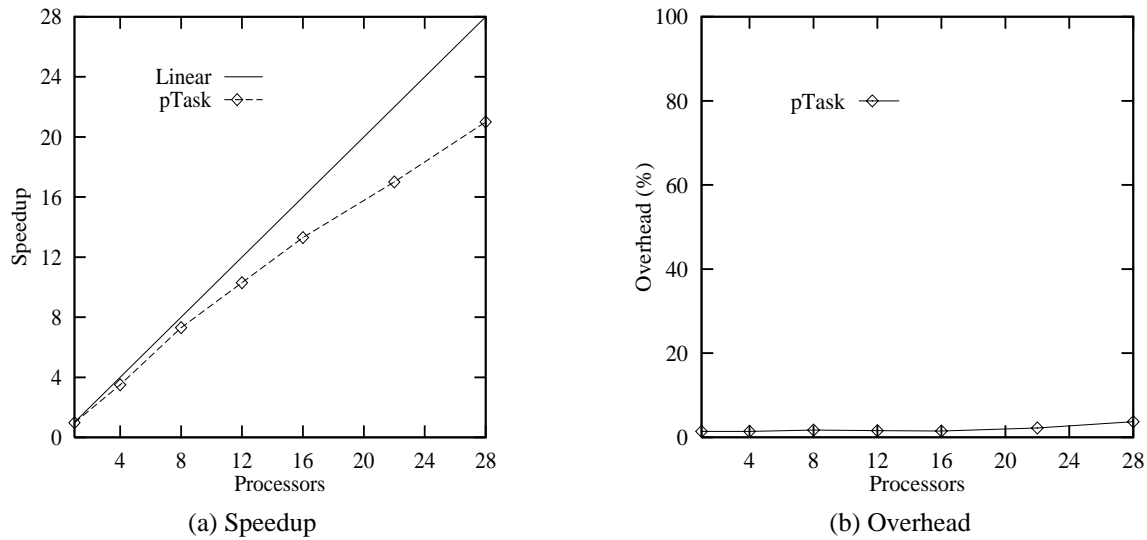


Figure 6-3: Performance of matrix multiplication (512x512 matrices)

For the pTask version of MM, we declare procedure `mm_row` as a task-procedure since it contains the bulk of the computation. Thus, the parallel task MM loops and creates tasks to produce the rows of the product matrix. These tasks may be executed concurrently since they each produce a distinct row of the product matrix. Hence, MM serves as a representative of data-parallel applications.

Figure 6-3 shows the performance of the pTask-generated MM for 512x512 matrices. The speedup from the pTask-generated MM is close to linear speedup, much attributed to the fact that MM is a data-parallel application and the overhead in the run-time system is quite low. With this application, we have demonstrated that pTask can automatically and efficiently exploit parallelism in data-parallel applications such as MM.

6.4 Search

Search is a program from Stanford University that uses a Monte-Carlo method to simulate the elastic scattering of electrons from an electron beam into various solids. The main computation simulates six different solids at 10 initial beam energies. Each solid-energy simulation tracks 5,000 electron trajectories.

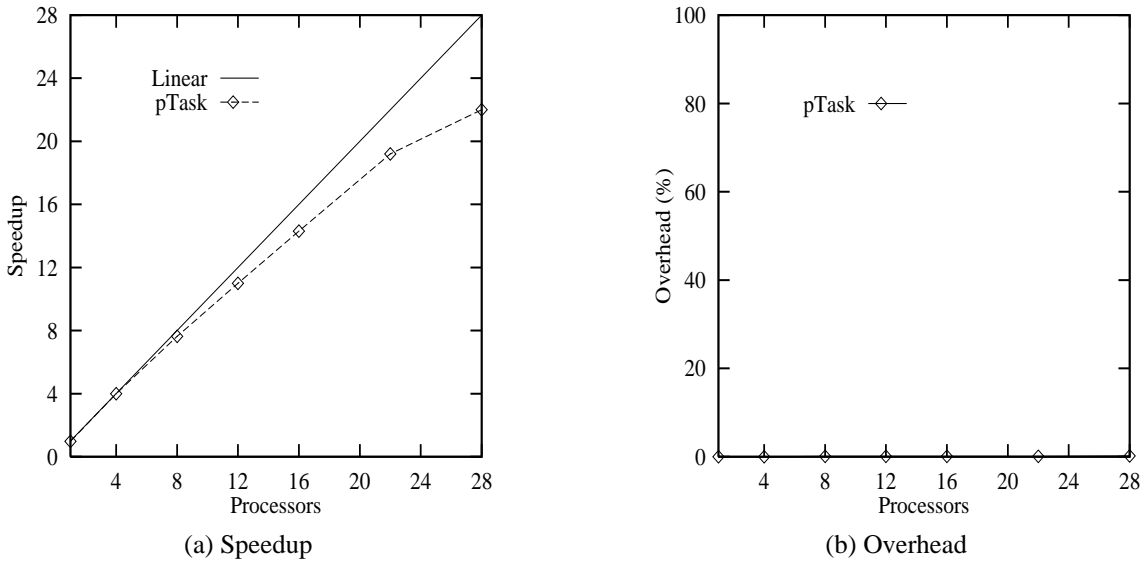


Figure 6-4: Performance of Search

For the pTask version of Search, we declare the procedure that performs a solid-energy simulation as a task-procedure. Tasks performing solid-energy simulations can execute concurrently. Figure 6-4 shows the performance of the pTask-generated Search. The results indicate negligible run-time overhead since the size of the tasks in this application are quite large. The speedup for this application is close to linear.

6.5 Multidimensional Polynomial Interpolation (MPI)

The main algorithm of MPI [22] is shown in Figure 6-5. For the pTask version of MPI, we declare procedure `polint`, which performs one-dimensional interpolation, as a task-procedure. Thus, the parallel task MPI simply creates tasks to perform one-dimensional interpolations. The tasks performing one-dimensional interpolation in the loop may be executed concurrently since they each produces a distinct element of array `ytmp`. The task performing the last interpolation may not proceed until these tasks completed because it consumes array `ytmp`. The task graph in Figure 6-6 depicts this sequence of execution.

```

/*
Given arrays x1a[1..m] and x2a[1..n] of independent variables, and a
submatrix of function values ya[1..m], tabulated at the grid points
defined by x1a and x2a; and given values x1 and x2 of the independent
variables; this routine returns an interpolated function value y, and
an accuracy indication dy.
*/
polin2 (x1a, x2a, ya, m, n, x1, x2, y, dy) {
  ymtmp = vector (1, m) /* allocate vector size [1..m] */
  for j = 1 to m
    polint (x2a, ya[j], n, x2, ymtmp[j], -1)
  endfor
  polint (x1a, ymtmp, m, x1, y, dy)
}
/*
Given arrays xa[1..n] and ya[1..n], and given a value x, this routine
returns a value y, and an error estimate dy if dy is not -1
*/
polint (xa, ya, n, x, y, dy){
  ...
}

```

Figure 6-5: Multidimensional polynomial interpolation

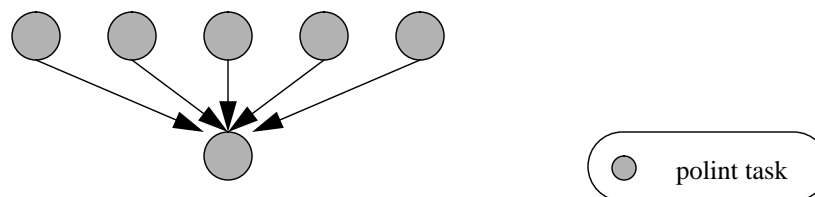


Figure 6-6: Task graph for MPI

Figure 6-7 shows the performance of the pTask-generated MPI for m and n set to 128 and 256, respectively. The results indicate low run-time overhead and good speedup. The speedup for MPI is not as high as that for matrix multiplication because the last `polint` task in the task graph represents a sequential fraction in the parallel program.

