COMPILATION ISSUES FOR DISTRIBUTED SHARED MEMORY ON CLUSTERS OF SYMMETRICAL MULTIPROCESSORS

by

SUMIT ROY

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

1998

MAJOR: COMPUTER ENGINEERING

Approved by:

____

Advisor **Date**

____

____

____

Dedication

This dissertation is dedicated to the memory of K. Roy and B.B. Nag.

Acknowledgments

I would like to thank my doctoral advisor, Dr Vipin Chaudhary for his invaluable advice and guidance throughout my graduate career. His vision helped shape a substantial part of this dissertation. My committee members, Charles Nietubicz, Dr Loren Schwiebert, and Dr Cheng Zhong Xu contributed their valuable comments at various stages of this work. Dr Guy Edjlali was incredibly generous with his time, helping me with the hairier implementation issues. My former classmates, Hitesh Shah and Shyamal Prasad, provided on-line answers for all kinds of problems and doubts I had with my DSM system.

My years at Wayne State were made very pleasant by all members of the Parallel and Distributed Computing Laboratory. Ramraghavan Srinivasan in particular helped me obtain focus on my work, while Jialin Ju shared all the adventures of the PhD program with me. Swamy Punyamurtula and Subburajan Ponnuswamy helped me negotiate some of the initial hurdles with UNIX, Vikas Sinha, Padmanabhan Menon, Manish Shah, all contributed in their own inimitable ways. I am grateful to Dr Ece Yaprak for providing me with summer support. A special thanks to two really great guys, Carlo Musante and Brian Thompson, who helped me immensely with various kinds of system and network problems. A big thanks to Dr Bikash Sabata, I would not have been here without his encouragement.

Most of all I thank my lovely co-PhD seeking wife, Mahalakshmi Parameswaran, and my parents and sister, for all their moral support.

Table of Contents

List of Tables

List of Figures

1. Introduction

In recent years it has been found that single processor based computers are not able to solve increasingly large and complex scientific and engineering problems since they are rapidly reaching the limits of possible physical performance. Multiple cooperating-operating processors have instead been used to study such *Grand Challenge Problems* as Fuel combustion, Ocean modeling, Image understanding, and Rational drug design. These parallel processing systems can be classified based on various architectural features, such as the interconnection medium and topology, whether the processors are tightly knit or geographically dispersed etc. Based on the memory organization, these machines can be divided into two basic classes, those that provide a shared memory view to all processing elements, and those that present a distributed memory interface. The former design is exemplified by parallel machines like the Cray J90, the Convex Exemplar and a Silicon Graphics Inc. Power Challenge node. Distributed memory machines include early parallel computers like the CM-2 and CM-5 from Thinking Machines, the Intel iPSC and Paragon, as well as modern machines like the IBM SP2. These machines typically require a programmer to use explicit messages to send values from one processor to another. Some machines provide a shared view of physically distributed memory, such as the KSR-1 from Kendall Square Research, the Stanford DASH and the MIT Alewife.

Efficient utilization of parallel machines requires writing parallel programs or designing parallelizing compilers that can convert existing sequential programs to parallel code. A shared memory architecture allows processors to easily exchange data and synchronization primitives, hence such machines are easy to program. In addition, parallelizing systems exist for such architectures eg. SUIF [1], KAP [2], Parafrase-2 [3], and PAT [4]. However, shared memory machines do not scale well with increasing problem sizes. There are limits to the number of processing elements that can be packaged together in a machine cost-effectively. The number of processors competing for data on a shared bus for access to memory is also currently limited to about 16 – 32. Special high-speed interconnects are sometimes used to avoid this limit, as in Cray-SGI Origin 2000 nodes.

Distributed memory machines on the other hand scale quite well since they typically share only the interconnection network. Programming them is more complex; data has to be explicitly passed by the programmer or programs have to be rewritten using new languages like HPF [5], Fortran 90, Fortran D [6] and Fortran M [7] that support data layout annotations. Networks are also slower than data buses, hence it is very important to schedule communications properly and to optimize data layout to minimize network traffic. Developing general purpose compilers is thus a very complex task and currently limited to the aforementioned special languages.

A third alternative is to provide an illusion of logically shared memory over physically distributed memory, known as Distributed Shared Memory (DSM). Recent research projects with DSM have shown good performance eg. Stanford DASH [8], KSR-1 [9, 10], the Cray T3D, and TreadMarks [11]. This approach provides for an ease of programming while maintaining the scalability of distributed memory machines. Though hardware based DSM machines can potentially provide a completely transparent view of shared memory, they are currently very expensive, or exist only as experimental prototypes. User level software implementations are more portable, but require explicit handling of shared data regions. Such software DSMs typically require special initialization procedures and need a mechanism to protect access to shared memory blocks. They usually provide features available in Symmetrical Multiprocessors (SMPs), like shared data regions and synchronization primitives. However, it is far too expensive and cumbersome to rewrite all existing code even using the shared memory paradigm and some recent research has looked at compilation issues for DSMs [12, 13].

It has also been observed that vendors of traditional workstations are increasingly using multiple state-of-the art microprocessors to build high performance SMP workstations. These machines tend to dominate the server and enterprise computing applications, but have recently started appearing on the desktop. These SMPs are often connected through high speed networks or switches to form a scalable high performance computing cluster. Examples include older machines like the SGI Power Challenge using a HIPPI interconnect, as well as more modern machines like the IBM RS/6000 SP2 series with multiple PowerPC based nodes, the Convex Exemplar series of machines, the DEC AdvantageCluster 5000 with the GIGAswitch, and the Cray/SGI Origin 2000 series with the scalable Craylink network.

1.1 Automatic Parallelization Environment

The recently proposed Automatic Parallelization Environment (*APE*) [14] seeks to integrate all the above facets of parallel computing into a single environment. *APE* is a system for automatic parallelization of scientific and engineering applications on distributed memory computers. The base architecture is illustrated in Figure 1.1.

It consists of a number of processing nodes connected by a scalable network. Each node is composed of one or more commodity processors with caches, memory, and a module of network interface. *APE* takes a sequential Fortran or C program as input and produces an output program targeting either shared-memory or message-passing programming models.

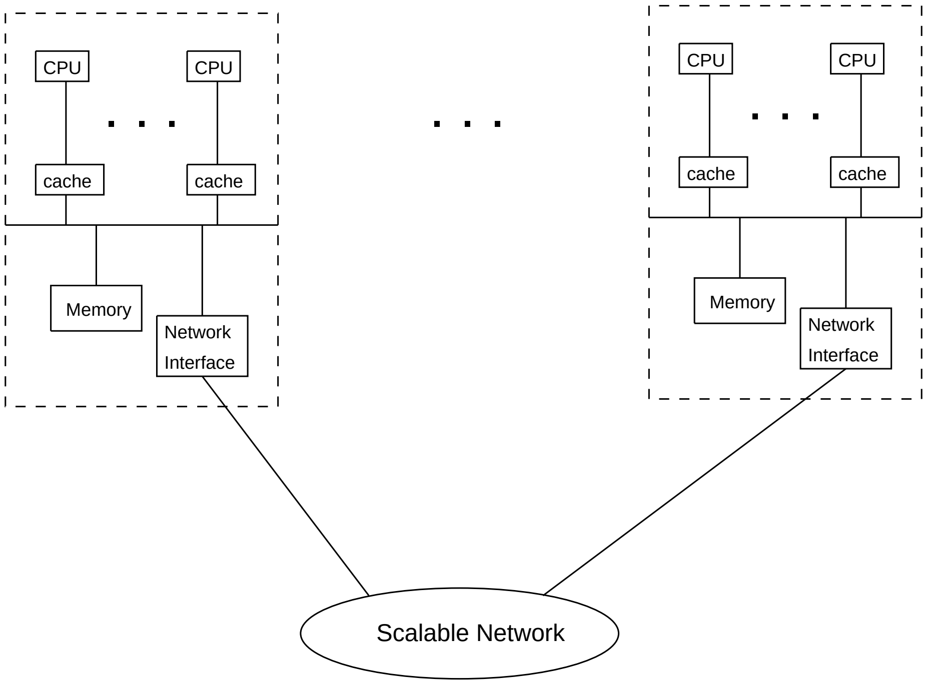

Figure 1.1: Targeted Base Machine of APE

Compile-time analysis and parallelizing techniques are integrated with scheduling support at run-time. After a survey of existing compilation tools [15], the Stanford SUIF compiler was chosen as the front-end for *APE*. The basic SUIF compiler takes a sequential FORTRAN or C program as input and generates multithreaded C code for a single SMP. Compile-time techniques for parallelization of non-uniform loops are encapsulated by enhancing the SUIF compiler, and run-time scheduling system is provided for parallelization of irregular loops, management of initial data redistribution, and communication optimization. *APE* consists of multiple modules: xSUIF, SPMD-converter, data distribution, data redistribution, and communication scheduling, which are organized as shown in Figure 1.2.

The initial version of *APE* will use a DSM as a compiler target. Since the underlying hardware layer consists of multiprocessor nodes, the computation on each node should

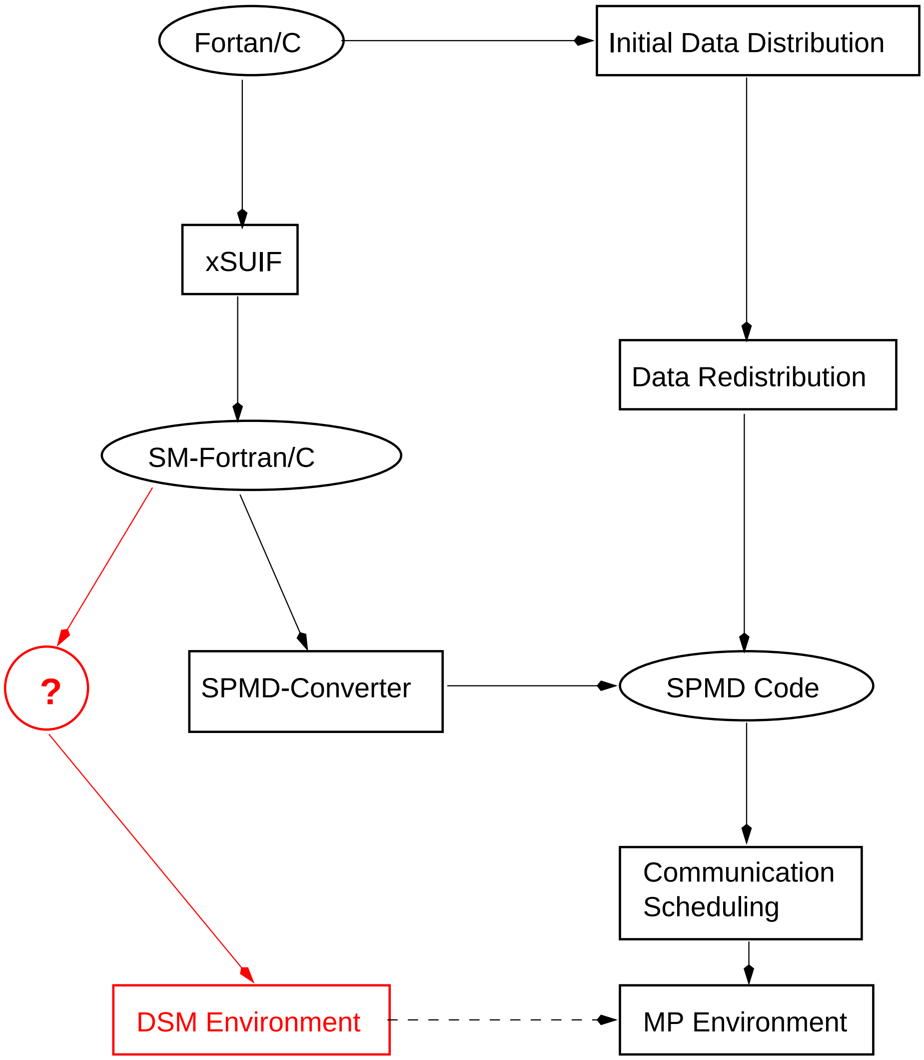

Figure 1.2: Architecture of APE

be able to make use of all the existing processors. This could be achieved by running programs with multiple threads of execution. Currently, none of the publicly available software DSMs is able to use multiple processors, though recent research , using a kernel level DSM implementation, has shown good results on this architecture [16]. In addition it has been shown that multithreaded approaches can enhance the performance of programs on general purpose distributed memory machines [17, 18, 19].

1.2 Problem Definition

The basic problem that this work solves is:

Adapt the SUIF compiler for use on a cluster of Symmetrical Multiprocessors

The features provided are:

1. The compilation target is a Distributed Shared Memory system to allow additional manual optimization if desired.
2. The system supports multiple threads of execution to make use of multiprocessor nodes.
3. The DSM is a user level software implementation and maintains portability by using a standard threading package and communication library.
4. Extensions to the SUIF compiler systems are implemented as additional passes and do not require any special language constructs.

This work allows existing sequential programs to be converted for execution on a distributed computing environment, without any input from the programmer. By setting up this basic framework, further research into parallelization environments can be done within one complete package.

1.3 Overview of Report

The following chapter presents a brief survey of parallelization tools and provides some details about the SUIF compilation system. Chapter 3 gives an overview of some of the essential features of DSMs. The next chapter shows some results with existing DSMs and explains the choice of the Quarks DSM system for further enhancement. The newly developed DSM system, *Strings* is described in detail in Chapter 5. The evaluation of *Strings* is then shown in the next chapter. Chapter 7 describes the additional passes that were required to interface the SUIF compiler to the DSM system and shows some results with benchmark programs. The last chapter provides conclusions and suggestions for future work.

2. Selection of Compiler

A preliminary phase of this work involved the selection of a suitable compiler to act as the front-end for the APE system.Various compilation and code translation systems were evaluated based on criteria such as support for FORTRAN and C as input languages, code maturity, and easy extendibility. Some of the tools were tested with sample programs, including matrix multiplication, three Computational Fluid Dynamics kernels and a complete CFD suite from the Army Research Laboratory, Aberdeen Proving Grounds.

2.1 Tools Investigated

The tools investigated include *Parafrase-2* [3], *ADAPTOR* [20], *Parallelization Assistant (PAT)* [4], *Petit* and *SUIF* [1]. The following sections describe some of the features that are present in these tools. Table 2.1 summarizes their relevant features.

Tool	Type	Input	Output		Features
Parafrase-2	Parallelizing	Fortran, C	Cedar Fortran	+	Graphical Interface
	Compiler			-	No output for C
ADAPTOR	Source	CM Fortran,	Message	+	Multi-platform
	translator	HPF	Passing Code	-	No parallelization
PAT	Interactive	Fortran	Clustered, VS or	+	Interactive Tool
	parallelizer		Parallel Fortran	-	Under development
Petit	Dependence	Fortran	Petit language	+	Interactive Browser
	analysis library			-	No code generation
SUIF	Parallelizing	Fortran, C	C for shared	+	Extendible
	Compiler		memory	-	Inefficient, unstable

Table 2.1: Comparison of Compilation Tools

2.1.1 Parafrase-2

Parafrase-2 [3] is a parallelization tool implemented as a source-to-source code restructurer. The compiler is considered multi-lingual since it uses an intermediate representation independent of the input language. Thus it could be extended by writing different preprocessors for new languages. The semantics of the input code can be recreated by converting the intermediate format back to the high level language using the appropriate post-processor.

Program transformation is achieved by using multiple passes that operate on a well-defined set of data structures. The passes are constrained to maintain the format of these data structures, hence the passes can be executed in any sensible order. The suite of passes to be applied is specified using a configuration file. Passes that are prerequisites for others are automatically executed. The parallelization and analysis passes include:

1. *bbinst*: inserts instrumentation to obtain basic block traces and profiles from a program.
2. *builddep*: builds the dependence graph based on information obtained from the depend pass. The output can be represented graphically using the draw_depend pass.
3. *callgraph*: builds a simple call-graph, however procedural parameters are ignored.
4. *codegen*: generates code for the specified language.
5. *constant*: does constant propagation and constant folding. It includes symbolic propagation.
6. *dead_code*: does simple dead code elimination.
7. *depend*: generates dependences in terms of In/Out sets for every statement.
8. *dotodoall*: examines loops in a program and checks whether they can be parallelized using the information from the *builddep* pass. The strategy adopted is to parallelize the outermost parallel loop.

9. *flow*: generates the flow graph for the program.

10. *hier*: builds a hierarchical graph using information from the flow/induction and depend passes.

11. *induction*: does symbolic analysis and recognizes all the induction expressions in the program. The pass also normalizes loop bounds when appropriate, does forward substitution, symbolic constant propagation, subscript expression normalization and dead code elimination.

12. *distribute*: does loop distribution on the program.

13. *loop_interchange*: does an interactive loop interchange for each loop nest pair that can be safely swapped.

14. *expand*: in-lines all subroutines.

15. *scalar*: performs scalar expansion.

Parafrase-2 also supports Static Program Analysis which can be used to obtain compile estimates of the costs and benefits of various program transformations. This can be useful since there is some overhead associated with parallelized code, that must be overcome before any performance gain can be realized.

A strong point of *Parafrase-2* is that it supports a graphic display of the flow graph, the dependence vectors and the function call graph. However the compiler suite generates output code only if the input code is in Fortran. The final code generated is in *Cedar Fortran*, for an experimental hierarchical-memory multiprocessor machine. Analysis is supported for C code but there is no code generation. It was found that this tool cannot recognize independence in *linearized* subscripts, such as used in the CFD kernel test suite. It is also not designed to be extended easily and the transformed code is not very portable.

2.1.2 ADAPTOR

ADAPTOR [20] consists of a source-to-source translator, which takes data parallel programs written in Fortran with array extensions, parallel loops and layout directives and converts them to parallel code with calls to a message passing library. The input languages supported include a subset of Connection Machine Fortran, as well as a subset of HPF. Recent versions of this tool have supported an increasingly larger subset of the HPF standard. The generated programs are linked with a library that provides support for a basic set of message passing routines as well as operations on sequential and distributed arrays. This library can be ported to different shared-memory, distributed-memory or shared virtual memory machines. Support for clusters of workstations is available using general message passing systems including PVM [21], P4 [22, 23], PARMACS [24] and MPI. For machines like the Intel Paragon XP/S and iPSC/860, the Meiko CS, the Connection Machine CM-5, the KSR 1 and the Alliant FX/2800, the implementation achieves greater efficiency by using the native message passing systems.