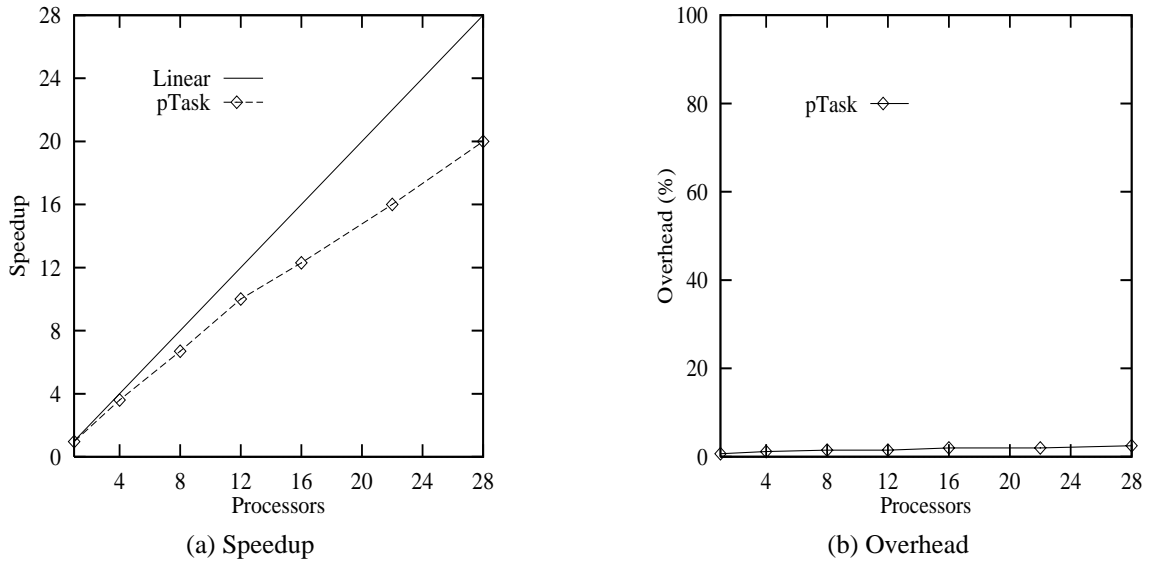


Figure 6-7: Performance of MPI ($m = 128$, $n = 256$)

6.6 Narrowband Tracking Radar

The narrowband tracking radar (radar for brief) [7] benchmark was developed by researchers at MIT Lincoln Labs to identify targets from a sequence of radar returns. The program inputs data from a single sensor along a number of independent channels. Every 5 milliseconds, the program receives 512 complex vectors of length 10, one after the other in the form of an 10×512 complex matrix. Each input matrix is corner turned (transposed) to form a 512×10 matrix. Computations to identify targets are then performed on each 512×10 matrix.

The radar program is shown in Figure 6-8. The program consists of a loop that iterates m times. For each iteration, procedure `radar` performs the following four steps. First, it calls procedure `receive_data` to input data into the channels of array `A`. It then calls procedure `fill_target` to fill the channels of array `A` with targets. Next, it invokes procedure `corner_turn` to turn a channel of array `A` into a channel of array `B`. Finally, it invokes procedure `compute` to identify the targets in the channels of array `B`.

```
radar (A, B) {  
  for j = 1 to m iterations  
    for i = 1 to nchannels  
      receive_data (A, i)  
    endfor  
    for i = 1 to nchannels  
      fill_target (A, i)  
    endfor  
    for i = 1 to nchannels  
      corner_turn (A, B, i)  
    endfor  
    for i = 1 to nchannels  
      compute (B, i)  
    endfor  
  endfor  
}
```

Figure 6-8: Narrowband tracking radar

For the `pTask` version of `radar`, we declare procedures `receive_data`, `fill_target`, `corner_turn`, and `compute` as task-procedures. For each iteration, the parallel task program executes as follows:

1. The `receive_data` tasks may be executed concurrently with one another since they each inputs a distinct channel of array A.
2. The `fill_target` tasks may be executed concurrently with one another since they each fills a distinct channel of array A with targets. A `fill_target` task can be executed as soon as the data along its channel is received, i.e., as soon as the corresponding `receive_data` task finishes.
3. The `corner_turn` tasks may be executed concurrently with each other since they each turns a different channel of array A into a channel of array B. A `corner_turn` task can be executed as soon as its channel is filled with targets, i.e., as soon as the corresponding `fill_target` task completes.

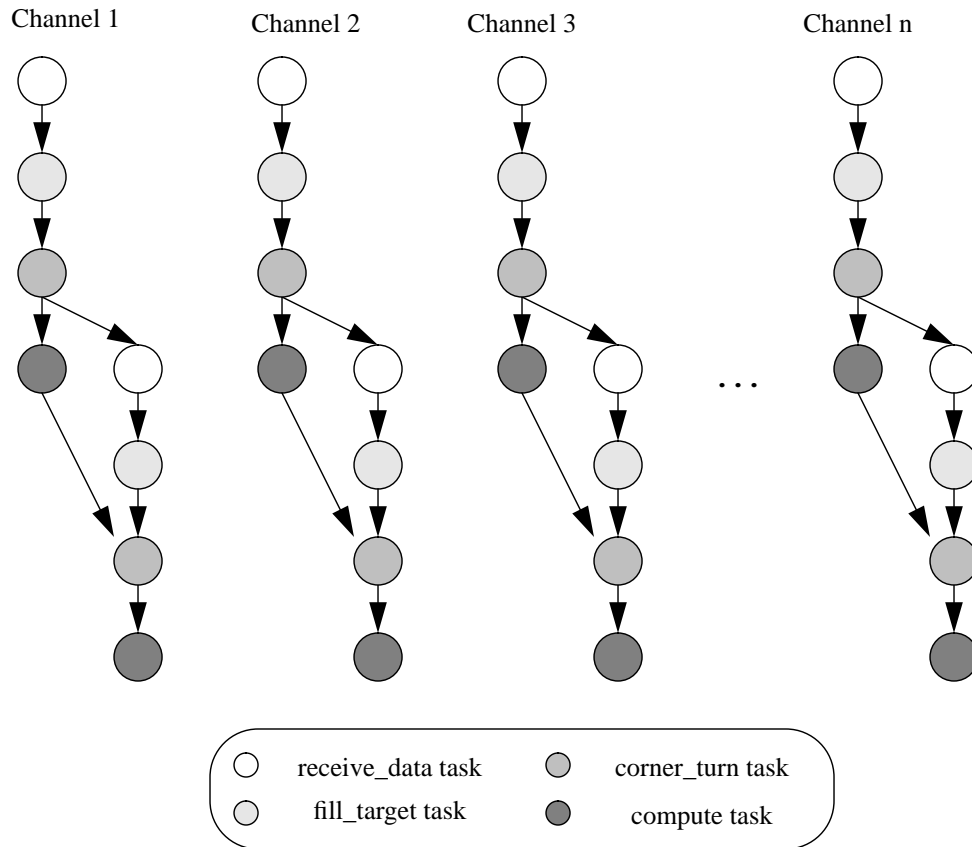


Figure 6-9: Task graph for radar (first 2 iterations)

4. The `compute` tasks may be executed concurrently with one another since they each computes along a distinct channel of array B. A `compute` task can be executed as soon as the data along its channel is ready, i.e., as soon as the corresponding `corner_turn` task completes.

Figure 6-9 shows the task graph for first two iterations of radar. The task graph indicates that there exist both loop-level and task-level parallelisms in this application. For instance, loop-level parallelism exists among the `compute` tasks from the first iteration and also among the `receive_data` tasks from the second iteration; while task-level parallelism exists among these two sets of tasks. Consequently, the tasks in these two sets may be executed concurrently.