Though ADAPTOR is a powerful tool, it does not have its own parallelizing front-end, the programmer must supply annotated Fortran code to the translator. The system is also restricted to only using Fortran as an input language.

2.1.3 Parallelization Assistant(PAT)

PAT [4] is a source analysis tool providing support for interactive transformation of sequential Fortran code. It supports automation of certain aspects of Fortran code parallelization, including changing DO loops from serial to parallel by inserting proper synchronization. The resultant Fortran code can contain Clustered Fortran, VS Fortran 2.5, or AIX Parallel Fortran parallel constructs. The target Fortran code is first pre-processed to an intermediate format. The analysis tool is then invoked on the original Fortran file and the generated intermediate files. At the time of evaluation this tool was being rewritten to use an object-oriented implementation. Like ADAPTOR, it too can only support Fortran code.

2.1.4 Petit

Petit [25] is a dependence analysis and program transformation tool based on *Tiny* [26] for programs written in the Petit language, a simple array language. Fortran input code has to be first translated into this language using a preprocessor. The system features non-linear dependence analysis, scalar analyzes, including induction variable recognition and forward substitution, scalar and array expansion and privatization, as well as an interactive interface for browsing dependences and dependence relation. The tool is designed to be integrated into other compilers. Petit does not have any code generation back-end, it appears to be designed as a research aid for dependency analysis.

2.1.5 SUIF

SUIF [1] is a parallel compiler research tool. It is not designed to be a particularly efficient compiler in itself, rather it forms the basis for conducting research in parallelizing compilers. The SUIF system consists of a set of passes implemented as separate programs. Fortran or C code is first converted into the Stanford University Intermediate Format, which resembles the instruction set of a typical RISC processor. The intermediate format is a mixed level program representation, it includes high level constructs like loops, conditional statements and array accesses, as well as low-level representations. This simplifies the design of analyzers and optimizers since they deal with a language independent representation of the input program. Each pass can then perform a single analysis or transformation and save the new results in a file. Since the SUIF files use the same intermediate format, the passes can be run in any reasonable order. High level information is also preserved and various passes can communicate with each other by adding annotations to different representation units, including individual symbols, instructions, blocks and procedures. The intermediate files can be passed through various compiler stages for doing dependence analysis, loop transformations and code generation. Adding customized transformations to this tool consists of inserting a pass that reads files in the SUIF format, applies transformations and then writes out files in the same format. The SUIF toolkit has recently been adopted for the National Compiler Infrastructure Project. It is easily extendible and supports cooperative-operative research. The output of a sequence of passes is finally converted to a C program with calls to a runtime library. This runtime library assumes a master-slave model on a shared memory machine.

2.2 SUIF Evaluation

Based on the evaluation in the previous section, the *SUIF* compiler tool-kit was chosen as the front-end for APE. This section describes the structure of the compiler in more detail. The execution model of a program parallelized by SUIF, some of the extensions that were added to get SUIF to work in the local environment, and performance results with some sample code are also presented.

2.2.1 Compiler Structure

The SUIF compilation system consists of a number of passes that are executed after the front-end converts sequential Fortran or C code to the intermediate format. The default parallelizing option causes the following passes to be executed:

1. *SF2C*: Converts Fortran code to C, retaining Fortran specific information.
2. *CPP*: Preprocesses the C code to add platform specific data.
3. *SNOOT*: The C front end for SUIF, generates non-standard SUIF code.
4. *FIXFORTRAN*: Restores the Fortran specific information from *SF2C* so that subsequent passes can use it.
5. *ENTRY_TYPES*: Propagates parameter type info for Fortran entry points, required for interprocedural analysis.
6. *PORKY_PRE_DEFAULTS*: Turns non-standard SUIF produced by the front end into standard SUIF.
7. *PORKY_DEFAULTS*: Does constant folding and removes empty symbol tables.
8. *PORKY_LINE_FIX*: Removes line marks that are immediately followed by another line mark.
9. *P_CONST0*: Does simple constant propagation.
10. *PORKY_SCALARIZE*: Converts local arrays into collection of scalars if all values are known constants.
11. *PORKY_FORWARD_PROP4*: Propagates the calculation of local variables to the point of use if possible.
12. *P_CONST1*: Constant propagation on the intermediate code.
13. *NORMALIZE*: Converts all FOR loops to a standard form.
14. *P_IVAR1*: Detects induction variables.
15. *P_IVAR2*: Further detection of induction variables.

16. *P_IVAR3*: Final pass for detection of induction variables.

17. *P_KNOW_BOUNDS*: Replaces comparison of bounds in inner loops with known results, useful after multi-level induction variable detection.

18. *P_CONST2*: Another round of constant propagation.

19. *P_FOLD1*: Folds constant where possible.

20. *SCE0*: Detects privatizable scalar variables in FOR loops.

21. *REDUCTIONS*: Detects reduction operations.

22. *EMPTY_FORS*: Dismantles FOR loops with empty bodies.

23. *PREDEP*: Preprocesses symbolic coefficients for dependence analysis.

24. *P_DEAD_CODE*: Does simple dead code elimination.

25. *SKWEEL*: Optimizes for loop-level parallelism and locality.

26. *PORKY_UNCBR*: Changes Fortran call-by-reference scalar variables to copy-in, copy-out, to avoid false pointer aliasing.

27. *PGEN*: Generates code for a shared address machine.

28. *P_FOLD2*: Does constant folding.

29. *PORKY_FORWARD_PROP5*: More forward propagation of local variables.

30. *P_DEAD_CODE2*: More dead-code elimination.

31. *PORKY_LOOP_INVARIANTS2*: Moves the calculation of loop invariant expressions outside the loop bodies.

32. *PORKY_FORWARD_PROP6*: More forward propagation of local variables.

33. *PORKY_CSE2*: Does simple common sub-expression elimination.

34. *P_DEAD_CODE3*: Final round of dead-code elimination.

35. *CFORM*: Converts SUIF representations of a Fortran program to C.

36. *P_UNUSED*: Removes unused symbols and types from the symbol tables.

37. *S2C*: Converts SUIF code to ANSI/ISO C.

Since SUIF does not have a native FORTRAN front-end, it always converts FORTRAN code to C before processing it further. This can lead to an increase in code size, which makes it more difficult to maintain and verify the resultant output program. Table 2.2 shows the original lines of code, the lines of code after translation to C, and the generated lines of code, for the programs in the NAS benchmark suite [27]. *IS* is an integer sort, written in C and shows the least increase in code size. For the Fortran programs, the primary source of code size increase is the Fortran-to-C converter.

Lines of Code	Benchmark							
	BT	CG	EP	FT	IS	LU	MG	SP
Original	4016	1452	603	1481	757	4043	1789	3356
sf2c output	23399	3407	1589	6527	NA	18265	6888	18232
s2c output	14484	3546	1367	5502	894	15805	7022	17379

Table 2.2: Code Size increase for NPB2.3 serial benchmarks

Another problem with the default SUIF parallelizer is that due to the large number of passes being used, the total time taken for parallelizing an application can be quite large. Table 2.3 shows the time taken for these passes for the programs in the NAS Parallel Benchmarks 2-serial suite. For future work it would be advisable to improve the performance of the SUIF passes.

SUIF Pass	Time per Pass for each Benchmark (s)							
	BT	CG	EP	FT	IS	LU	MG	SP
SF2C	6.20	0.40	0.20	1.00	NA	3.60	1.00	4.80
CPP	0.30	0.00	0.00	0.10	0.00	0.20	0.10	0.20
SNOOT	45.10	3.50	1.60	7.70	1.00	28.60	9.30	30.20
FIXFORTRAN	120.00	3.20	1.10	6.40	NA	81.00	11.60	68.60
ENTRY_TYPES	15.90	2.10	1.10	3.80	NA	13.80	5.00	12.00
PRE_DEFAULTS	9.00	1.20	0.70	2.30	0.50	8.70	2.80	7.10
PORKY_DEFAULTS	8.50	1.10	0.60	2.20	0.40	8.00	2.60	6.60
PORKY_LINE_FIX	6.20	1.00	0.60	1.90	0.30	5.50	2.10	4.80
P_CONST0	6.90	1.10	0.60	2.20	0.30	7.00	2.60	4.80
PORKY_SCALARIZE	6.90	1.10	0.60	2.20	0.30	7.00	2.60	4.70
FORWARD_PROP4	16.50	1.60	1.00	5.50	0.40	24.80	4.60	14.50
P_CONST1	8.30	1.40	1.00	3.00	0.40	9.70	3.00	6.10
NORMALIZE	10.90	1.50	1.10	3.40	0.40	13.70	3.30	8.00
P_IVAR1	8.50	1.50	1.20	3.40	0.40	11.30	3.70	6.20
P_IVAR2	8.50	1.50	1.20	3.40	0.40	11.20	3.40	6.20
P_IVAR3	8.40	1.50	1.20	3.40	0.40	11.30	3.40	6.20
P_KNOW_BOUNDS	8.60	1.60	1.20	3.30	0.80	10.30	3.70	6.30
P_CONST2	8.30	1.40	1.00	3.00	0.40	9.80	3.10	6.10
P_FOLD1	10.20	1.60	1.20	3.50	0.50	12.20	3.50	7.80
SCE0	7.80	1.30	0.90	2.40	0.40	8.10	2.60	6.10
REDUCTIONS	8.30	1.40	1.00	2.90	0.40	9.40	3.00	5.90
EMPTY_FORS	6.40	1.10	0.80	2.10	0.40	6.80	2.30	5.30
PREDEP	8.80	1.50	1.10	3.00	0.40	10.60	3.10	6.40
P_DEAD_CODE	35.30	1.80	1.40	3.60	0.50	17.30	4.70	30.00
SKWEEL	35.10	2.60	1.30	5.30	0.50	61.00	6.80	59.60
PORKY_UNCBR	8.00	1.30	0.80	2.40	NA	8.70	2.70	5.80
PGEN	16.70	2.90	1.10	4.90	0.80	31.30	6.20	22.90
P_FOLD2	17.10	2.60	1.40	4.40	1.00	21.70	5.80	19.80
FORWARD_PROP5	20.20	2.50	1.10	5.10	0.60	31.80	6.60	27.40
P_DEAD_CODE2	73.90	4.80	2.00	5.60	0.70	27.50	33.90	77.10
LOOP_INVARIANTS2	203.70	2.50	2.30	6.40	0.70	1678.60	9.40	40.80
FORWARD_PROP6	120.00	2.50	1.20	9.70	0.60	476.50	9.10	57.00
PORKY_CSE2	385.30	7.40	3.10	24.20	1.40	1088.10	31.80	232.10
P_DEAD_CODE3	145.30	4.90	1.80	7.70	0.70	142.40	49.50	126.10
CFORM	49.10	2.20	0.80	5.70	0.00	44.90	6.30	46.50
P_UNUSED	11.10	2.00	0.80	3.30	0.60	12.90	4.30	12.70
S2C	55.40	7.80	2.60	13.80	1.90	67.50	18.90	58.00
Total Time	1520.70	81.40	42.70	174.20	18.50	4022.80	278.40	1050.70

Table 2.3: Compilation Time for NPB2.3 serial benchmarks

Benchmark	Results
ADM	invalid result
SPICE	skweel dumps core
QCD	porky runs out of memory
MDG	validates, 125 $\times$ sequential time
TRACK	invalid result
BDNA	P_FOLD2 dumps core
OCEAN	cannot handle EQUIVALENCE, code dumps core
DYFESM	pgen dumps core, problem with EQUIVALENCE
MG3D	invalid result
ARC2D	invalid result
FLO52	validates, 15 $\times$ sequential time
TRFD	invalid result
SPEC77	validates

Table 2.4: SUIF on PERFECT benchmarks

Finally, a major problem with SUIF is that the compiler sometimes fails while executing a pass, or even produces incorrect code. Table 2.4 shows the result of attempting to parallelize the programs in the PERFECT benchmark suite with SUIF. Though the results for MDG, FLO52 and SPEC77 are validated, the execution time is worse than the original serial time.

2.2.2 Execution Model

The final stage of the SUIF parallelizing compiler converts the intermediate format to C program, with calls to a runtime library. Figure 2.1 shows the FORTRAN version of a sequential matrix multiplication program. Part of the output from the SUIF compilation system is shown in Figure 2.2.

The compiler converts each parallel loop into a function, eg. `_MAIN___1_func`. The *main()* routine or entry point is provided in the runtime library, and is responsible for setting up internal data structures and for creating slave tasks. These slave tasks then wait on an

```
      program matrix
c     FORTRAN version of MATRIX multiplication
c
      real A(1024, 1024),B(1024,1024),C(1024,1024)
      integer i,j,k
c
c     initialize arrays

      do 10 i=1, 1024
         do 20 j=1, 1024
            A(i, j) = 1
            B(i, j) = 1
            C(i, j) = 0
 20      continue
 10   continue

c
c     Multiplication
      do 40 i=1, 1024
         do 50 j=1, 1024
            do 60 k=1, 1024
               C(i, j) = C(i, j) + A(i, k) * B(k, j)
 60         continue
 50      continue
 40   continue
c     print results
c      do 70 i=1, 1024
c         print *, C(i,i)
c 70   continue
c
      stop
      end
```

Figure 2.1: Original Matrix Multiplication code

internal barrier. The main thread executes sequential parts of the program. When a parallel region is reached, the main thread indicates the particular function to be executed in the shared region `_suif_aligned_task_f` and passes the arguments in the region pointed to by `_suif_aligned_args`. It then releases the barrier and all the threads execute the parallel function on their section of data. At the end of the parallel function, the slave threads again wait at the barrier. The master then proceeds with the next sequential code until another parallel region is reached.

```
extern int *volatile _suif_aligned_my_nprocs;
extern void *volatile _suif_aligned_args;
extern void (**volatile _suif_aligned_task_f)(int);
struct _BLDR_struct_000 { ... };
struct _BLDR_struct_001 { int *_field_0;
  float ((*_field_1)[1024])[1024];
  float ((*_field_2)[1024])[1024];
  float ((*_field_3)[1024])[1024]; };

void _MAIN___0_func(int _my_id){ ... }

void _MAIN___1_func(int _my_id)
{
  int *i, _my_nprocs, j, __priv_i0, k, j_tile4, k_tile5;
  float ((*c)[1024])[1024], ((*a)[1024])[1024], ((*b)[1024])[1024];
  struct _BLDR_struct_001 *_my_struct_ptr;

  _my_struct_ptr = _suif_aligned_args;
  i = _my_struct_ptr->_field_0;
  c = _my_struct_ptr->_field_1;
  a = _my_struct_ptr->_field_2;
  b = _my_struct_ptr->_field_3;
  _my_nprocs = *_suif_aligned_my_nprocs;

  for (j_tile4 = 1; j_tile4 <= 1024; j_tile4 += 70)
    for (k_tile5 = 1; k_tile5 <= 1024; k_tile5 += 70){
      for (__priv_i0 = max(1024 * _my_id / _my_nprocs + 1, 1);
           __priv_i0 <= min(1024 * (_my_id + 1) / _my_nprocs, 1024);
           __priv_i0++){
        for (j = max(1, j_tile4); j <= min(1024, 69 + j_tile4); j++)
          for (k = max(1, k_tile5); k <= min(1024, 69 + k_tile5); k++)
                c[j - 1][__priv_i0 - 1]
              += a[k - 1][__priv_i0 - 1] * b[j - 1][k - 1];
      }
    }
}

int MAIN__()
{
  float (a[1024])[1024], (b[1024])[1024], (c[1024])[1024];
  struct _BLDR_struct_001 *_MAIN___1_struct_ptr;
  ...
  suif_named_doall(_MAIN___0_func, "_MAIN___0_func", 0, 1024, 1);

  *_suif_aligned_task_f = _MAIN___1_func;
  _MAIN___1_struct_ptr = _suif_aligned_args;
  _MAIN___1_struct_ptr->_field_0 = &i;
  _MAIN___1_struct_ptr->_field_1 = (float ((*)[1024])[1024])c;
  _MAIN___1_struct_ptr->_field_2 = (float ((*)[1024])[1024])a;
  _MAIN___1_struct_ptr->_field_3 = (float ((*)[1024])[1024])b;
  suif_named_doall(_MAIN___1_func, "_MAIN___1_func", 1, 1024, 1);
}
```

Figure 2.2: SUIF output of Matrix Multiplication (edited for clarity)

2.2.3 Extensions Required

SUIF 1.0.1 was initially evaluated. This version only had runtime library support for the Stanford DASH, and the KSR-1. All other architectures were limited to using a sequential version of the runtime. The library was ported to a number of architectures for evaluation, including a SPARCserver 630MP with 4 processors, using Solaris kernel threads, a Cray J916 with 8 processors, using light-weight *tfork* calls, and a SGI PowerChallenge with 12 processors, using a 64 bit version of the runtime.

Due to limitations in the Fortran front-end, it was also necessary to write a translator to convert non-standard Fortran construct, like the “DO”- “ENDDO” pair, to traditional Fortran 77 code (“DO line-number” - “line-number CONTINUE”).

2.2.4 Performance Results

After porting *SUIF* to the SPARCserver 630MP, various timing measurements were performed. The codes used were three test routines from the Army Research Lab, and a matrix multiplication program. The test programs form part of a Computational Fluid Dynamics (CFD) kernel. The difference in these programs is in the declaration of the main data structure.

Different problem sizes were used to investigate scaling issues in the runtime, as well as the quality of the generated code. Speed-ups were measured by changing the number of concurrent threads. On a multi-processor machine, each thread would be scheduled on an independent processor if possible. The parallelized CFD kernel code provides fairly good speed-up for the original problem size of 500×5000. For reduced problem sizes the startup cost dominates, at larger sizes the performance degrades due to a memory bottleneck on the SPARCserver, as can be seen in Figures 2.3, 2.4, and 2.5. When the number of threads is increased substantially beyond the physical number of processors on the machine, the speed-up starts to decreases, since time is spent in the kernel attempting to schedule the excess threads. In this case the machine had four processors, and it can be seen that the performance dropped off if more than six threads were created.

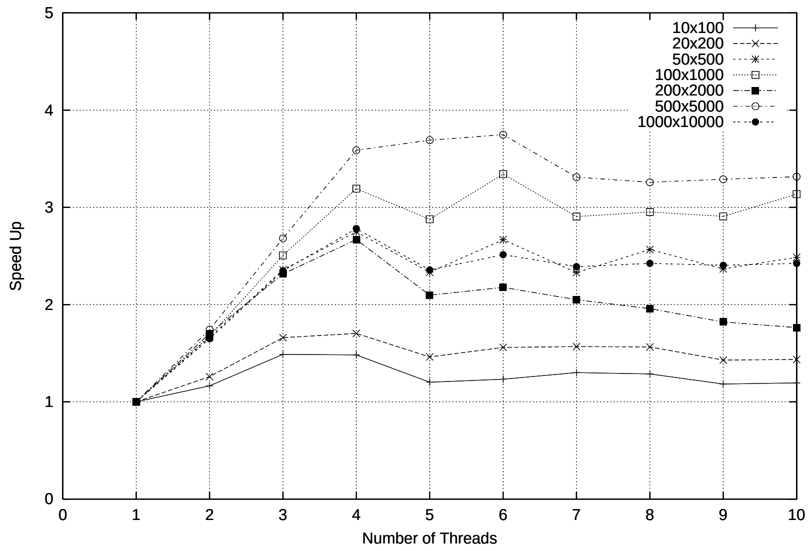

Figure 2.3: Speedups for CFD kernel version 1
Array: `dimension y1(2505501),y2(2505501),y3(2505501)`

The matrix multiplication program was used as a basis for a more detailed performance analysis. The matrix multiplication program consisted of two nested loops, the first for initializing the matrices, the second for multiplying them. The first loop was doubly nested, and the second loop was triply nested. Execution times were obtained for the following versions of the program:

1. *Fortran:* A sequential Fortran version using the native compiler.

2. *CC:* A sequential C version compiled with the native compiler.

3. *SUIF:* A Fortran version passed through the SUIF compiler with calls to the multi-

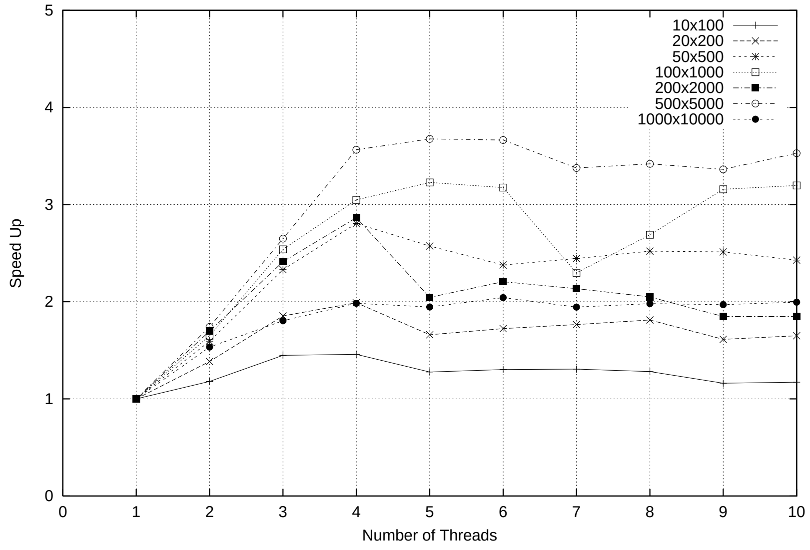

Figure 2.4: Speedups for CFD kernel version 2
Array: `dimension y(3,2505501)`

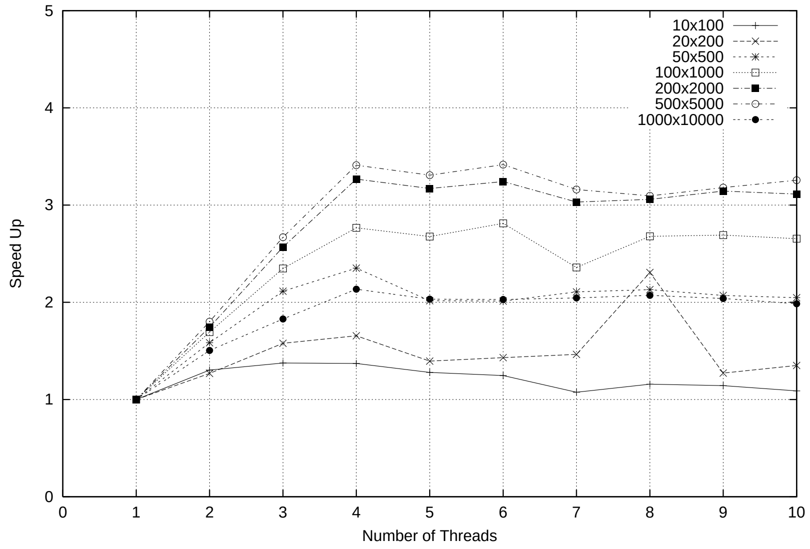

Figure 2.5: Speedups for CFD kernel version 3
Array: `dimension y1(5001,501),y2(5001,501),y3(5001,501)`

threaded runtime.

4. *Hand:* A hand-coded multithreaded C version.

Array Size	Execution Time (s)			
	F77	CC	SUIF	Hand
50×50	0.223	0.177	0.222	0.200
100×100	0.556	0.780	0.442	0.449
200×200	3.752	6.134	2.241	1.954
500×500	81.718	116.909	41.502	38.515
1000×1000	869.368	1231.317	466.334	482.201

Table 2.5: Execution Time for Floating Point Matrix Multiplication

All compilations were done with maximum optimization setting for the native compiler. The first version of the matrix multiplication program use arrays of *floating point* numbers, and the results for varying array sizes are shown in Table 2.5. As can be seen from the data, there is a performance penalty ranging from 50 % to approximately 100 % when using *C* for floating point operations as compared to *Fortran* on the SPARCserver 630 MP. Since SUIF generates its final output in the form of a C program, this penalty should be considered while analyzing it overall performance.

Array Size	Execution Time (s)			
	F77	CC	SUIF	Hand
50×50	0.266	0.244	0.289	0.239
100×100	1.566	1.477	0.733	0.626
200×200	12.625	12.416	3.766	4.329
500×500	251.399	253.193	74.828	71.689
1000×1000	2434.053	2468.806	717.615	714.940

Table 2.6: Execution Time for Integer Matrix Multiplication

Table 2.6 shows the execution times for a version of the code that uses *integer* instead of *floating point* numbers. In this case the results are identical for Fortran and C. On the SPARC 630MP, integers and floating point values are both 32 bits wide, hence both versions of the program have the same memory requirement for a fixed matrix size. However, it can be seen that the overall performance for the integer version is worse than the floating point version by a factor of almost 3, which indicates that the machine has a very efficient floating point unit.

Figures 2.6 and 2.7 show the relative performance for varying number of concurrent threads, for the floating point and integer version respectively. The results of the hand coded parallel program and the *SUIF* version show that there is very little overhead due to calls to the *SUIF* runtime library. Allowing for a statistical variation in the execution times, the results lie within 10 % of their mean. Hence, one would expect very little performance gain by manually embedding parallel routines in the target code.

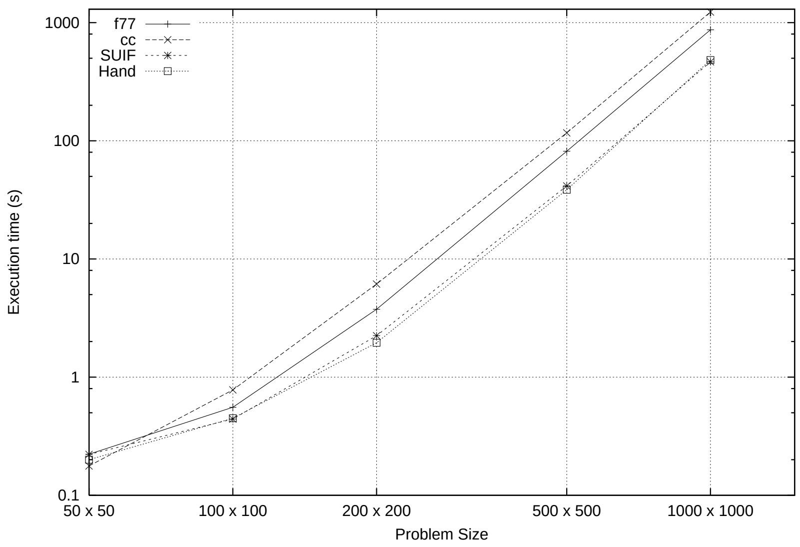

Figure 2.6: Execution Time for Floating Point Matrix Multiplication

Figure 2.8 illustrates the scaling of the matrix multiplication, run on one 12 processor node of a SGI PowerChallenge Array (PCA). The parallelized code shows near linear speedup in most cases. For a problem size of 100×100, the startup costs dominate and there is not much performance improvement when adding more threads. There is a superlinear speedup of 4 when using 3 processors, this is likely due to cache effects at this small problem size. Since the machine was not in a dedicated mode, there is some anomaly induced due to scheduling conflicts when using 5 – 7 processors.

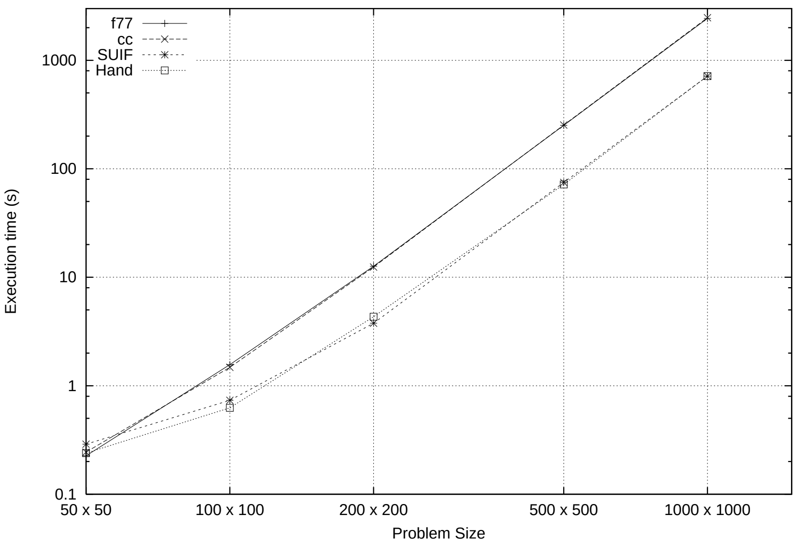

Figure 2.7: Execution Time for Integer Matrix Multiplication

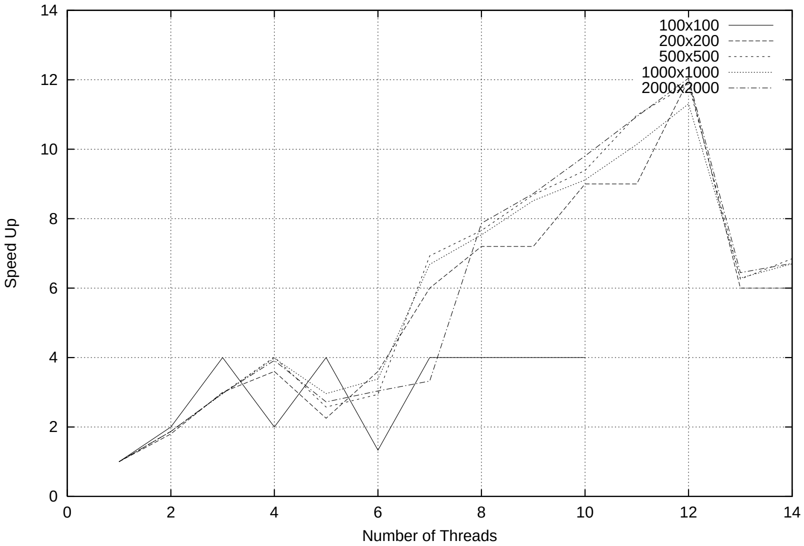

Figure 2.8: Speedups for Matrix Multiplication on SGI PCA Node

3. Distributed Shared Memory Features

A Distributed Shared Memory system implements a shared-memory abstraction on top of a message passing distributed memory system. It may be possible to instead maintain a globally explicit data space, but that is not convenient, since every reference could potentially be remote and would therefore have to be implemented as a function call. This implies very poor performance for accesses in loops. Another alternative would be to use an object level based data sharing, with remote operations for remote references, but that is more suitable for client-server type distributed computing. A shared virtual memory system layered off an existing virtual memory subsystem makes all remote accesses standard local memory references, once the memory page has been swapped in. Thus a DSM has a potential for improving performance since communication costs can be amortized over larger data blocks, and it may be possible to overlap communication with computation [28].

3.1 Main Design Issues

The basic requirement of a DSM system is to transform accesses to shared memory region into interprocess communication. Earlier researchers have investigated the design issues that have to be addressed when developing a DSM system [29]. The main points are listed next.

3.1.1 Data Location and Access

Data needs to be found and retrieved for the application. The data distribution could be static, which can lead to a bottleneck if the initial data distribution is poor. The alternate solution is to use dynamic data distribution, with replication and migration. In this case tracking the data may become difficult.

3.1.2 Coherence Protocol

A coherence protocol is required when replicating data, so that all participating processes in the distributed shared memory system see the same view of memory. The actual implementation of the coherence strategy can be classified based on the method used for *page synchronization* and the *page ownership*. Write broadcast based synchronization is more expensive since special hardware support would be required to sequence multiple writes at a time. Hence write invalidate based synchronization is more popular. Page ownership can be fixed or dynamic. Dynamic page ownership can be further classified as centralized or distributed. The distributed mechanism could use a fixed or dynamic distribution of ownership data. It has been shown that write invalidate based protocols perform well in practice [28]. Write update based protocols can be inefficient if there is high network latency.

3.1.3 Replacement Strategy

Since there is only finite space for caching remote data, some policy on swapping out stale pages has to be implemented. The algorithm has to ensure that there is always at least one valid copy of a data region in the DSM system.

3.1.4 False Sharing

When multiple nodes want to update the same data item, it may ping-pong between the competing nodes. This is termed as thrashing. DSMs with large coherence units also suffer from the additional problem of false sharing. In this scenario, multiple processors try to write to different data items in the same shared unit. Depending on the particular policy in use, this can lead to excessive travel of the page between the affected nodes.

3.2 DSM Algorithms

The DSM designer can solve the above problems by choosing appropriate algorithms, implementation levels and consistency models for the system [30]. Two basic problems need to be solved:

- distribution of shared data across the nodes to reduce communication costs and access latency.
- preserving data coherency so that the program executes correctly from the users point of view.

A frequently used strategy that allows simultaneous access to a shared data block is *replication*. *Migration* allows data blocks to move across nodes and thus reduces the communication overhead. Based on the implementation of these strategies, DSM algorithms can be classified as follows [31]:

3.2.1 Central Server

This is a Single Reader/Single Writer (SRSW) based algorithm, where all shared data is read from a single server. The single server could form a communication and computation bottleneck if it has to respond to many simultaneous requests. The algorithm could be enhanced by statically distributing the data across multiple fixed servers. Neither *replication* nor *migration* is allowed by this algorithm. Hence multiple reads cannot be allowed to the same data block. The Central Server algorithm is easy to implement and shows good performance when the ratio of writes to reads is high and when accesses to shared data are infrequent.

3.2.2 Migration

This modified SRSW based algorithm always lets the data migrate to the requesting client. A central server can be contacted to determine the location of the data block. *Thrashing* can result wherein two or more clients try to access the same data block, since it ping-pongs between them. Another problem is *false sharing*, where more than one client may try to access different data elements residing in the same data block. When the fault rate is low and there is high locality of reference, this algorithm performs well. Thrashing can be avoided by locking blocks of data, though this violates the shared memory view.

3.2.3 Read Replication

Multiple Reader/Single Writer (MRSW) accesses are allowed in this scheme. Remotely held data must be invalidated or updated by a writing process to ensure consistency. This algorithm performs well for interleaved access to regions since it avoids *false sharing*, as opposed to the migration algorithm.

3.2.4 Full Replication

This is a Multiple Reader/Multiple Writer (MRMW) type implementation, and requires some sort of sequencing so that write requests can be reliably multi-cast to all copies. This algorithm is expensive to implement for large replication and a large number of servers, though it has the potential to outperform all the others.

3.3 Implementation Level

The DSM implementation level defines the mechanism, performance and cost associated with looking up data, accessing non-local data and invalidating remote data on writes to preserve coherence. Based on the design, the approaches can be identified as hardware implementations, operating system or library level and compiler or language based [29].

3.3.1 Hardware Implementation

Hardware implementations essentially extend traditional caching techniques. They can provide good performance but are inherently non-portable. The physical unit of replication and coherence is usually a non-structured collection of bytes. The size is typically some multiple of the cache line size. This tends to reduce the effects of false sharing when compared to page based schemes. Prominent examples of hardware based DSMs are listed next.

Dash: [8] An implementation using modified SGI Iris workstations, connected in a mesh.

Memnet: [32] A 200 Mbps token ring based system that broadcasts invalidates and read requests.

SCI: [33] An interface standard (IEEE P1596) that defines a scalable directory protocol and large shared address space.

KSR1: [9] A hierarchical ring-based organization of clusters that implements an Allcache engine which transparently routes requests through the hierarchy.

3.3.2 Operating System/Library

Some DSM implementations are designed for special purpose operating systems or kernels, and are not very portable. Examples include:

Choices: [34] DSM incorporated into a hierarchical object-oriented operating system.

Clouds: [35] An object oriented distributed operating system with object migration.

Ivy: [28] Software on Apollo workstations connected in a ring and running a modified Aegis operating system.

Mirage: [36] Built on top of the Locus distributed operating system and introduced a time-to-live parameter for shared pages.

Plus: [37] A hybrid hardware software system built on top of the Plus kernel for the Caltech mesh.

Shared Data-Object Model: [38] Implemented on top of the Amoeba distributed system.

Shiva: [28] Ivy like system on the Intel iPSC/2 operating system.