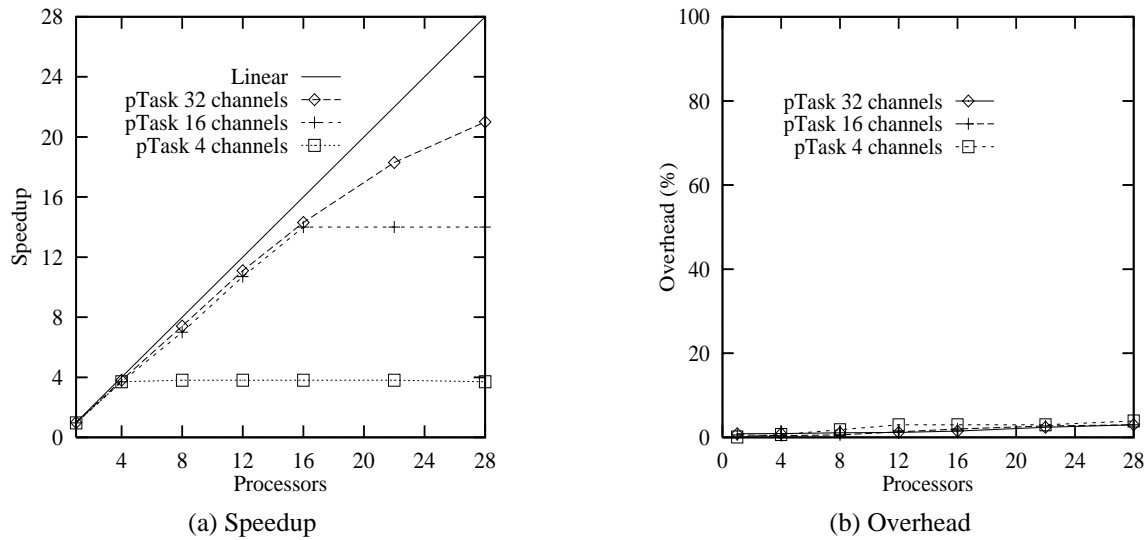


Figure 6-10: Performance of narrowband tracking radar

Figure 6-10 shows the performance of the pTask-generated radar program. Although the original program uses only 4 channels, we ran the application with 4, 16 and 32 channels. As shown in the figures, the run-time overhead is quite low and this is reflected in a speedup that is close to linear for the range of processors less than or equal the number of channels. The speedup for this application reaches a plateau when the number of processors equals the number of channels, although the additional processors can be and are utilized. The reason is that even though the `compute` tasks from one iteration are executed concurrently with the `receive_data` tasks from the next iteration, the size of the former set of tasks is small compared to the size of the latter tasks such that no noticeable improvement results from executing these tasks concurrently. A similar performance is found when this program is coded using Fx [7]. With this application, we have demonstrated that pTask can automatically and efficiently exploit task-level parallelism in a non-trivial benchmark program.

```
/* sorts array key of n elements */
mergesort (key, n) {
  for k = 1 to n-1
    for j = 0 to n-k-1
      merge (key + j, key + j + k, key + j, k, k)
      j = j + 2k
    endfor
    k = 2k
  endfor
}
merge (a, b, c, n, m){
  merges array a of n elements and b of m elements, and
  stores the result in c
}
```

Figure 6-11: A sequential mergesort

6.7 Mergesort

Mergesort is an external sort algorithm used when the data size is too large to fit within the main memory all at once. Mergesort sorts the input array in stages or passes. After the first pass, successive pairs of elements are in order. The number of successive elements in order is doubled after each pass. After the last pass, all elements are in order. At each pass, the desired ordering is achieved by merging the subarrays. For example, in the first pass, subarrays of one element are merged so that after the first pass, successive pair of elements are in order. Figure 6-11 outlines a mergesort algorithm in which procedure `mergesort` sorts the input array by calling procedure `merge` to merge the subarrays. This application is a representative of applications that have divide-and-conquer algorithm.

For the `parallel` version of mergesort, we declare procedure `merge` as a task-procedure since it is the core of the computation. Thus, the parallel task program basically loops and creates tasks to merge the subarrays. Each task merges two successive subarrays. A task is ready for execution when both input subarrays are in order. The last

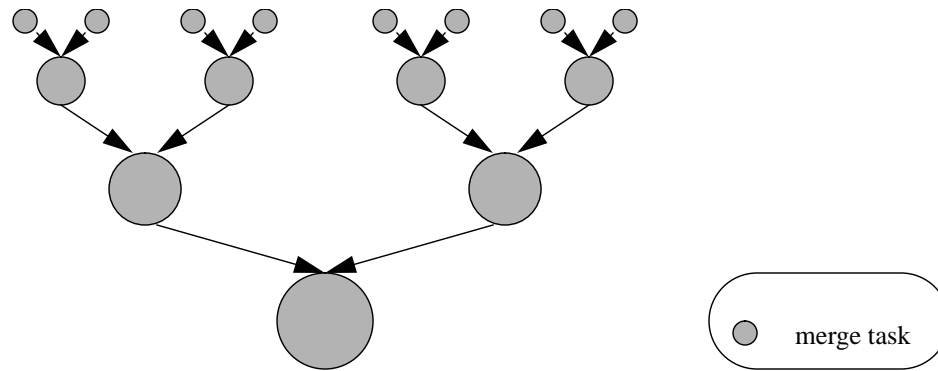


Figure 6-12: Task graph for mergesort

task merges the two halves of the array. The behavior of the parallel task program is depicted by the task graph in Figure 6-12.

As shown Figure 6-12, the task size doubles after each pass, or after each level in the task graph. At the extreme ends, the tasks at the top of the task graph merge subarrays of one element, while the task at the bottom merges two halves of the input array. To amortize the overhead required to exploit parallelism, we made simple modifications to the sequential program so that the tasks at the top of the graph merge subarrays of some reasonable minimum size instead of subarrays of a single element. Figure 6-13 shows the `pTask` version of mergesort with `MINSIZE` as the minimum size. Since the subarrays must be sorted before they can be merged, the subarrays of minimum size are sorted sequentially using the original sequential mergesort.

Figure 6-14 (a) shows the performance of the `pTask`-generated mergesort and of a manually-parallelized program which is implemented using the KSR1 pthreads library. The program is used to sort 1 M integers. To study the effects of the task size on the performance, we ran the `pTask`-generated program with `MINSIZE` set to 4096 and to $n/2p$ where n is the size of the array and p is the number of processors. We chose these

```

mergesort (key, n){
    for k = MINSIZE to n-1
        for j = 0 to n-k-1
            merge (key + j, key + j + k, key + j, k, k)
            j = j + 2k
        endfor
        k = 2k
    endfor
}
/*$ann [TASK()] */
void merge (a, b, c, n, m){
    if n <= MINSIZE
        sequential_mergesort (a, n)
    if m <= MINSIZE
        sequential_mergesort (b, m)
    ...
}

```

Figure 6-13: The pTask mergesort

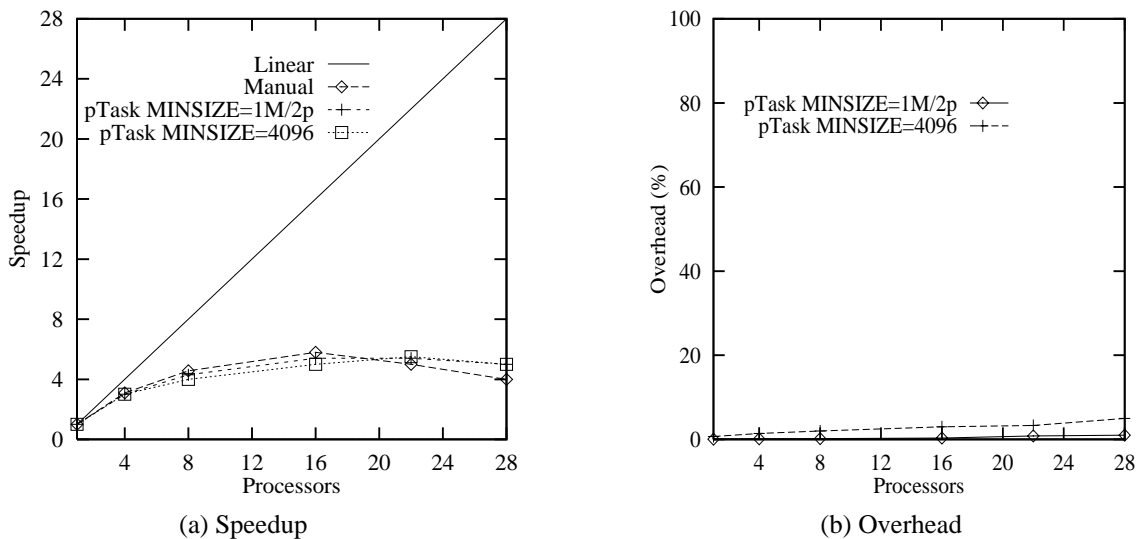


Figure 6-14: Performance of mergesort (1M integers)

values because 4096 ensures that even the smallest task would have reasonable size¹, while $n/2p$ conveniently divides the input array into large subarrays among the workers.