Purely user level software implementations under UNIX are discussed in detail in the next chapter.

3.3.3 Compiler/Language based

Languages exist that support distributed shared memory in terms of objects or by allowing layout directives. Such systems include Linda, Orca, HPF and Fortran 90. This approach is not very suitable since existing code would have to be rewritten to make use of such systems.

3.4 Memory Models

The memory consistency model defines the order in which memory references issued by a processor are seen by other processors [39]. There are two main types of memory consistency models, strong consistency models and weak consistency models.

3.4.1 Strong Consistency

Strong consistency models treat all memory accesses uniformly. Examples of strong consistency modes include:

Strict Consistency This model requires that a read operation return the value of the most recently written value. In the absence of a globally synchronized clock, the idea of “most recently” is ambiguous and some DSM systems allow weaker forms of memory coherence.

Sequential Consistency This requires that all processors observe the sequence of reads and writes as some interleaving of accesses from individual processes. Reads cannot bypass writes in this model [40].

Processor Consistency This model requires that writes by a single processor are always seen in the order that they are issued, but memory operation by different processors can be observed in any order.

3.4.2 Weak Consistency

Weak consistency models distinguish between ordinary and synchronization accesses. This is enforced by allowing ordinary accesses to complete only when all previous synchronization accesses have been completed and vice-versa.

Release Consistency This model introduces two synchronization operators: acquires and releases. Processor consistency is guaranteed for these operators and ordinary accesses are allowed to proceed between them. This model is used in the DASH machine and the Munin software DSM system [41].

Lazy Release Consistency This model enhances release consistency by propagating modifications only on the next relevant acquire. Modifications are thus combined with the acquire of the associated lock. This system is used in TreadMarks [42].

Entry Consistency A synchronization variable is associated with every shared data object. The modifications to the object are propagated only at the next acquire of the synchronization variable associated with it. The latency is thus reduced, since only a subset of the shared data is sent at every acquire. This system is used in Midway [43].

Scope Consistency The execution is divided into local and global scopes. Global scopes are limited by barriers and cause all pages to be updated. Local scopes are delimited by locks and only pages associated with the lock are updated [44].

4. Survey of Related Work

Pure user level software implementations of DSM have recently been shown to provide good performance at low costs. These often make use of the virtual memory management mechanism provided in modern operating systems, in which case they can be considered as hybrid hardware/software systems.

4.1 Details of Software DSMs

4.1.1 IVY

IVY [28] was one of the first implementations of a software DSM, built as a set of user-level modules on a ring of Apollo Domain workstations and a modified Aegis operating system. Different consistency maintenance mechanisms were tested, including:

monitor-like centralized manager: The manager resides on a single processor which maintains information on the *owner*, the *copy-set* and the *locks* for each page. The *owner* is the last node to write to a particular page and is responsible for sending a copy of this page when it receives a read request from the manager. The *copy-set* is the list of nodes that contain a copy of a particular page. The *lock* is used to protect access to a page by the server (if multiple nodes contact it) or by the client (if multiple processes fault on the same page). A maximum of two messages is required to locate a page.

improved centralized manager: The *copy-set* is handled by the *owner*, while the manager process only takes care of the current *owner*.

fixed distributed manager algorithm: Each processor manages a set of pages based on some hashing function. It outperforms the centralized algorithm when the page-fault rate is high, though it is difficult to find a hashing function that works well for all applications.

broadcast distributed manager: Each process maintains the *owner* of the page in its page table. Request to page access are broadcast to all processes. The performance of such a scheme is poor for more than four processors, since every broadcast has to be handled by each process.

dynamic distributed manager: Each processor maintains the probable owner in its page table and the requests are forwarded to the actual *owner*. The algorithm needs $O(N + K \log N)$ messages to find a page K times in a system with N nodes.

fewer broadcasts: This improves on the previous algorithm by broadcasting the page location every M page faults. This improves performance since it is easy to find a page soon after the location has been broadcast.

distribution of copy-sets: These are used to invalidate the page on a write request. By using a tree like algorithm, one can get a good performance on networks that do not have a broadcast mechanism.

The system was tested on a Jacobi program for solving 3D PDEs, parallel sort, matrix multiplication and a parallel dot-product.

4.1.2 Munin

This system introduced the concepts of type-specific coherence mechanisms and the release consistency model [41]. Each shared variable is kept consistent with a protocol suited to its expected or observed access pattern. A study of shared memory programs [45] showed that the most important distinction between data access patterns is between those best handled via an invalidate protocol and those handled best via an update protocol. Also synchronization variables are inherently different when compared to ordinary data variables and need to be treated accordingly. The characteristic access pattern does not change frequently through the execution, hence static access type assignment is sufficient. The four consistency protocols supported by Munin include *conventional, read-only, migratory,* and *write-shared*. Separate synchronization protocols are provided for locks, barriers and condition variables.

A programmer is required to add annotations to the declaration of shared variables which describe the sharing pattern and hence the protocol to be used. All variables on a page use the same protocol, hence separate variables have to be allocated on separate pages. It is claimed that the memory overhead due to this is low for the applications studied. A 16 processor prototype was implemented on an Ethernet network of SUN-3 workstations, using the *V* operating system. A penalty of at most 10 % was encountered when compared to optimal hand-coded programs.

4.1.3 Midway

Midway introduced a new model of memory consistency called *entry consistency* [43]. This exploits relationships between a synchronization primitive and the data that it protects. The consistency protocol is implemented in software and only requires a message passing interface from the underlying hardware. Program concurrency is expressed using Mach threads and controlled using locks and barriers. Shared data is associated with at least one synchronization object and multiple consistency protocols are supported.

Close-to-linear speed-ups were obtained on an ATM connected cluster of MIPS R3000-based DEC stations.

4.1.4 TreadMarks

TreadMarks is implemented as a user-level library on top of UNIX [11]. An important feature introduced in this system is a new consistency system, *Lazy Release Consistency*. Updates are delayed till a synchronization primitive is encountered. The consistency is enforced at acquire time, not release time (unlike *eager release consistency* as in Munin). This system also supports a Multiple Reader and Multiple Writer Protocol by maintaining *diffs* of local changes with respect to the original master copy. These *diffs* are used to update all copies at synchronization time. Performance was evaluated on a network of eight DECstation-5000/240s, running Ultrix V4.3 over a 100 Mbps ATM network. Near linear speed-ups were obtained on a Mixed integer programming problem. A Genetic linkage analysis problem showed good results for data sets with long sequential running times.

Recent variants of the TreadMarks system include CVM [46] and Cashmere-2L [47].

4.1.5 Quarks

This is another runtime library system that can be linked with an application program [48]. A central server is used to initially allocate shared regions and global barriers and locks. It does not pose a bottleneck since it is involved only at program initiation and termination, while the runtime library handles all other aspects. *Quarks* implements a write invalidate protocol as in *IVY* based on sequential consistency and a write update protocol proposed in *Munin*, based on release consistency. A distinguishing feature is the support for global threads built on top of user level *Cthreads*. A reliable communication system is built on top of the *UDP* protocol.

Quarks is available on SunOS 4.1, Irix and HP-UX, as well as on the DEC Alpha. As part of the experiments of this theses, *Quarks* was also ported to Solaris 2.x and AIX 3.2.5.

The distribution contains sample programs including matrix multiplication and successive over relaxation.

4.1.6 SAM

SAM [49] provides a shared name space and dynamic caching of remotely accessed data by tying synchronization to data accesses. Two types of data are provided, *values* and *accumulators*. A process writes to a *value* and thus shares a producer/consumer relationship with all readers of that value. An *accumulator* on the other hand can be modified multiple times, all modifications are guaranteed to be mutually exclusive, though in any order. Parallel applications can be speeded up by allowing *chaotic* access to data ie. using slightly out-of-date data. Memory management primitives are provided to control the number of *values* in the system. Low-level communication control is achieved by allowing asynchronous data fetches and explicit sends to remote processors.

SAM has been implemented on the CM-5 using Active Messages, the Intel iPSC/860 and Paragon, the IBM SP1, as well as a network of workstations using PVM. Performance was evaluated on Block Cholesky factorization, Barnes-Hut *n*-body algorithm and computation of the Grobner basis of a set of polynomials. The design of SAM allows some Message Passing functionality within a shared memory system by explicitly "pushing" data and allowing asynchronous fetches.

4.1.7 C Region Library

The C Region Library (CRL) is an all-software DSM system featuring system and language independence, portability and efficiency [50]. The programmer defines arbitrarily sized regions and delimits access using annotations. This avoids the problem of false sharing as well as inefficiency when using too small a coherence unit, which are common in page-based and hardware based systems respectively. CRL also uses main-memory as a large secondary cache to offset the latencies of communications operations. Operations on regions are considered indivisible units, hence CRL implements sequential consistency for read and writes. Individual loads and stores are handled using entry or release consistency. A fixed-home, directory-based invalidate protocol is implemented without the “three-party optimization” (as in DASH).

Current implementations exist for the CM-5, the MIT Alewife as well as a PVM version for a network of workstations. Performance results are shown for Blocked LU decomposition and the SPLASH-2 benchmarks Water and Barnes-Hut. Comparison with the native Shared Memory implementation of the Alewife shows results within 12 %. The CM-5 results illustrate the need for low latency and high bandwidth communication substrates to compete with hardware solutions.

4.2 Selection of DSM

The DSMs that were considered for incorporation into the APE runtime system include SAM, CRL and Quarks. It has already been proposed that message passing applications can benefit from multithreading, by relieving the programmer from having to overlap communication and computation by hand [51]. Since the target architecture consists of Multiprocessor-based nodes, it would be also be desirable to have multiple threads run concurrently on each node, to exploit available processing power. Out of the systems surveyed, only Quarks provides any kind of multithreading. However, the base version uses *Cthreads* which are available with the Mach kernel, but not very portable. SAM and CRL use PVM for providing communication and the process control layer. The version of PVM used for these systems is not thread safe. Multithreaded extensions to PVM exist in the form of TPVM (a threaded PVM) [52] and LPVM (a light-weight PVM for SMPs) [53] but neither of these systems can currently be used on multiple SMP nodes.

4.2.1 Performance Results with CRL and Quarks

The performance results for CRL and Quarks were obtained for some programs from the Splash-2 benchmarks, and for matrix multiplication. Each program was adapted for use with the particular DSM. In the case of CRL, the region size was typically selected based on hints in the original code. SAM was found to have very low level shared memory primitives, and is targeted to a specific parallel language, Jade. Hence it was not evaluated any further.

The program parameters are given in Table 4.1.

Program	Size
FFT	1048576 complex doubles
LU-c	2048 $\times$ 2048 doubles, block size 128
OCEAN-c	258×258 grid
WATER-n2	1728 molecules, 3 steps
WATER-sp	1000 molecules, 3 steps
MATMULT	1024 $\times$ 1024 doubles, 16 blocks

Table 4.1: Problem Sizes for Quarks and CRL comparison

The results for FFT in Figure 4.1 show that the performance for an object based DSM, like CRL, can be very poor if the regions size is not selected carefully. In this program, the whole one-dimensional data array is allocated as one region, as per the original code. This sequentializes access to the region by the various processes in the update phase. Extensive modifications to the code could possibly have improved the performance.

The results with LU-c and OCEAN-c show that for some applications the performance of both CRL and Quarks does not scale well when the number of nodes is increased, refer

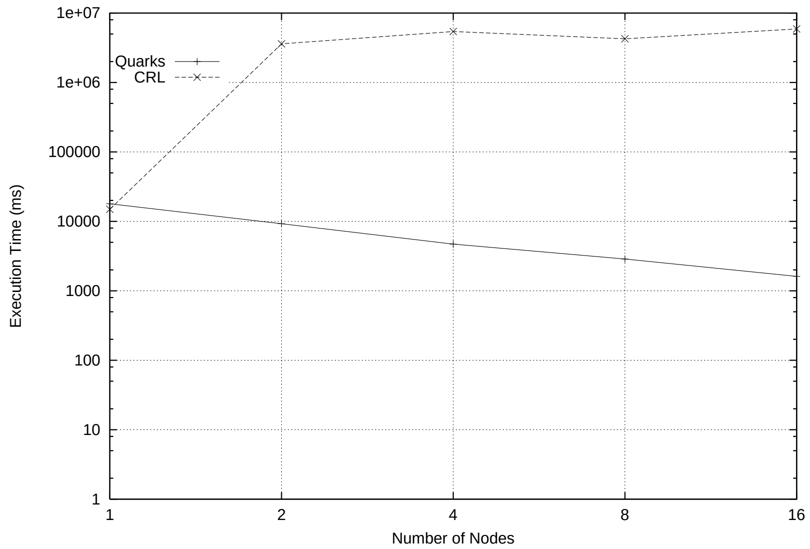

Figure 4.1: Execution Time for FFT with CRL and Quarks

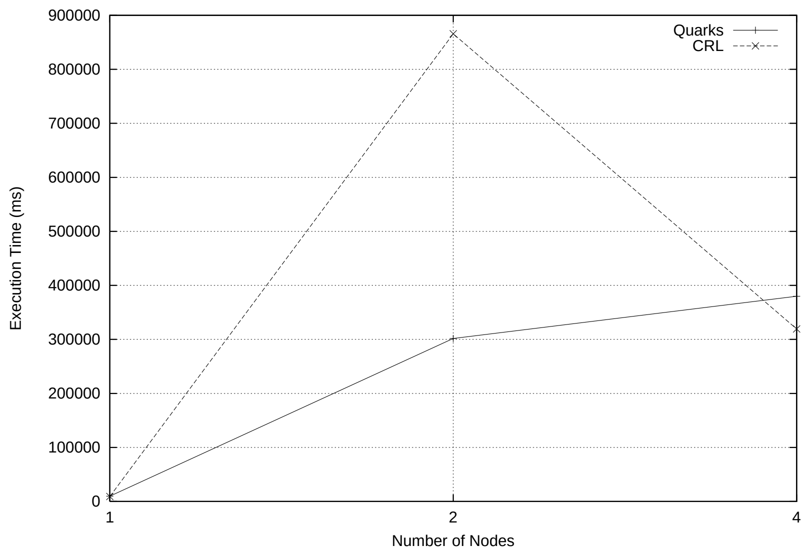

Figure 4.2: Execution Time for LU-c with CRL and Quarks

to Figures 4.2 and 4.3.

Figures 4.4 and 4.5 illustrate another problem with using an object based DSM. When the region size is too small, the execution time decreases with the number of nodes, but the initial sequential time is far higher than for Quarks, due to the very large overhead.

Figure 4.6 indicates that even the basic, untuned version of Quarks can provide some performance improvement in the case of a highly data parallel application like matrix multiplication.

Based on this preliminary evaluation, Quarks was selected as the candidate for further improvement in this work.

4.2.2 Deficiencies with Quarks

Though Quarks was selected as the target DSM, it has some important deficiencies.

Single Application Thread Only one application thread is allowed per process. If the nodes are SMPs, then multiple processes have to be instantiated and they cannot share common pages, since the address space is distinct. Quarks should be extended to allow multiple application threads per process.

Iterative Server It was found that the iterative DSM thread can lead to deadlocks under certain circumstances. It also poses a performance bottleneck if multiple independent requests arrive simultaneously at the same node. Hence the DSM thread should be changed to be a concurrent server.

User Level Threads The user level threading library *Cthreads* cannot make use of multiple processors on an SMP node, since the kernel does not know that multiple threads could be running at the same time. The runtime should be converted to use kernel

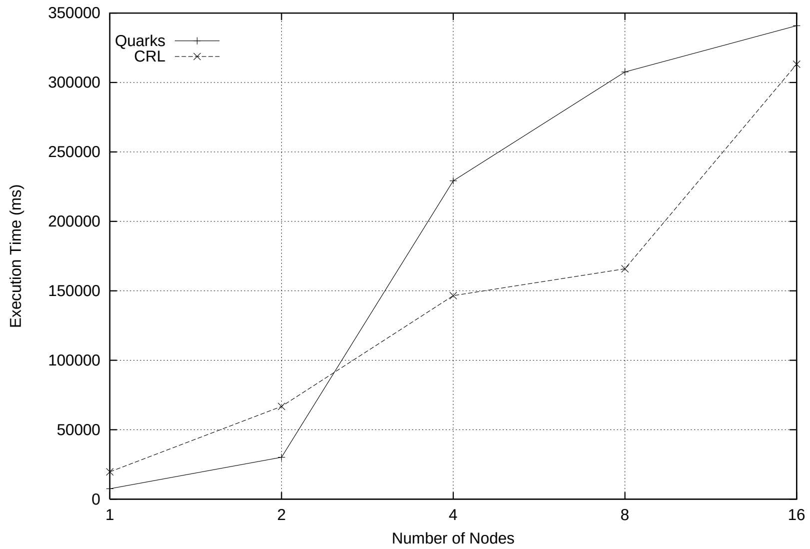

Figure 4.3: Execution Time for OCEAN-c with CRL and Quarks

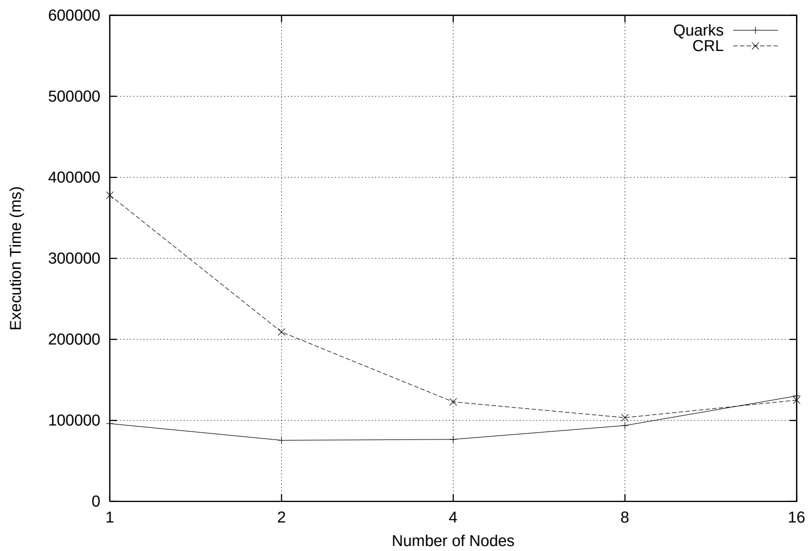

Figure 4.4: Execution Time for WATER-n2 with CRL and Quarks

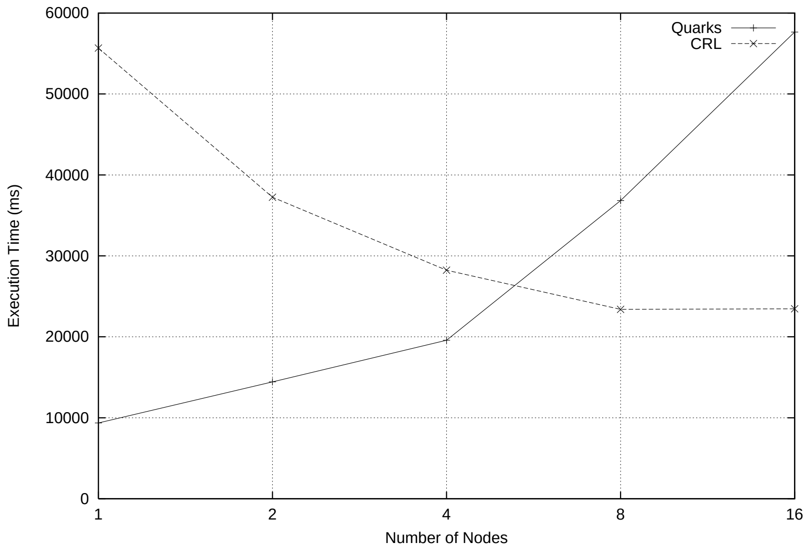

Figure 4.5: Execution Time for WATER-sp with CRL and Quarks

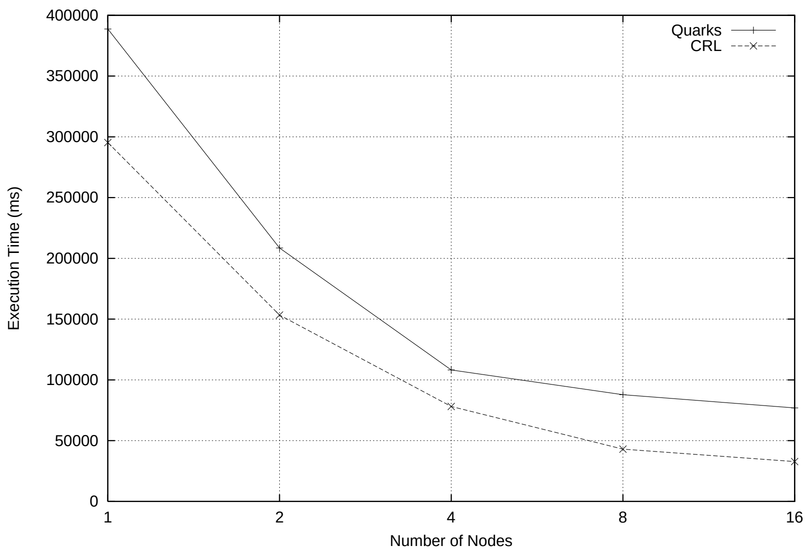

Figure 4.6: Execution Time for MATMULT with CRL and Quarks

schedulable threads, like Posix threads.

Synchronization Variables The current version of Quarks does not support condition variables, which are used for flag based synchronization in some applications. Support for this kind of synchronization primitive should be added.

5. Design Details of *Strings*

The *Strings* Distributed Shared Memory [54] is based on the Quarks system developed as part of the Flux Operating system project at the University of Utah [48]. This chapter describes the implementation details of *Strings* and highlights the differences between the two systems.

The *Strings* system consists of two main components, a shared memory server process and a runtime library. Shared memory programs that adhere to the *Strings* Application Programming Interface (API) can make use of the Distributed Shared Memory features by linking with this library.

5.1 Execution Model

A typical *Strings* program goes through three distinct phases during its lifetime: initialization, computation and termination.

5.1.1 Initialization Phase

Figure 5.1 illustrates the execution during the initialization phase of a *Strings* program.

When a *Strings* program starts up, the *master* process first determines the set of machines for the session, and the location of the *server* process from resource configuration files. It sets up a listening port and creates a *communication* thread, which polls this port for incoming messages. Since this is a blocking call, ordinarily the operating system will suspend this thread and schedule any other runnable thread. Hence the polling thread does not waste any CPU resources. A *DSM* thread is also created at this time, which waits for messages to be forwarded to it by the *communication* thread. The *master* process then *registers* itself with the *server*, obtains a unique nodeid, and also supplies its port address as part of the transaction. Child tasks are then created on remote machines using the *rsh(1)*

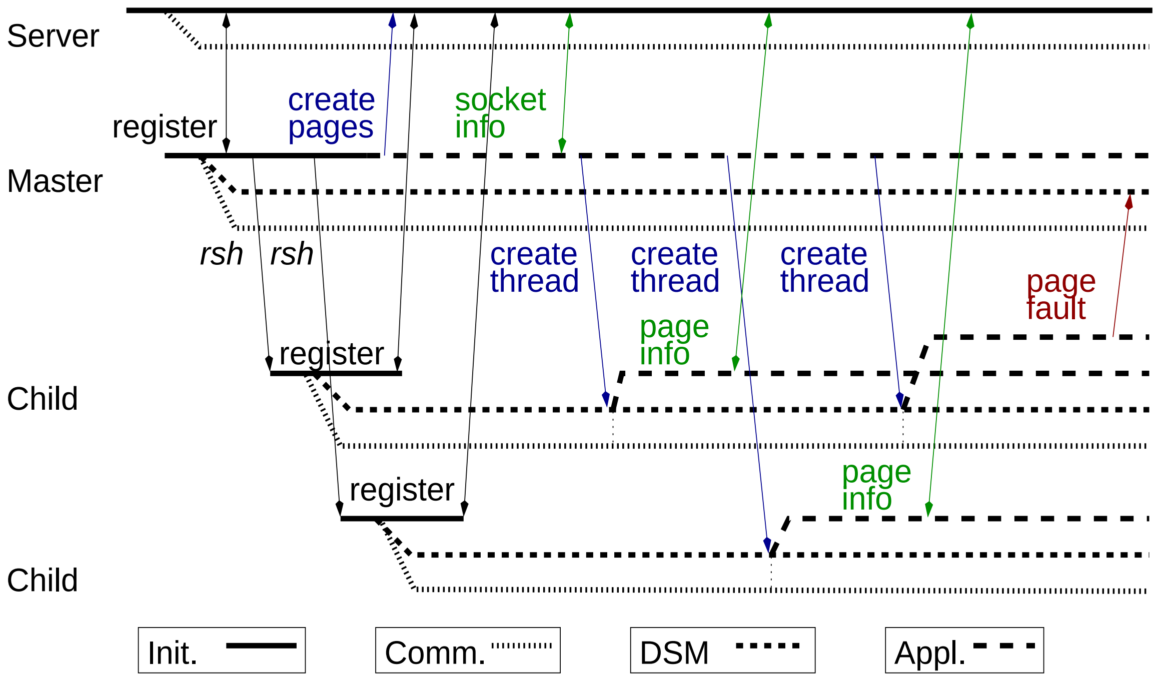

Figure 5.1: Initialization Phase of *Strings* Program

call. The child processes go through a similar initialization procedure, but after registration they typically exit from the main thread. At this time, their respective *DSM* threads are ready and waiting for requests from other processes.

The original Quarks runtime followed a similar sequence of steps. However, since Quarks uses user level threads, it cannot issue a blocking call when it listens on a network socket. Hence all network I/O was signal driven.

5.1.2 Computation Phase

After the initialization phase is over, the *master* program is ready to enter the computation phase and execute the actual application program code. For typical data parallel programs, data is first allocated and initialized by the main thread. The multiple slave tasks are created to work on their partition of the data, and finally the results are collected by the main thread. Details about each of these tasks are shown next.

Global Memory Allocation

The *master* process creates shared memory regions by sending a request to the *server*. The *server* assigns a global address and ensures that the region has a unique identifier. The *master* then creates a local copy of the region by creating a file of the appropriate size and mapping it into its memory space at the appropriate global address, using *mmap(2)*. The region is also mapped a second time at a fixed offset, this is called the shadow map. The function of the shadow map will be shown later. An entry in a local page table is created for this region. The information recorded includes the region identifier, the size of the region, its local and global addresses, the address of the variable that points to this region, and the current access state and owner. All the pages in the region are marked as read-write on the master process using the *mprotect(2)* system call. Subsequent shared memory regions are allocated the same way, except that the mapped file is extended, instead of creating a different file for each region.

Thread Creation

Remote threads are now created by sending requests to the *DSM* thread on each *child* process. As part of the request, a list of regions allocated, the addresses of the respective pointers, and a list of the initialized synchronization variables are also sent. The remote nodes contact the server and register themselves as users of these regions. They also obtain the global address and size for each region from the server. Finally they create local copies like the master process. However, child processes mark all regions as not-accessible initially, and set the owner to be the master process. When a *child* tries to access a global region, it incurs an access trap and the thread executes a fault handler that obtains the relevant page from the owning node. The page protections are then upgraded to readable or writable depending on the type of fault.

5.1.3 Termination Phase

During the termination phase, the master process sends a termination message to each child process. The child processes contact the server and deregister themselves as users of the shared regions. Once all the child processes have finished, the *master* process frees the memory allocated by the server and the whole *Strings* program terminates.

5.2 Shared Memory Server

The shared memory server process is started on a well known host in the network and is responsible for managing global memory and synchronization primitives. When the server starts up, it first initializes tables for the nodes, locks, and regions. The node table maps each node in a *Strings* distributed program to its port address and assigns a unique identifier. Client processes can then obtain these port addresses from the server for establishing communication with each other. The lock table is used to ensure that all locks are created with unique identifiers. The region table maps the addresses of globally shared data regions to region identifiers. The region identifier can be used as opaque handles, which can be used by other programs. A monitor program could request the *server* for the addresses of some regions, using their respective identifiers.

After initialization, the server listens at a well known port for requests from client processes. In most cases, this centralized server is not really a bottleneck for performance since it is accessed only during the startup phase and when global regions are allocated.

In the original Quarks runtime the server process also handled global barrier requests. It collected all the *barrier_wait* requests from the participating nodes, and sent out a *bar-*

rier_crossed message when all had arrived. The original server implementation also allocated memory for the global regions, though that was never used. In *Strings* the barrier is no longer handled by the server, instead the master process is the barrier manager. This reduces the number of messages that have to be sent on the network, since execution threads that run on the master node communicate via a UNIX domain socket.

5.3 Multilevel Threaded Implementation

Multithreading has been suggested for improving the performance of scientific code by overlapping communications with computations [51]. The use of multiple threads in DSM is a comparatively recent development, and all implementations have been in user space [48, 55]. The distinguishing feature of *Strings* is that it incorporates Posix1.c threads multiplexed on kernel light-weight processes for better performance. The kernel can schedule multiple threads across multiple processors using these lightweight processes.

The original Quarks system used user level *Cthreads* for implementing part of the system, but allowed only a single application thread. *Strings* allows multiple application threads to be created on a single node. This increases the concurrency level on each node in a SMP cluster.

The *Strings* runtime uses threads at three different levels. Application threads are based on internal threads, which are in turn built on kernel level threads provided by the underlying operating system.

5.3.1 Kernel Level Threads

Threads are light-weight processes that have minimal execution context, and share the global address space of the program. The time to switch from one thread to another is very small when compared to the context switch required for full-fledged processes. Moreover the implicit shared memory leads to a very simple programming model. Thread implementations are distinguished as being user level, usually implemented as a library, or kernel level in terms of light-weight processes. Kernel level threads are a little more expensive to create, since the kernel is involved in managing them. However, user level threads suffer from some important limitations. Since they are implemented as a user level library, they cannot be scheduled by the kernel. If any thread issues a blocking system call, the kernel considers the process as a whole, and thus all the threads in it, to be blocked. Also, on a multiprocessor system, a program using user level threads is considered as a monolithic process and can only run on one processor at a time. User level threads do allow the programmer to exercise very fine control over their scheduling within the process.

Kernel level threads can be scheduled by the operating system across multiple processors. Most modern UNIX implementations provide a light-weight process interface on which these threads are then multiplexed. The thread package adopted for *Strings* was the standard Posix 1003.1c threads as available on operating systems like Solaris 2.6, and Linux 2.x. Since thread scheduling is handled by the kernel some parts of the original Quarks system had to be changed to ensure correctness. Previous work on multithreaded message passing systems has pointed out that kernel-level implementations show better results than user level threads for a message size greater than 4k bytes [56]. Since the page size is usually 4k or 8k bytes, it suggests that kernel threads should be useful for DSM systems.

5.3.2 Internal Threads

The internal threads are built on top of the kernel thread. Each internal thread has a unique global identity, composed of the node identifier and the local thread identifier. A separate message buffer and incoming message queue is associated with each such thread. This allows the *DSM* server to create multiple concurrent threads to handle incoming requests. Once the request has been handed off to a new thread, all communication with the client is handled by this thread and the *DSM* thread is free to handle new messages.

5.3.3 Application Threads

The application threads are at the highest level and are created by an application process. Since these are layered on top of the internal threads, each application thread has a unique global identifier which can be used to communicate with it. Application threads also associate the global page table, the global synchronization primitives and the shared memory regions with their context.

5.4 Communication System

The Quarks communication system provides a reliable messaging system on top of UDP. This same system is used for *Strings*. Each process establishes a listening socket and registers this socket with the *server* process. The *server* process is assumed to listen on a well known host at a well known socket. The message structure is as shown in Figure 5.2. The *Type* field encodes whether it is an asynchronous or synchronous transmission, and

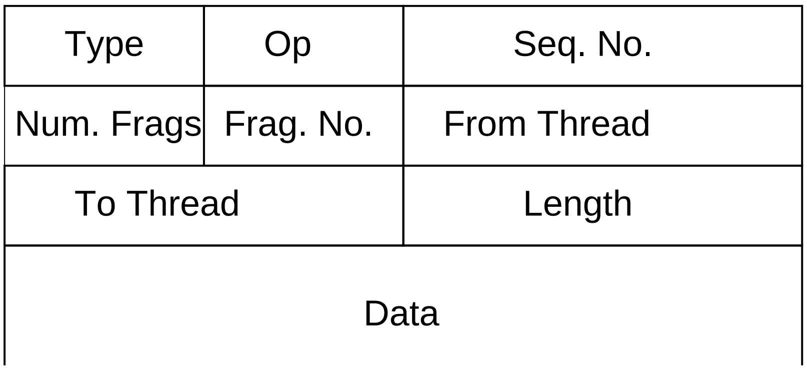

Figure 5.2: Message Structure

whether the message is an initial message or a reply. The *Op* field indicates the operation requested, whether thread creation, shared memory allocation, lock request etc. The sequence number is used to ensure that stale messages get discarded. In general the runtime used synchronous messages which block the thread until a reply is received. Hence out of order message arrival is not a problem. Messages larger than a fixed maximum size are fragmented. The maximum message size has been set to accommodate a complete memory page within one message. Early generation DSM systems used interrupt driven I/O to obtain pages, locks etc. from remote nodes. This can cause considerable disruption at the remote end, and previous research tried to overcome this by aggregating messages, reducing communication by combining synchronization with data, and other such techniques [13]. *Strings* uses a dedicated communication thread, which monitors the network port, thus eliminating the overhead of interrupt driven I/O. Incoming message queues are maintained for each active thread at a node, and message arrival is announced using condition variables. This prevents wasting CPU cycles with busy waits. Once a message has been processed and replied to, the reply message is saved in a buffer. If a request is repeated due to a timeout, the stored reply can be sent directly by the communication thread. The original Quarks message sub-system used signal driven I/O, since the runtime was based on user level threads.

The advantage of using UDP lies in the fact that multiple nodes can communicate with a single node using only one socket address. For switched networks it was found that very few messages were ever dropped.

5.5 Fault Handler

Strings uses an access fault handler to maintain a consistent view of memory. When a process accesses a memory region that has insufficient permissions, an access violation trap will be generated. The fault handler first converts the local faulting address to a global address using the page table. Depending on the consistency model, it then initializes a fetch of the page contents from the owning node, or promotes the page directly. This process is detailed in Figure 5.3.

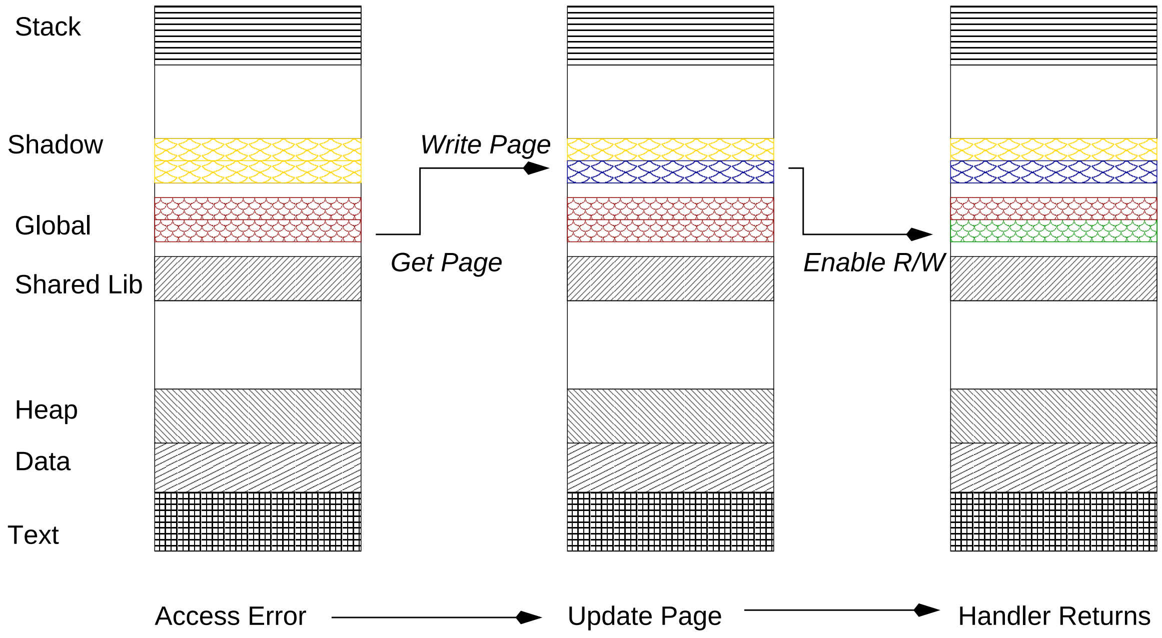

Figure 5.3: Memory Access Violation

After obtaining the page from the current owner, its contents have to be updated and its permissions have to set to readable or writable, depending on whether a read-fault or a write-fault occurred. To update the page, its permissions would have to be made at least writable, however another application thread that accesses the page at that time could then see partially updated data. In the case of systems that use a purely user level thread implementation, like Quarks and CVM, it is possible to make the update atomic by not allowing any other thread to be scheduled during this time. Since kernel threads are scheduled by the operating system, their execution cannot be directly controlled by the program. A possible solution is to suspend all other threads while the update is in progress. However this approach can lead to deadlocks if one of the suspended threads is holding a vital lock, and it would sacrifice all possible parallelism during the update. The solution adopted for *Strings* is to map each regions twice, and to do page updates to the *shadow* copy. The primary page is promoted to the appropriate access permissions only after all the data has been copied in. Another approach attempted was to write directly to the mapped file, and then upgrade the memory region. However it was found that the direct file update causes a performance degradation of approximately 20 %.