1. It takes less than a microsecond for a task to merge two subarrays of 1024 elements.

The results indicate that the best performance from the pTask-generated program is close to that of the manually-parallelized program, which is attributed to the low run-time overhead shown in Figure 6-14 (b). The results demonstrated that pTask can exploit task-level parallelism in applications that exhibit divide-and-conquer parallelism just as effective as manually-parallelized programs.

The results also show the effect of the task size on the performance. As `MINSIZE` is increased, the run-time overhead decreases, and this is reflected in the improved performance. However, it should be emphasized that `MINSIZE` should be chosen so that there are enough tasks to evenly distribute the load among the workers.

The speedup for this application is, however, far from perfect. This non-ideal performance is not due to pTask's inability to exploit parallelism; nor due to the run-time overhead; nor due to the run-time system's inability to balance the load; nor because poor data locality or false sharing since T_c is at most 6% greater than T_s . It is because of the lack of parallelism and the large sequential portion in the program. Indeed, the task graph in Figure 6-12 depicts the number of tasks being halved after every sorting pass such that eventually, there would not be enough tasks to keep all processors busy. In addition, the task graph also shows that the task size also increases as the degree of parallelism decreases (toward the bottom of the task graph), causing the parallel execution to spend more time in the less parallel portion of the program. In fact, the task at the bottom of the task graph represents a large sequential portion of the program. This lack of parallelism and the large sequential portion prevent the performance from being ideal.

In summary, we have again shown that pTask can automatically exploit task-level parallelism. In order to obtain good performance, the user is required to make minor modifications to the sequential program to ensure that tasks have sufficiently large grain sizes. We also have pointed out that the best performance from pTask for applications that exhibit divide-and-conquer parallelism such as merge sort is only limited by the nature of the computation.

```
/* sorts the elements in the subarray a[start:end-1] */
qsort (a, start, end){
    /*find pivot element in subarray a[start:end-1] and
    store the pivot element in pivot. Return zero if
    all elements have the same value.*/
    if (find_pivot (a, start, end, pivot) != 0) {
        /*partition the subarray a[start:end-1], given the
        pivot element. Return the size of the first partition*/
        k = partition (a, start, end, pivot);
        qsort (a, start, k);
        qsort (a, k, end);
    }
}
```

Figure 6-15: A sequential quicksort

6.8 Quicksort

Quicksort is perhaps the most widely used internal sort when all the data to be sorted fits entirely within the main memory. The algorithm is based on a divide-and-conquer approach. Given an array of elements, an element is chosen as a pivot and the array is partitioned so that the first part consists of elements whose values are all less than the pivot element, and the remaining part consists of elements whose values are all greater than or equal to the pivot element. The algorithm is then recursively applied on each subarray until the array is sorted. Figure 6-15 shows a quicksort algorithm in which procedures `find_pivot` and `partition` are used to find the pivot element and to partition the array, respectively. This application serves as a representative of recursive and divide-and-conquer applications.

For the pTask version of quicksort, we declare procedure `qsort` as task-procedure. Thus, in the parallel task program, whenever procedure `qsort` is invoked, a task is created to sort an array; and as long as the array can be partitioned, the task creates subtasks to sort the subarrays. Depending on the values of the data elements in the input array, the array may be partitioned into subarrays of unequal sizes. Consequently, it may

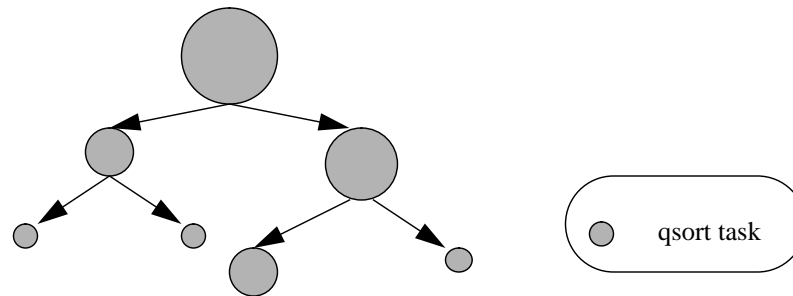


Figure 6-16: Task graph for quicksort

take longer to sort one partition of an array than the other partition. The task graph in Figure 6-16 depicts the behavior of the parallel task quicksort.

Tasks in quicksort do not all have the same size. The task at the top of the task graph operates on the whole input array, while tasks at the bottom handle small subarrays. To ensure that tasks have reasonable size, we made simple modifications to the sequential program so that if an array reaches a certain minimum size then it is sorted sequentially, rather than recursively by invoking subtasks. Figure 6-17 shows the `pTask` version of quicksort with `MINSIZE` as the minimum array size and procedure `sequential_qsort` as the sequential quicksort. We annotated the regions for calls to the recursive procedures since Sigma II does not provide regions for calls to these procedures. The annotation amounts to no more than recognizing which portion of the array is accessed, parameterized by procedure arguments.

Figure 6-18 (a) shows the speedup from the `pTask`-generated quicksort and from a manually-parallelized quicksort which is implemented using the KSR1 pthreads library. The program sorts 1 M integers. To study the effects of task size on the performance, we ran the `pTask`-generated program with `MINSIZE` set to 4096 and to $n/2p$ where n is the size of the array and p is the number of processors. These values for `MINSIZE` are chosen for the same reasons as in mergesort.

The results indicate that the best performance from the `pTask`-generated program is close to that of the manually-parallelized program, which is attributed to the low run-

```

/*$ann [TASK()] */
void qsort (a, start, end){
    if (find_pivot (a, start, end, pivot) != 0){
        k = partition (a, start, end, pivot);
        if k - start < MINSIZE
            /*$ann [REGION (w(a[start:k-1]))] */
            sequential_qsort (a, start, k);
        else
            /*$ann [REGION (w(a[start:k-1]))] */
            qsort (a, start, k);
        if end - k < MINSIZE
            /*$ann [REGION (w(a[k:end-1]))] */
            sequential_qsort (a, k, end);
        else
            /*$ann [REGION (w(a[k:end-1]))] */
            qsort (a, k, end);
    }
}

```

Figure 6-17: The pTask quicksort

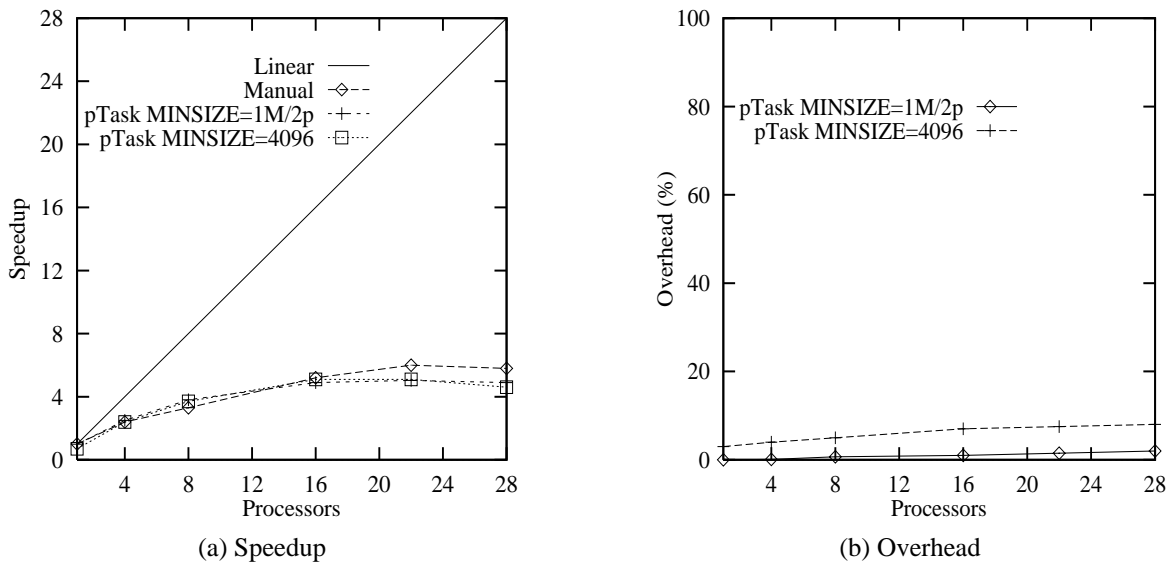


Figure 6-18: Performance of quicksort (1 M integers)

time overhead shown in Figure 6-18 (b). The results again indicates that pTask can

exploit task-level parallelism in applications that exhibit divide-and-conquer parallelism just as effective as manually-parallelized programs.