Quarks used anonymous mappings of memory pages to addresses determined by the system, but this works only with statically linked binaries. Programs that use shared libraries can cause the operating system to map the same page to different addresses in different processes. While an address translation table can be used to access opaquely shared data, it is not possible to pass pointers to shared memory this way, since they would potentially address different regions in different processes. An alternate approach would be to preallocate a very large number of pages on the heap, as done by CVM and TreadMarks, but this associates the same large overhead with every program, regardless of its actual requirements.

Strings solves this problem by allocating all regions starting at a fixed base address, the bottom of the stack segment. Quarks currently supports sequential consistency using an invalidate protocol, as well as release consistency using an update protocol [57, 48]. *Strings* has an improved implementation which aggregates multiple diffs to decrease the number of messages sent.

5.6 Synchronization Primitives

Strings provides global locks, barriers and condition variables for synchronization in shared memory programs.

5.6.1 Global Locks

The original Quarks lock code did not support requests for the same lock from different threads on the same node. The *Strings* implementation is enhanced to support multiple application threads.

Global locks are allocated by the *master* process by obtaining unique identifiers from the *server*. The lock ownership is migratory, with the *master* being the initial owner. When a lock is created, a local lock structure is initialized with the lock identifier, the probable owner, the lock status, and a queue for lock requests.

The lock acquisition code is shown in Figure 5.4. When an application thread on a node tries to acquire a lock, it first locks access to the local lock structure. If the lock is found to be local and free, the lock status is set to busy. Else the thread checks whether any requests are pending, in which case the holder is set to the local node. This ensures that only one request goes to the current lock owner. A message is then sent to the current owner, and the thread waits until the lock is granted. After the lock arrives the incoming queue is appended to the local queue, and the lock state is updated.

When a request for a lock arrives at a node, the code in Figure 5.5 is executed. The handler first locks access to the local lock structure. If the lock is local and free, it is sent to the requesting node. If the request is from another thread on the same node, the handler marks the lock as held. This prevents a possible race condition, if another handler is invoked before the local requesting thread can mark the lock as busy. If the lock is currently held,

```
void Acquire(Id lockid)
{
    Locallock *lk = &lltable[lockid];
    int done =0;
    Message   *msg, *reply;
    Waiter *rec;
    Id holder;

    /* guarantee exclusive access to the state */
    pthread_mutex_lock(&lk->mutex);

    if (lk->lockPO == Nodeid && !lk->held){
      lk->held = 1;  /* lock is free, just grab it */
      pthread_mutex_unlock(&lk->mutex);
      return;
    }

    holder = lk->lockPO;

    /* Mark first request if required */
    if(lk->acq_req_pending)
      holder = Nodeid;
    else if(lk->lockPO != Nodeid)
      lk->acq_req_pending =1;

    MSG_INIT(msg, MSG_OP_ACQUIRE_LOCK);
    MSG_INSERT(msg, lockid);
    pthread_mutex_unlock(&lk->mutex);

    ssend_msg(holder, DSM_THREAD, msg);

    while(!done){
      reply = receive(0); /* wait for reply with lock */
      switch(reply->op){
        case MSG_OP_LOCK_GRANTED:
          done = 1;
          break;
        case MSG_OP_LOCK_REQ_QUEUED:
          break;
      }
    }

    /* finally got the lock */
    unpack_list(reply->waiter_queue,lk->waiters_queue);

    /* update the lock status */
    pthread_mutex_lock(&lk->mutex);
    lk->lockPO = Nodeid;
    lk->acq_req_pending=0;
    lk->held = 1;
    pthread_mutex_unlock(&lk->mutex);
}
```

Figure 5.4: Lock Acquire Code (edited for clarity)

```
void handle_lock_request(Id lockid, Thread req_th)
{
    Locallock *lk = &lltable[lockid];
    Waiter *rec;
    Message *msg, *reply;

    pthread_mutex_lock(&lk->mutex);
    /* local node is owner and the lock is free */
    if (lk->lockPO == Nodeid && !lk->held){
      MSG_INIT(msg, MSG_OP_LOCK_GRANTED);
      MSG_INSERT(msg, lockid);
      lk->lockPO = Node(req_th);

      /* Ensure atomicity if transfer within node,
         by setting lk->held = 1. */
      if(Node(req_th) == Nodeid)
        lk->held =1;
      ssend_msg(req_th, msg);
      pthread_mutex_unlock(&lk->mutex);
    }else if (lk->held || lk->acq_req_pending){
      /* queue the request */
      rec = (Waiter *) malloc(sizeof(Waiter))
      rec->th = req_th;
      list_append(lk->waiters_queue, rec);
      MSG_INIT(msg, MSG_OP_LOCK_REQ_QUEUED);
      MSG_INSERT(msg, lockid);
      pthread_mutex_unlock(&lk->mutex);
      ssend_msg(req_th, msg);
    } else if (lk->lockPO != Nodeid){
      /* lock has passed on, forward request */
      forward_lock_req(lockid, req_th, lk->lockPO);
      pthread_mutex_unlock(&lk->mutex);
    }
}
```

Figure 5.5: Lock Handler Code (edited for clarity)

or other requests are pending, the incoming request is queued. Thus the system maintains a distributed queue for locks. If the lock has been passed to another node, it is forwarded to the last known owner.

When a lock is released, the thread first acquires exclusive access to the local lock state, as shown in Figure 5.6. The current set of shared pages is purged as part of the coherency protocol. The local queue is checked for pending requests. If the queue is not empty, the lock and the remainder of the queue are handed over to the next owner. If the transfer is local, the lock is still marked as busy to avoid race conditions. The new owner appends the contents of the incoming queue to the local queue.

In the current implementation, the queue is drained in FIFO order. No preference is given to pending requests from other threads on the owning node. This approach guarantees progress and fairness, at the cost of increased wait times and traffic. Other researchers have shown that serving local requests out-of-order works well in practice [55].

5.6.2 Barriers

Barriers are allocated locally, since the *master* also serves as the central barrier manager. When an application thread arrives at a barrier, it sends a request to the master. In the case of *release consistency*, page updates are also sent at this time. The barrier manager replies with a wait message to each arriving request, except for the last one. The last arriver is sent a *barrier_crossed* message. After this, all the waiting threads are also sent *barrier_crossed* messages. A possible optimization that can be used here is to wait for all local application threads to arrive before sending the message to the barrier manager. This optimization should work well for purely data parallel programs, where it is assumed that all application threads execute the same code and arrive at the same barrier. In a more general model, the

```
void Release(Id lockid)
{
    Locallock *lk;
    int       llindex;
    int       qlen=0;
    Waiter    *rec;
    Threadid  th;
    Id        al_seqno;
    Id        node;
    Message   *msg, *reply;

    if ((llindex = lltable_index(lockid)) == -1)
        PANIC("Lock unknown");

    lk = &lltable[llindex];

    pthread_mutex_lock(&lk->mutex);
    PurgeDUQ();

    if (! list_empty(lk->waiters_queue)){
        node = Node(th);
        lk->lockPO = node;
        /* if transfer within the node, memorize it */
        if(lk->lockPO == Nodeid)
          lk->in_transfer = 1;

        MSG_INIT(msg, MSG_OP_LOCK_GRANTED);
        MSG_INSERT(msg, lockid);
        pack_list(msg, lk->waiters_queue);

        /* if the receiving thread is on this node,
           it will extract the message
           without holding the lock */
        reply = ssend_msg(th, msg);
    }

    if(!lk->in_transfer)
      lk->held = 0;
    else
      lk->in_transfer = 0;
    pthread_mutex_unlock(&lk->mutex);
}
```

Figure 5.6: Lock Release Code (edited for clarity)

local process cannot know how many application threads participate in a barrier.

5.6.3 Condition Variables

Strings also implements condition variables for flag based synchronization. An application thread that needs to wait for a condition to be true sends a request to the *server*. The server queues this request locally. An application thread that wants to signal or broadcast the condition sends a message to the *server*. The *server* can then forward the signal to the application threads in the queue. To avoid the problem of lost signals, an associated global lock has to be owned by a process before it can wait on a condition variable or signal it.

5.7 Concurrent DSM thread

The original Quarks *dsm_server* thread was an iterative server that handled one incoming request at a time. It was found that under certain conditions, lock requests could give rise to a deadlock between two communicating processes. *Strings* solves this by forking separate threads to handle each incoming request for pages, lock acquires and barrier arrivals. Relatively fine grain locking of internal data structures is used to maintain a high level of concurrency while guaranteeing correctness when handling multiple requests.

The advantages of these design decisions are shown in the next chapter.

6. Performance Evaluation

The evaluation of *Strings* was done in three parts. The first shows the effect of the various design decisions on the runtime performance. The next part compares the performance with respect to three other DSM systems, CRL [50], CVM [46], and Quarks [48]. The final part shows the effect of using different networking technologies on the overall performance of the system.

6.1 Experimental Environment

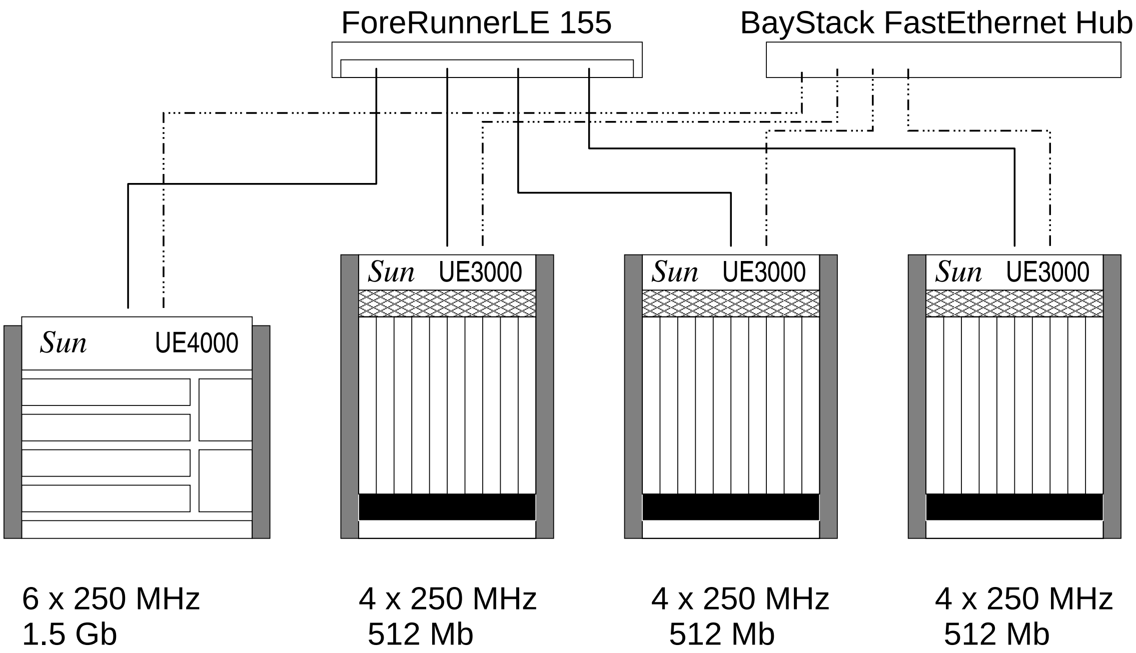

Figure 6.1: Evaluation Environment

All the experiments were carried out on a cluster of four SUN UltraEnterprise Servers, as shown in Figure 6.1. The first machine is a 6 processor UltraEnterprise 4000 with 1.5 Gbyte memory. For *Strings* and Quarks the *server* process was always run on this machine. The master program for all DSMs was always started on this machine. The other machines are 4 processor UltraEnterprise 3000s, with 0.5 Gbyte memory each. All machines use 250 MHz UltraSparcII processors, with 4 Mb external cache. Two networks are used to interconnect the machines in the cluster, 155 Mbps ATM with a ForeRunnerLE 155 ATM switch, and 100 Mbps FastEthernet with a BayStack FastEthernet Hub.

Most previous work on DSMs has presented performance in a similar environment. TreadMarks has been studied on ATM networked DECstation-5000/240s [42, 11], Pentium 166s with 100 Mbps Switched Ethernet [58], and an 8-processor IBM SP-2 [59]. *CVM* results were presented on the IBM SP2 and a DEC Alpha cluster [60] and *CRL* on the Alewife [50].

6.2 Test Programs

The suite of test programs used to do the evaluation consists of some programs from the SPLASH-2 benchmark suite [61], a complete application program for deblurring images obtained from Magnetic Resonance Imaging (MRI), matrix multiplication (MATMULT), as well as a kernel for doing Successive Over Relaxation program (SOR)on a grid of numbers.

6.2.1 SPLASH-2 Benchmark Programs

The SPLASH-2 Benchmark programs have been written for evaluating the performance of shared address-space multiprocessors and include application kernels as well as full fledged code. This suite was selected for the evaluation, since the assumed execution model is very similar to SUIF generated code. The practical experience gained while adapting the programs for use with the DSMs was used in the design and implementation of the compiler extensions.

The execution model for SPLASH-2 programs follows the Single Program Multiple Data (SPMD) type, and has three phases. In the first phase the main task reads in command-line parameters and the initial data. It sets up a shared memory region and allocates globally shared data from this region. Synchronization variables are also initialized. After this initialization phase, tasks are created to execute the actual *slave()* routine in the computation phase. The master also runs a copy of this routine. In the termination phase the master thread collects the results and generates timing statistics. The SPLASH-2 program suite has been designed to be portable by encapsulating shared memory initialization, synchronization primitives, and task creation within ANL macros [24]. These macros should then be adapted for any shared memory platform.

The data access patterns of the programs in the SPLASH-2 suite have been characterized in earlier research [62, 63]. FFT performs a one-dimensional Fast Fourier Transform of n complex data points. Three all-to-all interprocessor communication phases are required for a matrix transpose. The data access pattern is hence regular. Two programs for the blocked LU factorization of a dense matrix form part of the suite. The non-contiguous (LU-n) version has a single producer and multiple consumers. It suffers from considerable fragmentation and false sharing. The contiguous version (LU-c) uses an array of blocks to improve spatial locality. The sorting kernel (RADIX) performs a radix sort on a set of integers. The program suffers from a high-degree of false sharing at page granularity during a permutation phase. Each processor writes the values of its keys into a random location in a shared array. The Ocean simulation program with contiguous partitions (OCEAN-c) simulates large scale ocean movements. This version uses a red-black Gauss-Seidel multigrid equation solver and has a regular nearest-neighbor type access pattern. The last two programs tested evaluate the forces and potentials occurring over time in a system of water molecules. The first version (WATER-n2) uses a simple data structure, which results in a less efficient algorithm. The second version (WATER-sp) uses a 3-D grid of cells so that a processor that owns a cell only needs to look at neighboring cells to find interacting molecules. Communication arises out of the movement of molecules from one cell to another at every time-step.

6.2.2 Image Deblurring

The application tested is a parallel algorithm for deblurring of images obtained from Magnetic Resonance Imaging. The images generated may suffer a loss of clarity due to inhomogeneities in the magnetic field. One of the techniques for removing this blurring artifact is the demodulation of the data for each pixel of the image using the value of the magnetic field near that point in space. This method consists of acquiring a local field map, finding the best fit to a linear map and using it to deblur the image distortions due to local frequency variations. This is a very computation intensive operation and has previously been parallelized using a message passing approach [64]. Each thread deblurs the input image around its chosen frequency points in parallel. After deblurring aroung a frequency point, the relevant portions of the final image have to be updated. Since the portions updated by different threads could overlap, the update has to be done under the protection of a global lock.

6.2.3 Matrix Multiplication

The matrix multiplication program computes the product of two dense square matrices. The resultant matrix is divided across the processors using a row-wise block distribution. The size of each of the blocks has been set to a multiple of the page size of the machine, ie. 8 kbytes. Each application thread computes at least one complete block of contiguous values. Hence there is no false sharing in this implementation and the application is close to ideal for execution on a page based DSM system.

6.2.4 Successive Over Relaxation

The successive over relaxation uses a red-black algorithm and was adapted from the CVM sample code. Most of the traffic arises out of nearest neighborhood communication at the borders.

6.3 *Strings* Evaluation

The first performance evaluation consisted of testing the various design feature of the *Strings* system. All the runs were carried out on the ATM network. Release Consistency model was used for maintaining memory consistency.

6.3.1 Program Parameters

The programs and their parameters selected for this part of the evaluation are shown in Table 6.1. All other parameters were left at their default values.

Program	Parameters	Resident Size (Sequential)
FFT	1048576 complex doubles	104 Mbyte
LU-c	2048 $\times$ 2048 doubles, block size 128	71 Mbyte
LU-n	512 $\times$ 512 doubles, block size 32	11 Mbyte
RADIX	1048576 integers, radix 1024	34 Mbyte
WATER-n2	4096 molecules, 3 steps	45 Mbyte
WATER-sp	4096 molecules, 3 steps	76 Mbyte
MRI	14 frequency points, PHANTOM image	22 Mbyte
MATMULT	1024 $\times$ 1024 doubles, 16 blocks	55 Mbyte

Table 6.1: Problem Sizes for *Strings* Evaluation

Referring to the third column of table 6.1, it can be seen that none of the test programs exceeded the memory available on a single node of the cluster. This implies that the sequential execution time is not increased abnormally due to excessive page swapping, which could have led to optimistic speed-up results.

For each case, the following runs were carried out:

P16T1-KT: sixteen processes, four per machine, with a single application thread per process. The runtime uses kernel threads. This base case approximates the typical environment used in previous work on DSMs.

P4T4-KT: four processes, one per machine with four application threads per process. Kernel threads are used throughout. Multiple application threads can be scheduled across processors in this case, and multiple requests can be handled by the *DSM_server* thread.

P4T4-UT: same as above, but using user level threads. This was approximated by allowing only the default process level contention for the threads. These were then constrained to run on a single processor per node.

P4T4-SIG: four processes, one per machine with four application threads per process with signal driven I/O. Instead of using polling, incoming messages generate a SIGIO signal, and a signal handler thread then processes the request.

P4T4-SS: four processes, one per machine with four application threads per process. Compared to the P4T4-KT case, the *DSM_server* thread is changed to handle all requests sequentially, except for lock requests. This was required to avoid a deadlock problem in the original runtime.

6.3.2 Experimental Results

The overall execution times are shown in Figure 6.2, with results normalized to the total execution time of the *P16T1-KT* case. The data is separated into various components including:

Page Fault: the total time spent in the page-fault handler, adjusted for multiple overlapping faults on the same node.

Lock: the time spent to acquire a lock.

Barrier Wait: the time spent waiting on the barrier after completing Release Consistency related protocol actions.

Compute: this includes the compute time, miscellaneous components like the time spent sending diffs to other nodes, as well as the startup time.

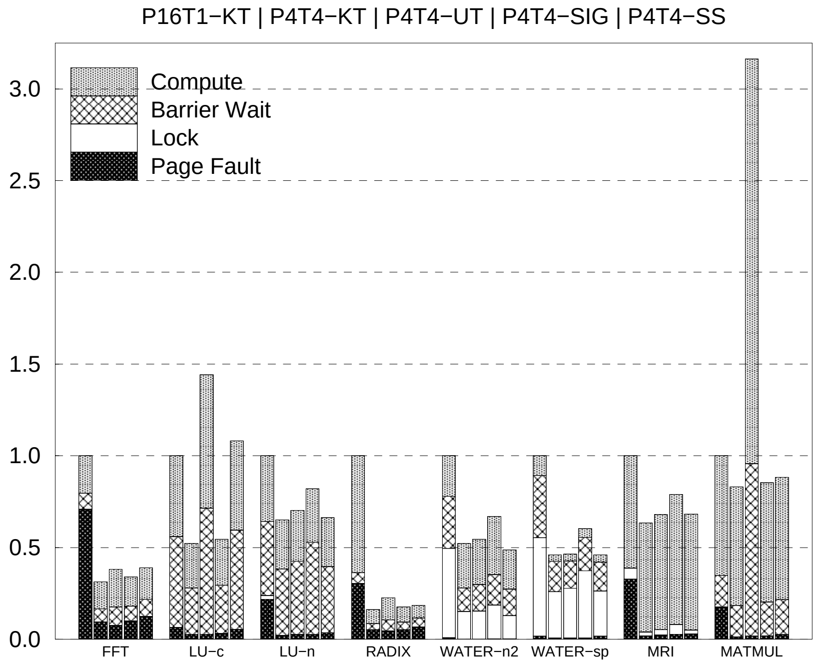

Figure 6.2: Execution Time Normalized to P16T1-KT

The execution time for LU-c, LU-n, WATER-sp, and WATER-n2 can be seen to be dominated by the cost of synchronization events ie. barriers and locks. This is primarily due to the use of the release consistency model, which requires that all modifications to writable pages be flushed at synchronization points in the code.

It can be seen that using multiple application threads on top of kernel threads can reduce the execution time in most case by up to 50 % as compared to using multiple processes. The primary reason is due to the reduction in the number of page faults, as can be seen in Figure 6.3. When multiple threads on a node access the same set of pages, they need to be faulted in only once.

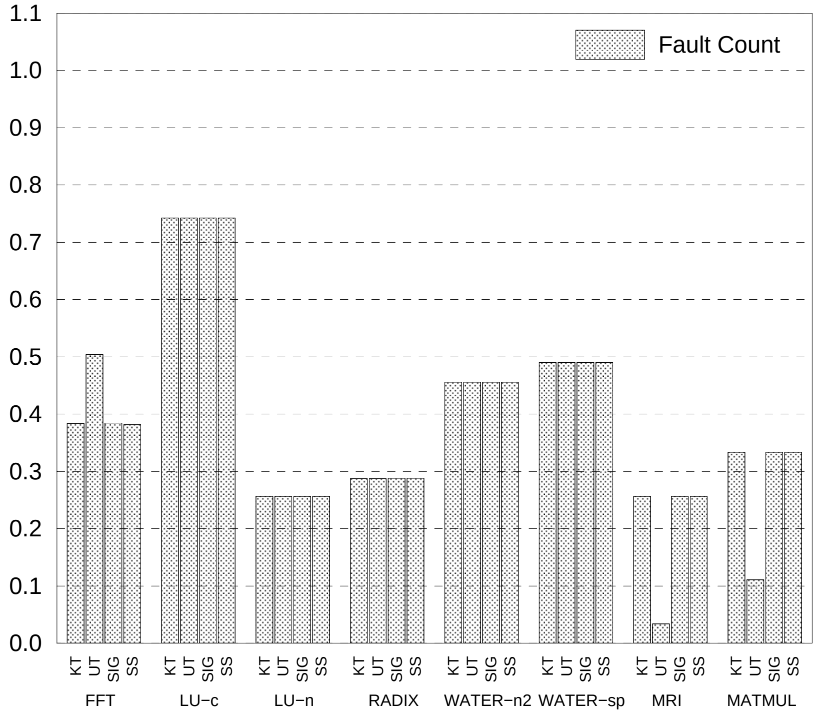

Figure 6.3: Page Fault Count Normalized to P16T1-KT

Applications with a high computational component, like LU-c, RADIX, and MATMULT illustrate the problem with using only user-level threads. Each compute bound thread competes for the same processor of each machine, leading to very inefficient utilization. This can also be seen by the larger average barrier cost in case of MATMULT, since the first arriving thread has to wait for the remaining threads to finish computing.

Program	Bytes	Messages	Avg. Size
FFT	36 M	7578	4750
LU-c	56 M	9516	5885
LU-n	10 M	2686	3723
WATER-sp	20 M	24045	831
RADIX	13 M	2825	4601
WATER-n2	13 M	95008	136
MRI	4 M	1319	3032
MATMULT	11 M	2553	4309

Table 6.2: Traffic per Node

The behavior of signal driven I/O compared to polled I/O can be explained by referring to Table 6.2. The overhead of signal generation becomes apparent as soon as the message size drops below 4000 bytes. This translates into visibly increased execution times for LU-n, WATER-sp, WATER-n2 and MRI.

Program	Concurrent Server	Iterative Server
FFT	4626	9
LU-c	7027	9
LU-n	2079	9
WATER-sp	20050	90
RADIX	1905	39
WATER-n2	90784	30456
MRI	815	16
MATMULT	1373	12

Table 6.3: Threads Created per Node

Table 6.3 shows that the concurrent server can create a large number of threads for servicing DSM requests. The iterative server creates threads only for handling lock requests, the difference in counts is due to page requests and barrier calls.

The advantage of using a concurrent *DSM_server* is most visible in the time taken per page fault, as shown in Figure 6.4. If multiple requests for different pages arrive at the same node, the iterative server serializes the requests and thus increases the perceived latency. The lock time is not affected since both approaches use the same algorithm.

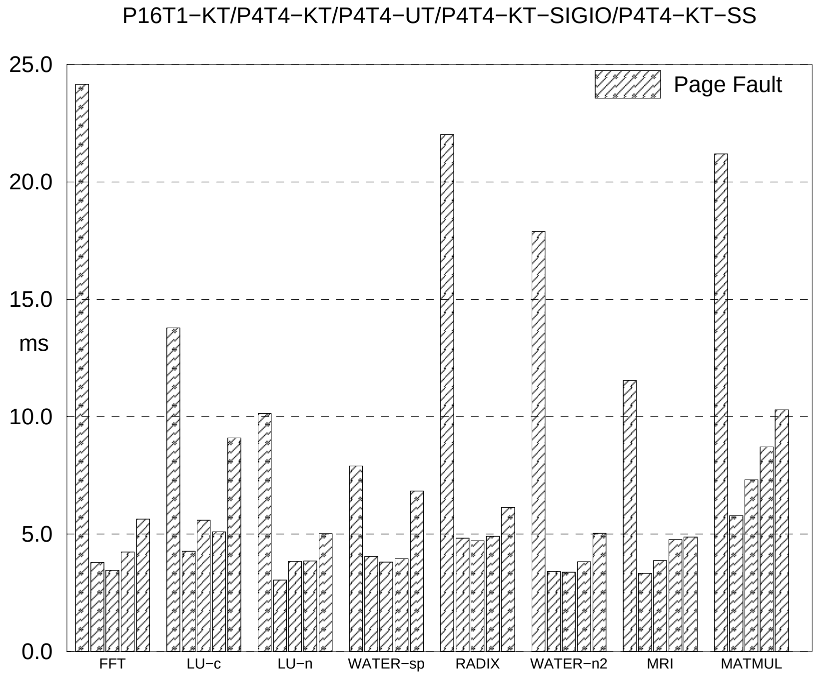

Figure 6.4: Page Fault Time

Additional runs were carried out with varying number of application threads to look at speed-up results.

P1T1-KT: one process, on the UltraEnterprise 4000, ie. the sequential case.

P4T1-KT: four processes, one per machine, with a single application thread per process.

P4T2-KT: four processes, one per machine, with two application threads per process.

P4T4-KT: four processes, one per machine, with four application threads per process.

The results for LU-c, LU-n, MRI, and MATMULT are shown Figure 6.5. It can be seen that in most cases the execution time drops by 50 % whenever the number of application threads is doubled. This also supports the notion of using multiple threads to increase the utilization of such SMP clusters.

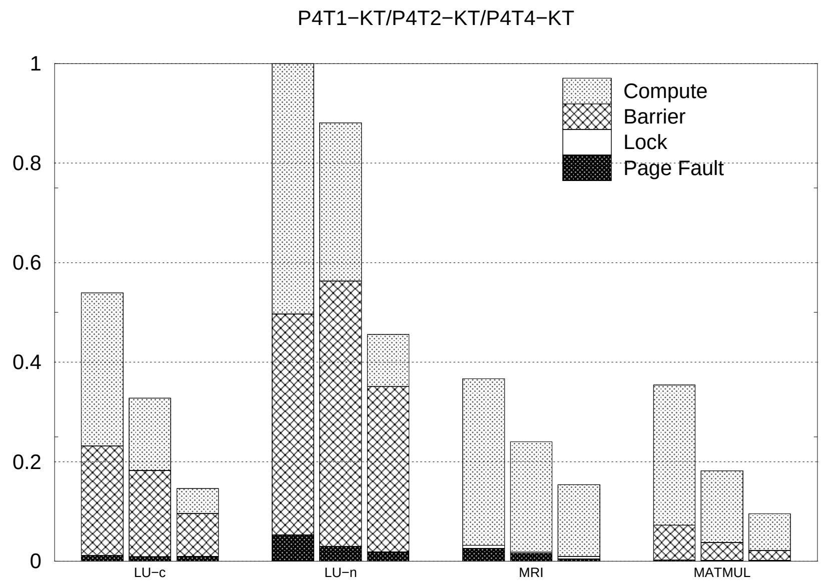

Figure 6.5: Execution Time Normalized to P1T1-KT

6.3.3 Summary

The performance of each implementation depends to some extent on the data sharing and communication pattern of the application program. While going from the *P16T1-KT* base case to *P4T4-KT* the performance in general improves. This is primarily the result of using multiple kernel threads to reduce the cost of page-faults, lock acquisition and barriers. The *P4T4-SIG* runs show that using a dedicated communication thread to poll for incoming messages is a preferred alternative to signal driven I/O. Though previous research has tried to use user-level thread packages [48, 55] to improve the performance of DSM systems, MATMULT and MRI clearly show that kernel threads provide far better performance. The concurrent *DSM_server* approach reduces the latencies for page-faults by allowing multiple requests to be handled concurrently. However the overall impact is not very high since the execution time is dominated by the barrier time in most applications.

When compared to message passing programs, additional sources of overhead for traditional software DSM systems have been identified to include separation of data and synchronization, overhead in detecting memory faults, and absence of aggregation [65]. Researchers have attempted to use compiler assisted analysis of the program to reduce these overheads. Prefetching of pages has been suggested by a number of groups for improving the performance of TreadMarks, by saving the overhead of a memory fault [13, 59]. This technique sacrifices the transparency of a page oriented DSM, but can be incorporated in parallelizing compilers. In *Strings*, a faulting thread does not block the execution of other application threads on the same process, hence the benefit of prefetching is not expected to be very large. Asynchronous data fetching was also identified to be a source of performance improvement [12]. In *Strings*, the dedicated *DSM_server* and communication thread together hide the consistency related actions from the application threads.

6.4 Comparison with Other DSM Systems

This section presents performance results of *Strings* when compared to some other publicly available DSM systems. The systems evaluated are CRL, CVM and Quarks. The experimental readings in this section were obtained on the FastEthernet network, since CRL was not able to use the ATM network with PVM.

6.4.1 C Region Library

The C Region Library (*CRL*) [50] is an all-software DSM system featuring operating system and language independence, portability and efficiency . The programmer defines arbitrarily sized regions and delimits access using annotations. This avoids the problem of false sharing as well as inefficiency when using too small a coherence unit, which are common in page-based and hardware based systems respectively. *CRL* also uses main-memory as a large secondary cache to offset the latencies of communications operations. Operations on regions are considered indivisible units. The shared memory operations must be explicitly marked by the programmer in this system with *rgn_start_read/write* and *rgn_end_read/write* calls.

In the execution model, a master program starts child copies of the program on each node and then exits. Variables that have global file scope but are private on each node can be initialized in parallel on all the nodes. Shared regions created on one node are assigned portable identifiers which must then be broadcast to the others. Before a shared region can be accessed, it must first be mapped into the local address space using the region identifier. The same region may be mapped to different local addresses at different times, hence the region identifiers must be used instead of global pointers. Mapped regions are read and written by explicitly indicating the start and end of an access. Since accesses are explicit, no locks are required while writing to shared regions. A single barrier is implemented and managed by the master process.

Current implementations exist for the CM-5, the MIT Alewife as well as for a network of workstations. The network version uses PVM to set up the processes and broadcast the *UDP* sockets at which each node listens. During the execution of the program all communication is done via interrupt driven I/O on these sockets. PVM is used again during the program termination phase.

6.4.2 CVM: Coherent Virtual Machine

CVM [46] is a page-based DSM implemented as a user level library which can be linked in with shared memory programs. The base system provides a generic coherence protocol, user level lightweight threads and reliable network communication on top of *UDP*. The shared memory system is created by pre-allocating a large number of pages from the heap and then managing this space. *CVM* programs create remote processes using *rsh*. Each process then executes the sequential part of code until application threads have to be created, at which time the remote processes wait for a *create* message from the master. This model restricts the use of *CVM* to data parallel programs where every thread executes identical code. All variables initialized by the master process will also be initialized by the remote slave processes. The shared memory region is protected using *mprotect*. When a process accesses a page in the shared region, a fault-handler fetches the page from the master process. *CVM* uses fixed-home pages, locks are assigned round-robin and barriers are managed by the master process. The page ownership can be assigned to the master, round-robin or evenly across the nodes. An extensible interface for adding additional coherence protocols is provided. The current release supports Lazy Multi-Writer, Lazy Single Writer, and Sequential consistency. It runs on Solaris 2.5, AIX 4.1 and Digital Unix 3.2. *CVM* "holds" on to a freshly arrived page for 16 μs to reduce the effect of false sharing.

6.4.3 Test Program Modifications

Each SPLASH-2 benchmark program was modified locally so as to conform to the execution model assumed by the respective DSM environment. *CVM* only requires an insertion of an initialization call near the beginning of the program. Allocation of shared memory and initialization of locks and barriers is done by writing appropriate ANL macros. *Strings* requires that variables that are global in file scope be explicitly initialized by all nodes. The code used for *Quarks* is identical to the *Strings* version. Since *CRL* has no provision for global pointers, some data structures have to be rewritten to use portable region identifiers. One must also ensure that only the master program assigns such region identifiers to global regions, else every node would have a private copy of the region. Each access to a shared region has to be delineated with a *rgn_start_read/write* and *rgn_end_read/write*. The region size was in general selected based on data locality hints in the original SPLASH-2 code. However in some cases, it was found that the original code could give rise to deadlocks. This happens when two nodes are both updating parts of regions owned by them, using valid data from the other node. Once each node has indicated that it wants to write to its region, the other node cannot obtain that region for reading. This problem was solved by reading the data into a temporary area, and then doing the update.

6.4.4 Program Parameters

The programs and the parameters used for this part are shown in Table 6.4. The program sizes had to be altered in some cases since some of the DSM systems could not support larger datasets.

Program	Parameters
FFT	1048576 complex doubles
LU-c	512 $\times$ 512 doubles, block size 16
OCEAN-c	258 $\times$ 258 grid
WATER-n2	1728 molecules, 3 steps
WATER-sp	1000 molecules, 3 steps
MATMULT	1024 $\times$ 1024 doubles, 16 blocks

Table 6.4: Program Parameters for DSM Comparison

In the case of CRL and Quarks, each task corresponds to a separate process, since they do not support multiple application threads per process. CVM and *Strings* used a single process per node with multiple application threads. The tasks distribution for the systems is shown in Table 6.5.