The results in Figure 6-18 show the familiar effects of the task size on the performance. As the task size increases, the performance improves since there is less runtime overhead. Figure 6-18 (b) shows small overhead when `MINSIZE` is small, and negligible overhead when `MINSIZE` is large.

In general, the speedup for quicksort is not ideal, much due to the divide-and-conquer nature of the algorithm. As depicted by the task graph in Figure 6-16, the program lacks parallelism since the number of tasks starts from one, and grows as the computation proceeds. In addition, the size of the tasks is largest where the degree of parallelism is least (top of task graph). The task at the top of the task graph represents a large sequential portion of the program. Consequently, the parallel execution spends most of the time in the less-parallel portion of the program. The performance of the parallel quicksort may suffer from additional lack of parallelism. Indeed, unlike in mergesort where the task graph resembles a balanced tree, depending on the values of the elements of the input array, the task graph for quicksort may become unbalanced. In short, the lack of parallelism, the large sequential fraction, and the additional load imbalance prevent the performance from being ideal.

In summary, we have demonstrated that, with `pTask`, users can easily achieve task-level parallelism and performance improvements even for programs with tasks that are recursive in nature. We also have shown that although the performance of the parallel program is limited by the characteristics of its corresponding sequential algorithm, reasonable speedup can still be obtained by using `pTask`.

6.9 TSP

In this problem, a salesperson visits N cities, returning finally to the city of origin. Each city is to be visited only once. Given the distances between all pair of cities, the

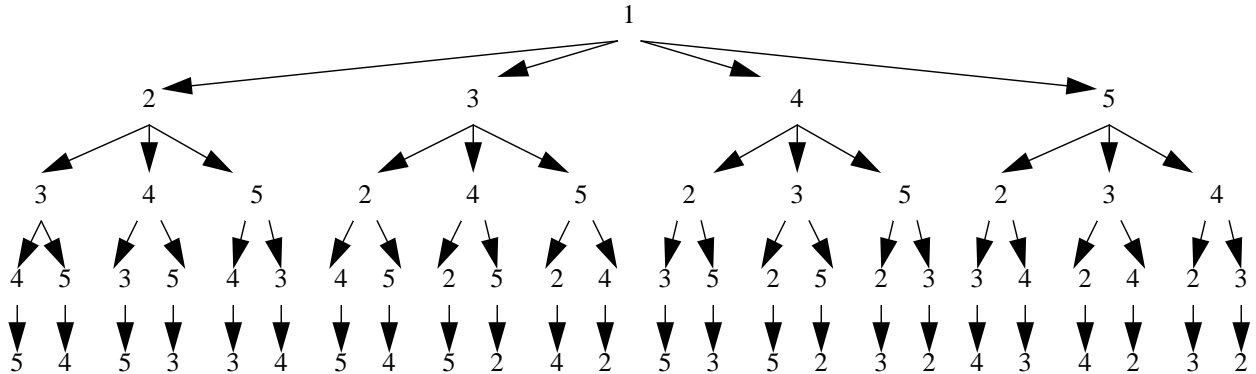


Figure 6-19: Path tree for 5 cities

objective is to make the route as short as possible. This application serves as a representative of non data-parallel applications.

There are at most $(N-1)!$ possible routes. One can construct a path tree that describes all possible routes from a particular city, and evaluate the routes by traversing all paths in the tree. Figure 6-19 shows a path tree for 5 cities. A way to solve TSP is to use a branch-and-bound technique which consists of the following steps:

1. Choose one of the cities as the starting city.
2. Find the length of a path and denote it the current *bound* of the problem.
3. For subsequent paths, if the partial length of the path is greater than the bound then there is no need to continue the traversal of any part of the tree from there on. If the length of a path is less than the bound then the bound is updated to this new value.

Figure 6-20 shows a TSP algorithm which starts from the first city and calls procedure `traverse` to recursively traverse the paths in a subtree under a second city. Hence, for the path tree in Figure 6-19, the first three paths to be traversed are $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 1$, $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 1$, and $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 5 \rightarrow 1$. For each path, procedure `traverse` compares `cbound` to the partial length of the path, and updates `cbound` if a shorter path is found.

```

TSP (ncities){
  cbound = max_integer
  1st_city = 1
  for 2nd_city = 2 to ncities
    for subtree = 1 to ncities - 2
      /* traverse a subtree under 2nd_city, reading
      and possibly updating cbound */
      traverse (1st_city, 2nd_city, subtree, cbound)
    endfor
  endfor
}

```

Figure 6-20: A sequential TSP

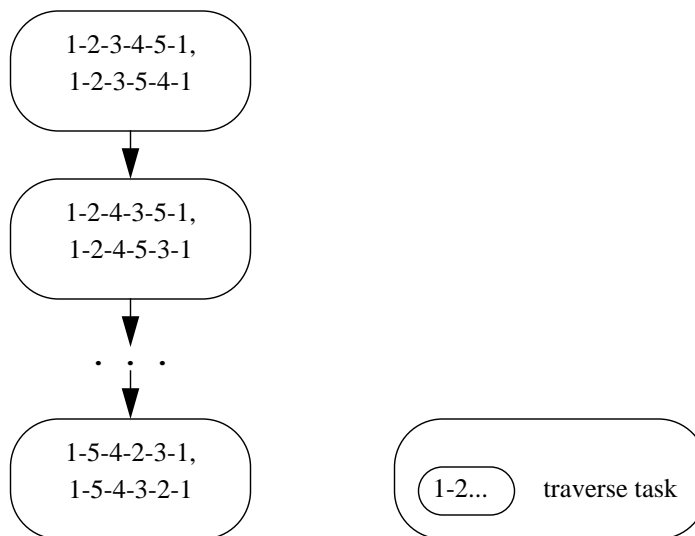


Figure 6-21: Task graph for TSP for 5 cities

For the `pTask` version of TSP, we declare procedure `traverse` as task-procedure since it contains most of the program computation. However, as the program stands, because procedure `traverse` is recursive, the parallel task program would create many tasks to traverse small subtrees. To ensure tasks having reasonable sizes, we make procedure `traverse` non-recursive by having it invoke a procedure which recursively traverses a subtree. Consequently, each `traverse` task would traverse the paths in a subtree under a second city. Figure 6-21 shows the task graph for the parallel task program for 5 cities.

```

TSP (ncities){
  cbound[1:ncities][1:ncities] = max_integer
  1st_city = 1
  for subtree = 1 to ncities - 3
    for 2nd_city = 2 to ncities
      for 3rd_city = 2 to ncities
        if (3rd_city == 2nd_city) continue
        /* traverse a subtree under 3rd_city, reading
           and updating cbound[2nd_city][3rd_city] */
        /*$ann [REGION (w(cbound[2nd_city:2nd_city]
                           [3rd_city:3rd_city]))] */
        traverse (1st_city, 2nd_city, 3rd_city, subtree,
                  cbound[2nd_city][3rd_city])
      endfor
    endfor
    /* set each element of cbound to the shortest bound */
    update_bounds (cbound)
  endfor
}

```

Figure 6-22: The pTask TSP

The task graph shows a dependence between a task and its previous task because both tasks read and write the same global variable `cbound`. Thus, in order to enforce the data-dependence constraints, pTask would execute the tasks sequentially such that the performance of the pTask-generated program on p processors would be similar to (and possibly worse than) that on a single processor. In short, the input program is coded such that there exists no task-level parallelism.

To expose task-level parallelism, we make few modifications to the program. The modifications, as highlighted in Figure 6-22, are as follows:

1. Interchanging the two loops in the original program so that the inner loop creates tasks traversing the paths in a subtree under a different second city. Hence, for path tree in Figure 6-19, the first three tasks would traverse the following

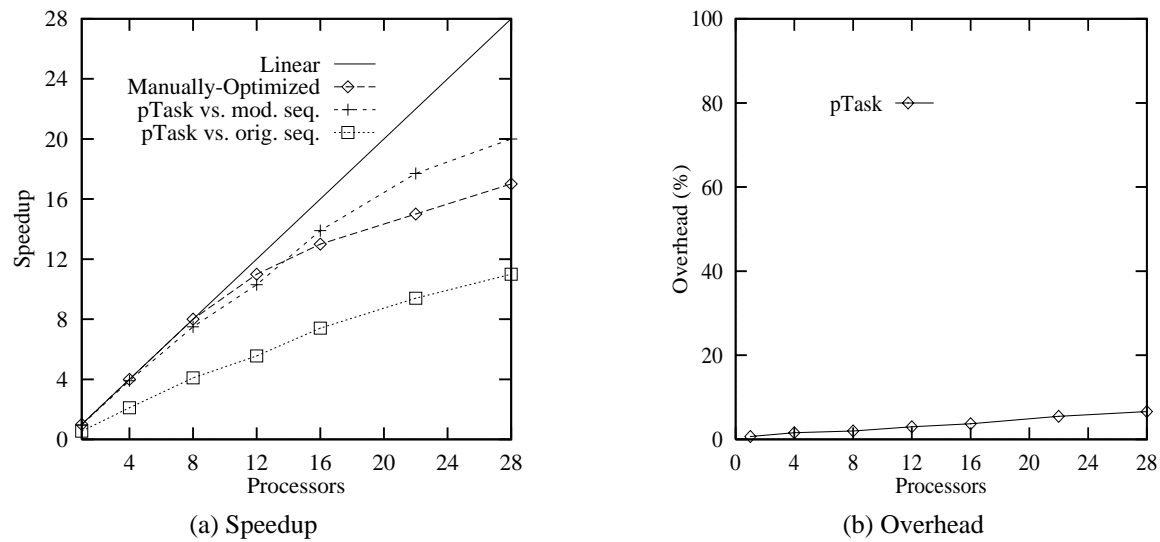


Figure 6-24: Performance of TSP (13 cities)

paths: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 1$, $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 1$; $1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 1$,
 $1 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 1$; and $1 \rightarrow 4 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 1$, $1 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 1$.

2. Introducing a third loop so that each `traversal` task would traverse the paths in a subtree under a third city, instead of under a second city. Hence, for path tree in Figure 6-19, the first three tasks would traverse the following paths: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 1$, $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 5 \rightarrow 1$, and $1 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 1$.
3. Replicating `cbound` so that each `traversal` task has a private copy of `cbound`.
4. Introducing a new task called `update_bounds` such that after a set of subtrees under all third cities are traversed, `update_bounds` would select the best among the bounds in `cbound`, and set all bounds in `cbound` to the best bound.

We also annotated the regions for `traverse` since it invokes a recursive procedure. Figure 6-23 shows the task graph of the modified program for 5 cities.

Figure 6-24 (a) shows the performance of the pTask-generated program and of a manually-optimized program for 13 cities. For pTask, the figure shows the speedup over

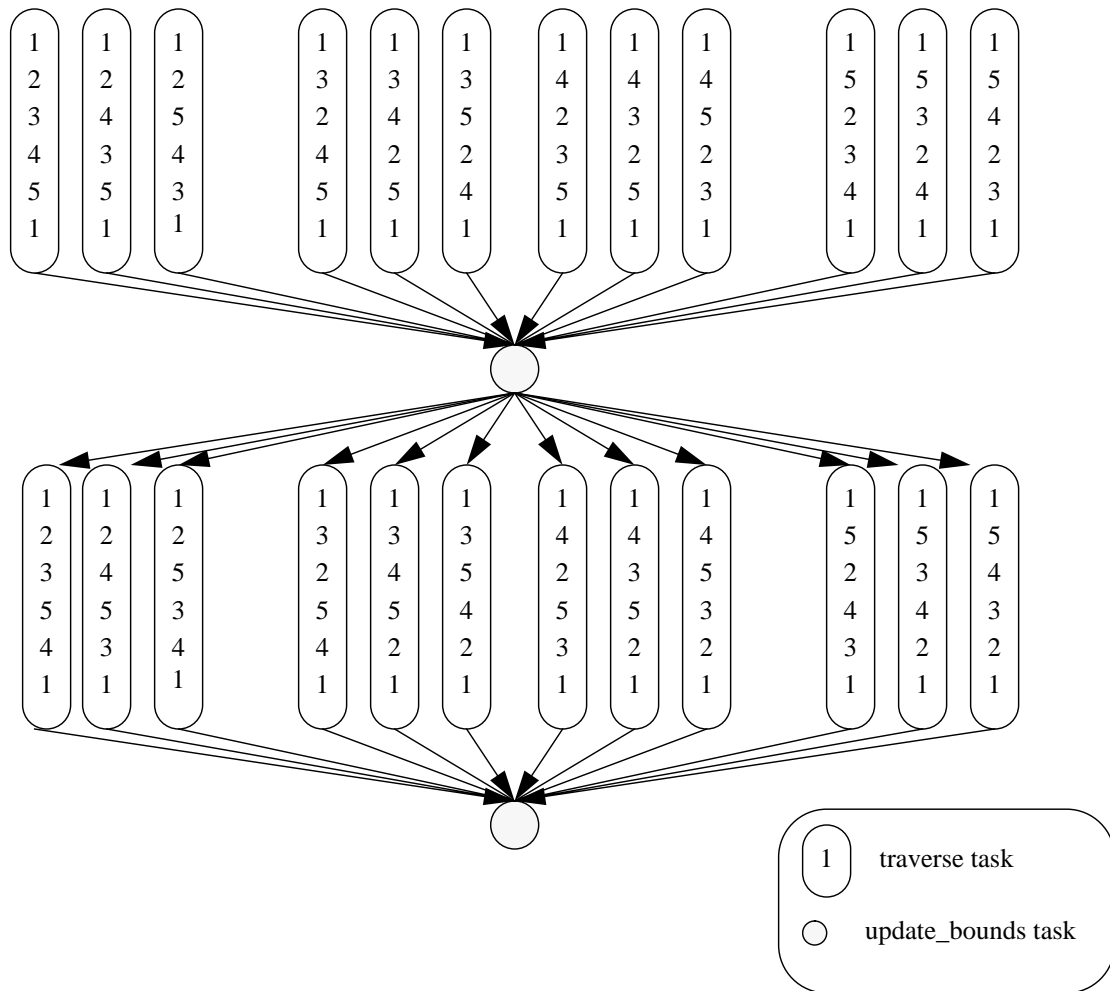


Figure 6-23: Task graph for pTask TSP for 5 cities

the original sequential program shown in Figure 6-20 (labeled as pTask vs. orig. seq.), as well as the speedup over the modified sequential program shown in Figure 6-22 (labeled as pTask vs. mod. seq.). The speedup over the original sequential program is not high because the sequential execution time of the modified program is twice as much as that of the original sequential program. As shown in the figure, the speedup over the modified sequential execution approaches the ideal speedup.

The good speedup from pTask is attributed to the fact that the run-time overhead is negligible as shown in Figure 6-24 (b). However, this speedup is not perfect due to two reasons. First, the `update_bounds` tasks in the parallel task program act as barriers and as sequential portions in the parallel execution. At each level in the task graph, a

`update_bounds` task must wait for the last `traverse` task to complete before it can proceed. The subsequent set of `traverse` tasks are executed only after the `update_bounds` task has completed. In applications having fork-join synchronization pattern such as this, better performance would be achieved if concurrently-executed tasks have similar execution times. However, this is not the case for this application due to the branch-and-bound strategy.

The manually-optimized program which was implemented using the KSR1 pthreads library, achieved good speedup over the original sequential program. This superior performance is attributed to the fact that, in this program, the parallel threads are not required to synchronize at barriers to obtain the best bound. They all share the bound, and obtain exclusive access to the bound using fine-grain synchronization primitives such as locks. Consequently, the best bound is available immediately to all parallel threads, allowing them to further prune the path tree. Thus, the good performance is attributed to the fact that the parallel execution was able to reduce the amount of computation in the sequential execution. However, it should be noted that it is not trivial to manually optimize the TSP to obtain such good performance.

In summary, with this application, we have pointed out that the performance of a `pTask` program depends on the degree of parallelism that exists in the sequential program. If the sequential program is coded such that tasks in the parallel program must be executed sequentially in order to enforce data-dependence constraints, then there would be no benefit from using `pTask`. However, we have demonstrated that with some modifications, the programmer can easily expose task-level parallelism in the program, and obtain fair performance improvement from using `pTask`. For non data-parallel applications such as TSP, we expect the performance of manually-parallelized programs using low-level fine-grain synchronization primitives to be better than that of `pTask`; however, we expect `pTask` to produce scaling performance.

Application	p=16	p=22	p=28
Radar	3%	5%	5%
Quicksort	6.6%	13%	9%
Mergesort	16%	13%	21%

Table 6-2: Improvements from schedule tasks for data locality

6.10 Scheduling Tasks for Data Locality

This section discusses the effects of scheduling tasks for data locality on the performance of the seven applications. Depending on how the data is accessed in an application, this scheduling strategy may or may not enhance the performance of the application. For instance, if the application makes only a single pass over the data, then there is no opportunity for data reuse. This is why this scheduling strategy is available as a run-time option (by setting the environment variable `PTASK_LOCALITY`).

For each task, the run-time system maintains the time the scheduler spent to determine worker-task data affinity. We define T_l as the sum of this time over all tasks, and $L_{\bar{o}}$ as the percentage of T_l to the sequential execution time T_s , i.e., $L_{\bar{o}} = T_l / T_s \times 100$. Hence, $L_{\bar{o}}$ represents the run-time overhead associated with this scheduling strategy.

Out of the seven applications, this scheduling strategy enhanced the performance of three applications; namely, radar, quicksort and mergesort. For the rest of the applications, this scheduling strategy resulted in either no performance improvement or very little performance loss since the overhead is quite low. Table 6-2 shows the additional performance improvement for these three applications at 16, 22 and 28 processors. In all cases, the overhead $L_{\bar{o}}$ is at most 1.5%.

For the radar application, the overall performance improvement is not large, but it increases as the number of processors increases. For quicksort, the improvement is largest (13%) at 22 processors, and drops to 9% at 28 processors. We suspect that for this

application the overhead from determining task-worker data locality reduces the benefit as the number of processors increases beyond 22. The best improvement is found in mergesort. As shown in the table, the improvement is largest (21%) when the number of processors is largest. This application benefits most from scheduling tasks for data locality because the whole input array is accessed by tasks in every sorting pass, giving the opportunity for tasks to reuse parts of the input array that were brought into local memories or subcaches by previous tasks.

6.11 Summary

The experiments in this chapter have demonstrated the effectiveness of pTask for a number of applications. Data-parallel and non data-parallel applications were considered. The algorithms range from the simple matrix multiplication to the rather complex branch-and-bound. pTask automatically detects task-level parallelism in most applications; for those involving recursive procedures, region annotations are required.

pTask obtained good speedup for some applications automatically. For others, the performance of the pTask-generated program depends on a number of factors. First, the task size must be large in order to amortize the overhead required to exploit task-level parallelism. Second, the application must exhibit high degree of task-level parallelism in order to obtain good performance improvement from pTask. Examples of applications having limited amount of parallelism are those having tree-structured task graphs such as mergesort and quicksort. Third, although the problem at hand may contain a high degree of task-level parallelism, the sequential application may be coded such that pTask executes tasks sequentially to enforce inter-task data dependencies. The original sequential TSP falls in this category. However, we have shown that with simple modifications, a programmer can expose the parallelism from the application and obtain

Application	Automatic	Adjust Task size	Annotate regions	Break dependencies
MM	✓			
Search	✓			
MPI	✓			
Radar	✓			
Mergesort	✓	✓		
Quicksort		✓	✓	
TSP		✓	✓	✓

Table 6-3: Summary of programming effort with pTask.

fair performance improvement. Table 6-3 summarizes the programming effort required to obtain good performance for each of the applications.

We have demonstrated the effectiveness of pTask for different classes of applications. For data-parallel applications, the speedup from pTask is close to ideal. For applications that exhibit divide-and-conquer parallelism, the speedup from pTask is comparable to that of manually parallelized programs, and we believe the obtained speedups are very close to the attainable speedups when parallelism is exploited in this fashion. For non data-parallel applications, although the performance from manually-parallelized programs may be better than that from pTask, we expect pTask to yield scaling performance.

Finally, we have shown that additional performance improvement can be obtained from scheduling tasks for data locality. This scheduling strategy is both efficient and effective. We expect the gain from this scheduling practice to increase as the size of the input data set increases, as the number of processors increases, and as the number of iterations over the data increases.

Chapter 7

Conclusions

7.1 Thesis Summary

Task-level parallelism can overcome the limitations that exist in loop-level parallelism. However, with existing systems that support task-level parallelism, a programmer must still make considerable modifications to a sequential program in order to obtain the desired parallelism. In this thesis, a system called `pTask` was designed and implemented to automatically detect and exploit task-level parallelism in sequential array-based C programs. With `pTask`, the programmer only has to select large-grain procedures to be invoked asynchronously as tasks. The compile-time module compiles the input sequential program into an equivalent parallel task program which contains code to create tasks, to describe data access regions, and to direct the synchronization of tasks. The run-time system creates, schedules, executes, and synchronizes tasks. It uses the regions supplied by the compile-time module to maintain the serial data access order of tasks, and to exploit run-time available concurrency.

The system has several desirable features. First, it accepts sequential array-based C programs as inputs. Second, by defining tasks as procedure invocations, it promotes

modular programming. Third, it supports dynamic and nested tasks. Fourth, it automatically handles both task creation and synchronization, thereby freeing the programmer from the burden of parallel programming. The system is, however, limited by the quality of available compiler analysis, by data dependences that exist in the sequential programs, and by the choice of only procedures as units of parallelism.

The experimental results indicated that pTask is both effective and efficient for different classes of application. For most applications, pTask was able to obtain good performance improvement automatically. For some applications, adjusting task grain-size is required in order to achieve good speedup. For others, the data dependences in the sequential program must be removed in order to expose parallelism. Nonetheless, the modifications are relatively simple. In most cases, the performance from pTask is comparable to that of manually parallelized programs.

7.2 Future Work

We suggest some future work for pTask in particular and for task-level parallelism in general. First, to reduce the run-time overhead, one can shift some of the dependence analysis from the run-time system to the compile-time module. The idea is for the compile-time module to perform static dependence analysis and be able to convey to the run-time system that tasks in a particular set are independent of one another. This would eliminate the need for the run-time system to determine the dependencies among these tasks at run-time. Although, one must be conservative when using static dependence analysis, being conservative here would only result the normal run-time overhead, not loss of parallelism.

A second extension is to enhance pTask to automatically select large-grain procedures to be invoked as tasks. This may be accomplished by either compile-time analysis or by the use of profiling information as feedback to the compile-time module.

A third extension is to enhance $\rho\tau$ ask to handle pointers. This may involve a study of the kind of information that is required by the run-time system in order to handle both pointer and array-based accesses, and come up with compile-time tools that can provide the required information.

A fourth extension is to implement an analysis module that exclusively provides side-effects of statements in sequential programs. Thus, instead of offering a wide set of features such as dependence analysis, code generation and manipulation, and side-effect analysis, this module would concentrate on defining and providing the kind of information that would benefit systems supporting task-level parallelism. This module should be able to accurately describe accesses to dynamic data structures such as linked-lists and trees. To determine data accesses of recursive procedures is another challenge.

Appendix A

pTask Manual

NAME

pTask - a parallel Task system

SYNOPSIS

pTask [CC options] sourcefiles ...

DESCRIPTION

pTask is a system that automatically exploits task-level parallelism in array-based C program. The system consists of a compile-time analysis module and a run-time system. pTask first compiles the sequential input program into a parallel-task program which contains constructs for the creation and synchronization of tasks. The parallel-task program is then linked with the run-time library to produce the final executable. The current version of pTask runs on the KSR1 multiprocessors.

The program must be coded in ANSI C style. A procedure can be declared to be invoked asynchronously as tasks using annotation as follows:

```

/*$ann [TASK()] */
void f1 (...) {
    ...
}

```

In the case where Sigma II fails to return the correct regions for a statement, the programmer can annotate the regions by preceding the statement with region-annotation comment which has the following syntax:

```

region-annotation ::= /*$ann annotation-string */
annotation-string ::= [REGION (access [, access])]
access ::= access-type (access-var [expr:expr] [[expr:expr]])
access-type ::= w | r
access-var ::= C-variable
expr ::= C-expression

```

Where C-variable and C-expression are valid C variables and expressions, respectively. As an example, the region of a procedure invocation is annotated as follows, assuming the procedure invocation inclusively reads from element 0 to element size of an array called buffer:

```

...
/*$ann [REGION (r (buffer[0:size]))] */
f1 (buffer, size);
...

```

The program can query the number of processors allocated by calling the global procedure `sys_nprocs_get`, which is part of the run-time library.

A simple session with pTask on the KSR1 would be:

```

% pTask -o fft -O2 fft.c -lm
% allocate_cells -A 16 fft

```

ENVIRONMENT

pTask requires the environment variables `PTASK_HOME` and `SIGMA_HOME` set to pTask and Sigma II home directories, respectively. In addition, the cfront program (CC) must be in the path.

If the environment variable `PTASK_LOCALITY` is set, the run-time system schedules tasks for data locality. This variable is not set by default.

If the environment variable `PTASK_CHECK_VAR` is set, a task is ready for execution as soon as it is created, and it incrementally checks for its regions. If this variable is not set, then a task is ready for execution only if it can access all of its regions. This variable is not set by default.

The default task stack size is 100 Kbytes. This default value can be overridden by setting the environment variable `PTASK_STKSIZE`. Since the default pthread's stack size on the KSR1 is about 15 Mbytes, this default task stack size allows a worker to suspend up to about 150 tasks.

SEE ALSO

CC (1C++)

NOTES

Users are recommended to read and be aware of Sigma II's bug list (Appendix B)

Appendix B

Sigma II Limitations

Following are problems with Sigma II:

1. Sigma II cannot analyze regions of C `while`, `do while` loops, of recursive procedures, and of data accesses using pointers. Hence, C `while` statements must be re-coded as `for` statements; calls to recursive procedures must be annotated; and data must be accessed by array indexing.
2. If the terminating condition of a C `for` loop is strictly less than (`<`) or strictly greater than (`>`) then the regions returned by Sigma II are off by one. Thus, terminating conditions of C `for` loops must be specified with equalities, e.g., less than or equal (`<=`), or greater than or equal (`>=`).
3. The first procedure in a source file must not be declared as a task-procedure. This is a problem with Sigma II's annotation system.

Bibliography

- [1] U. Banerjee. *Dependence analysis for supercomputing*. Kluwer Academic Publishers, 1988.
- [2] W. Bolosky and M. Scott. False sharing and its effect on shared memory multiprocessors. In *Proceedings of the Fourth Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 57-71, 1993.
- [3] H. Burkhardt III, S. Frank, B. Knobe, and J. Rothnie. *Overview of the KSR1 computer system*. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, MA, February 1992.
- [4] M. Chandy, I. Foster, K. Kennedy, C. Koelbel, and C. Tseng. Integrated support for task and data parallelism, *International Journal Supercomputer Applications* (To appear), 1995.
- [5] R. Chandra, A. Gupta, and J. Hennessy. COOL: a language for parallel programming. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and compilers for Parallel Computing*, pages 126-148. MIT Press, Cambridge, MA, 1990.
- [6] K. Cooper, M. Hall, R. Hood, K. Kennedy, K. McKinley, J. Mellor-Crummey, L. Torczon, and S. Warren. The ParaScope parallel programming environment. In *Proceedings of the IEEE*, Vol. 81, No. 2, pages 244-262, February 1993.
- [7] P. Dinda, T. Gross, D. O'Hallaron, E. Segall, J. Stichnoth, J. Subhlok, J. Webb, and B. Yang. The CMU task parallel program suite. Technical Report CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, March, 1994.

-
- [8] D. DiNucci, and R. Babb II. Scientific parallel processing with LGDF2. In *Proceedings of the third SIAM Conference on Parallel Processing for Scientific Computing*, pages 307-311, December 1987.
 - [9] J. Dongarra and D. Sorenson. A portable environment for developing parallel Fortran programs. In *Parallel Computing*, Vol. 5, pages 175-186, 1987.
 - [10] R. Eigenmann and W. Blume. An effectiveness study of parallelizing compiler techniques. In *International Conference on Parallel Processing*, Vol. 2, pages 17-25, 1991.
 - [11] D. Gannon, J. Lee, B. Shei, S. Sarukai, S. Narayana, N. Sundaresan, D. Atapattu, and F. Bodin. Sigma II: A tool kit for building parallelizing compilers and performance analysis system. In *Proceedings of the Programming Environments for Parallel Computing*, Edinburgh, pages 17-36, April 1992.
 - [12] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of ACM/SIGPLAN Conference on Programming Language Design and Implementation*, pages 15-29, 1991.
 - [13] T. Gross, D. O'Hallaron, and J. Subhlok. Task parallelism in a high performance Fortran framework. In *IEEE Parallel and Distributed Technology*, Vol. 2, No. 3, pages 16-26, 1994.
 - [14] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. In *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3, July 1991.
 - [15] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, November 1991.
 - [16] S. Hummel, E. Schonberg, L. Flynn. Factoring: a method for scheduling parallel loops. In *Communication of the ACM*, Vol. 35, No. 8, pages 90-100, August 1992.
 - [17] Kendall Square Research Corporation. *KSR parallel programming*, 1993.
 - [18] M. Lam and M. Rinard. Coarse-grain parallel programming in Jade. In *Proceedings of the Third ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 94-105, 1991.
 - [19] D. O'Hallaron. The Assign parallel program generator. In *Proceedings of the 6th Distributed Memory Computing Conference*, pages 178-185, 1991.

-
- [20] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. In *Communication of the ACM*, Vol. 29, No. 12, pages 1184-1201, December 1986.
- [21] C. Polychronopoulos, M. Girkar, M. Haghghat, C. Lee, B. Leung, and D. Schouten. The structure of Parafraze-2: an advanced parallelizing compiler for C and Fortran. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and compilers for Parallel Computing*, pages 423-453. MIT Press, Cambridge, MA, 1990.
- [22] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical recipes in C: the art of scientific computing*. Cambridge University Press, 1988.
- [23] H. Printz, H. Kung, T. Mummert, and P. Sherer. Automatic mapping of large signal processing systems to a parallel machine. In *Proceedings SPIE Symposium on Real-time Signal Processing*, pages 2-16, 1989.
- [24] D. Scales, M. Rinard, M. Lam, and J. Anderson. Hierarchical concurrency in Jade. In *Fourth Workshop on Languages and Compilers for Parallel Computing*, August 1991.
- [25] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13-22, 1993.
- [26] P. Suhler, J. Biswas, K. Korner, and J. Browne. TDFL: a task-level dataflow language. In *Journal of Parallel and Distributed Computing*, Vol. 9, pages 103-115, 1990.
- [27] M. Wu and D. Gajski. A programming aid for hypercube architecture. In *Journal of Supercomputing*, Vol. 2, pages 349-372, 1988.
- [28] T. Yang and A. Gerasoulis. PYRROS: static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the International Conference on Supercomputing*, pages 428-437, 1992.