Total Tasks	Nodes Used		Processes per Node		Tasks per Process	
	CRL Quarks	CVM Strings	CRL Quarks	CVM Strings	CRL Quarks	CVM Strings
1	1	1	1	1	1	1
2	2	1	1	1	1	2
4	4	1	1	1	1	4
8	2	2	4	1	1	4
16	4	4	4	1	1	4

Table 6.5: Task Distribution for DSM Comparison

6.4.5 Experimental Results

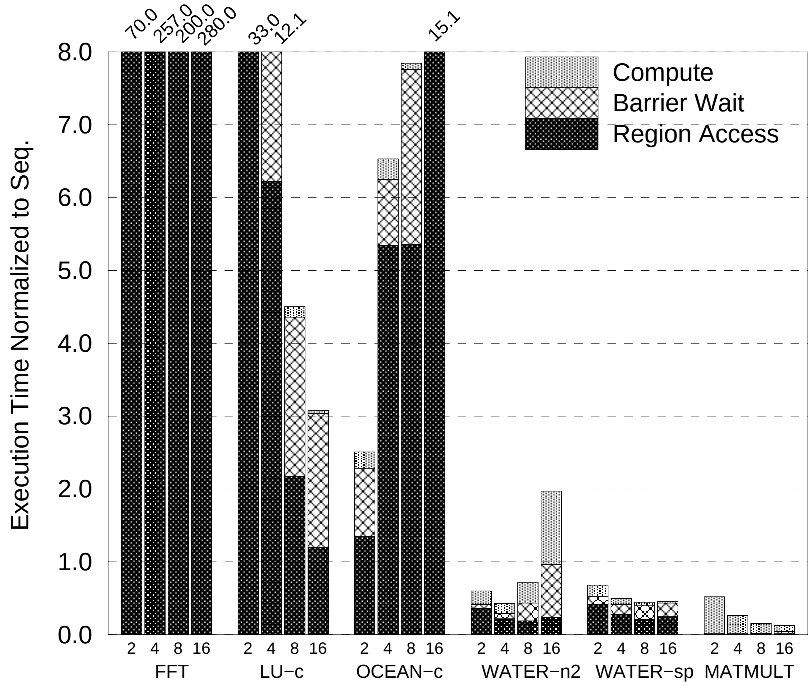

Figure 6.6: Performance Breakup for CRL

The time spent per task by the CRL version of the benchmark programs on computation, barrier waits and region accesses is shown in Figure 6.6. FFT uses a very large region size and hence shows a slowdown compared to the sequential case. The computation is almost sequential, with a large time spent contending for access to the region. Similar effects can be seen for OCEAN-c. LU-c illustrates the problem of selecting too small a region size. Though the relative performance improves when more tasks are used, the overall performance is poor due to the high overhead of acquiring the regions. MATMULT treats each row of a matrix as a region, since there is no false sharing or contention, the overall performance is good. The program is sufficiently large to appear completely compute bound in the split up graph. WATER-n2 shows a slow down when 16 processors are used due to a greater load imbalance.

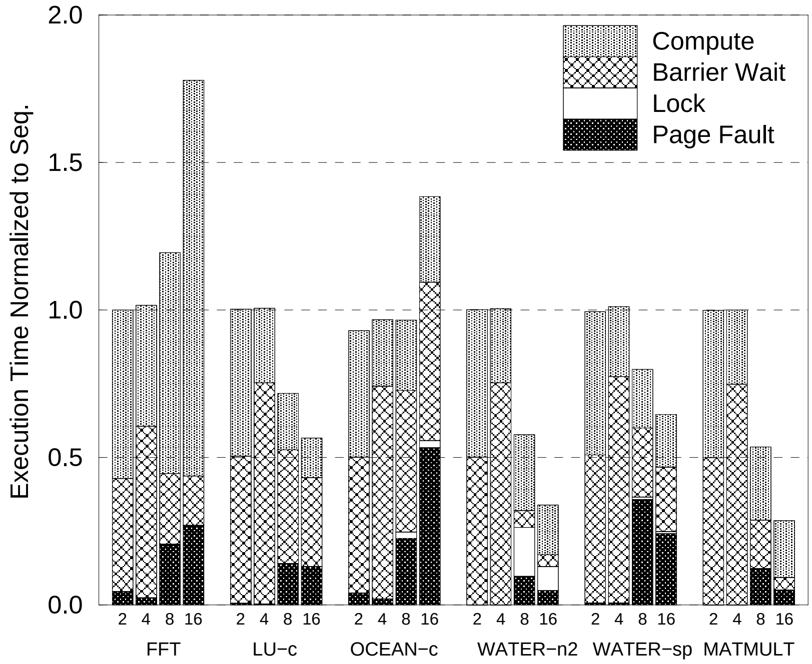

Figure 6.7: Performance Breakup for CVM

The CVM breakup in Figure 6.7 indicates that using multiple user level application threads on a node does not provide any performance improvement. The reduced compute time in the 4 task case is compensated for by the increase in the barrier wait time when compared to the 2 task case. For short running programs like FFT and OCEAN-c, the startup costs increase as the number of nodes is increased, and the time spent on paging related activities also goes up. CVM allows out-of-order access to locks, and uses a centralized lock manager. Since the consistency model requires sending write invalidates only to the next lock acquirer, the overall effect is seen as a very small time for locking, when compared to other DSMs. For LU-c and WATER-n2, the relative performance improves as the number of nodes is increased, but the absolute performance is not as good as that of *Strings*.

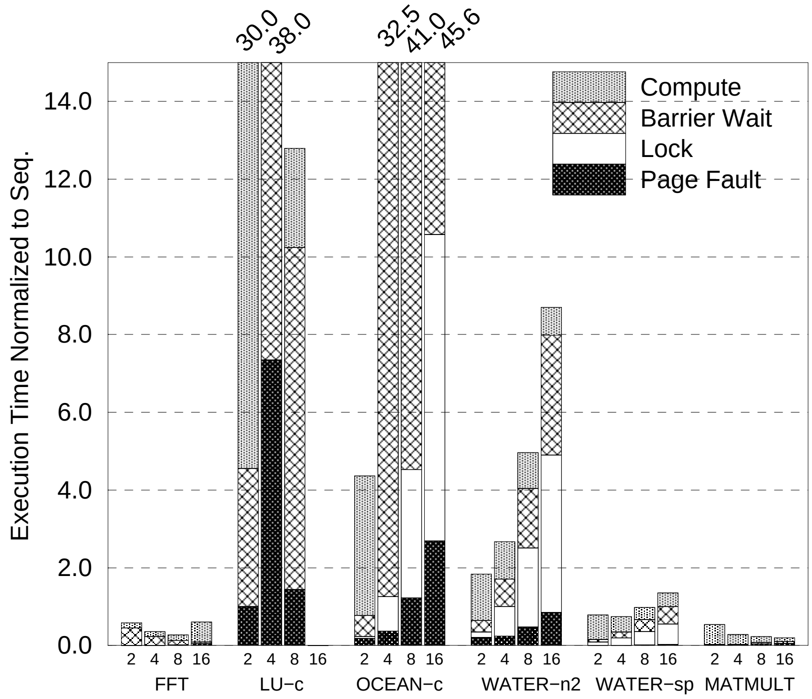

Figure 6.8: Performance Breakup for Quarks

Figure 6.8 shows the performance split-up for Quarks. There are excessive network collisions in the case of LU-c, as one goes from using 2 nodes to 4 nodes. The performance is so bad that it was not possible to obtain results with 16 tasks for this program. OCEAN-c, WATER-n2, and WATER-sp indicate that there is a large cost associated with lock related traffic. CVM shows much better results for these programs, since multiple requests on the same node preempt requests from other nodes. Quarks on the other hand tries to implement a very fair scheme, maintaining a distributed FIFO queue. The release consistency model also requires that all modified pages are flushed to all nodes before a lock can be released. Similarly to CRL, the barrier costs for Quarks are very high. CVM optimizes barriers by collecting arrivals from all the local threads in a process before passing on the notice to the barrier manager.

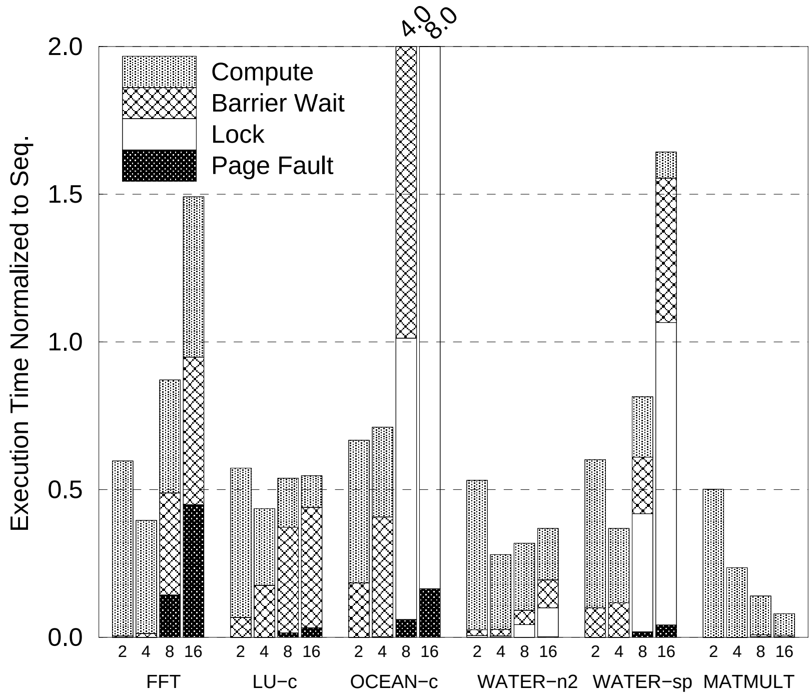

Figure 6.9: Performance Breakup for *Strings*

The *Strings* performance is shown in Figure 6.9. It can be seen that, though the overall performance is good, in case of OCEAN-c and WATER-sp, the lock time forms a large component of the total execution time. *Strings* uses the same algorithm for lock management as Quarks, hence suffers from similar problems. Release consistency requires that updates are flushed to all nodes that share pages, at lock release time. The advantage of using multiple kernel threads is seen in compute bound programs like MATMULT, LU-c and WATER-n2. The barrier costs are noticeably higher when compared to CVM. The current version sends one message per arriving thread to the barrier manager. This part of *Strings* should be optimized further.

Figures 6.10 – 6.15 summarize the overall results with the different DSM systems. It should be noted that the problem sizes selected tend to favor the 4 task version of *Strings* in terms of overall performance. This is to be expected, since this version is closest to the original hardware shared memory implementation.

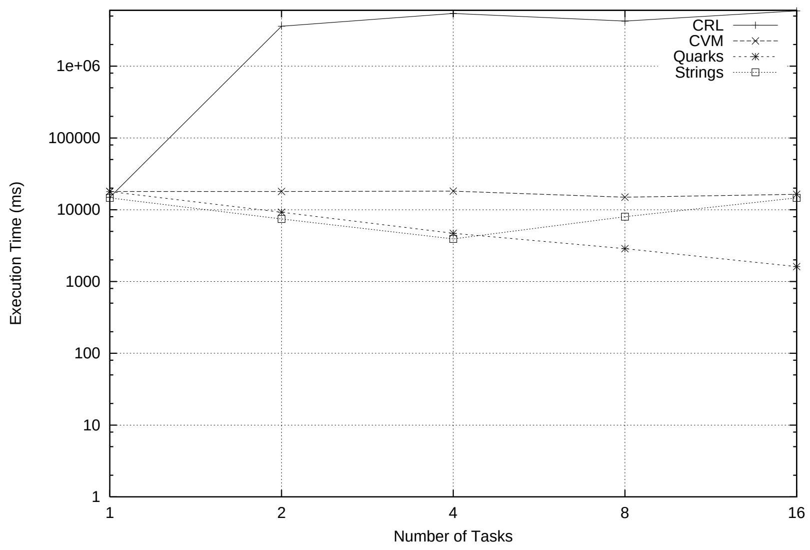

Figure 6.10: FFT results for Different DSMs

6.4.6 Summary

Based on the experiments so far, the performance of the DSMs studied depends to some degree on the characteristics of the application program. *CRL* only performs well for MATMULT, a program with near ideal data sharing and object size. This confirms findings by

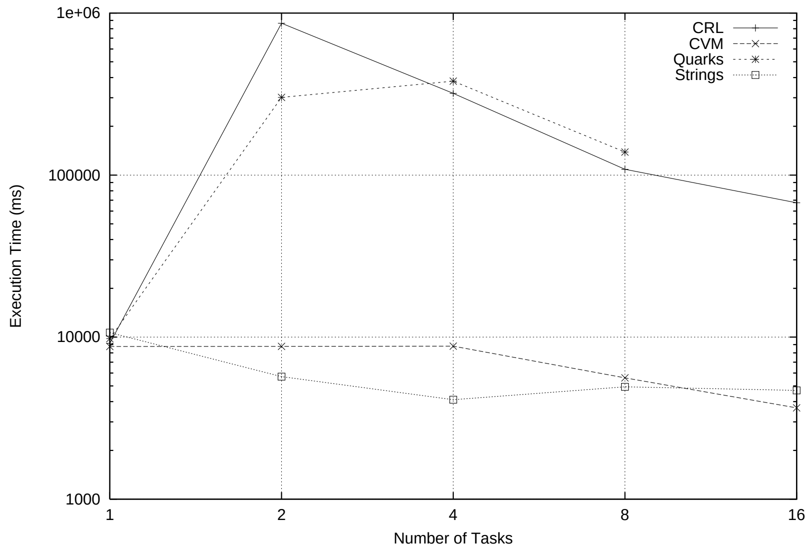

Figure 6.11: LU-c results for Different DSMs

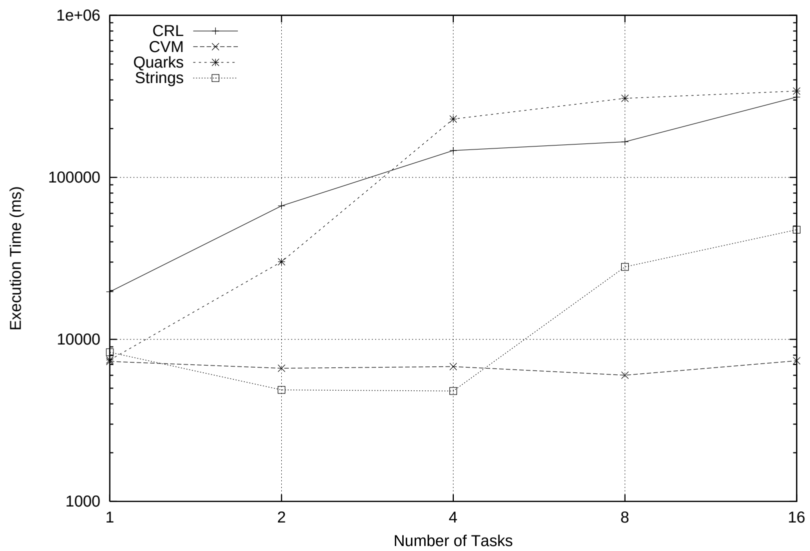

Figure 6.12: OCEAN-c results for Different DSMs

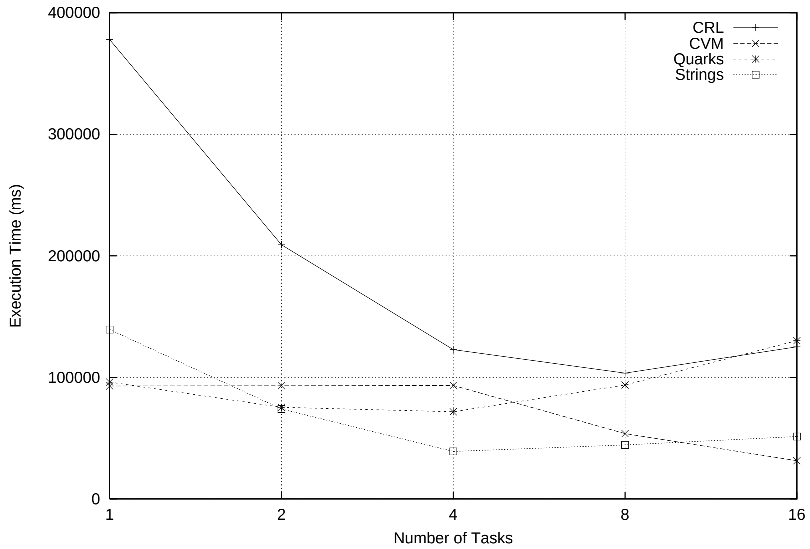

Figure 6.13: WATER-n2 results for Different DSMs

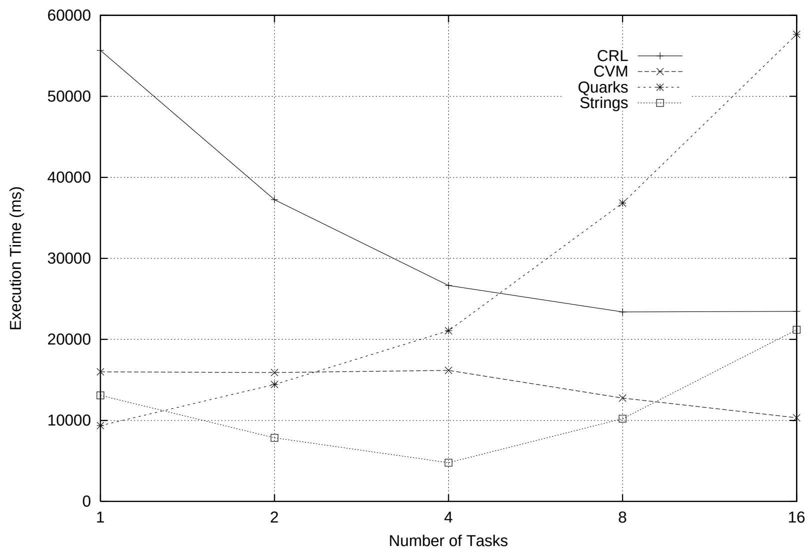

Figure 6.14: WATER-sp results for Different DSMs

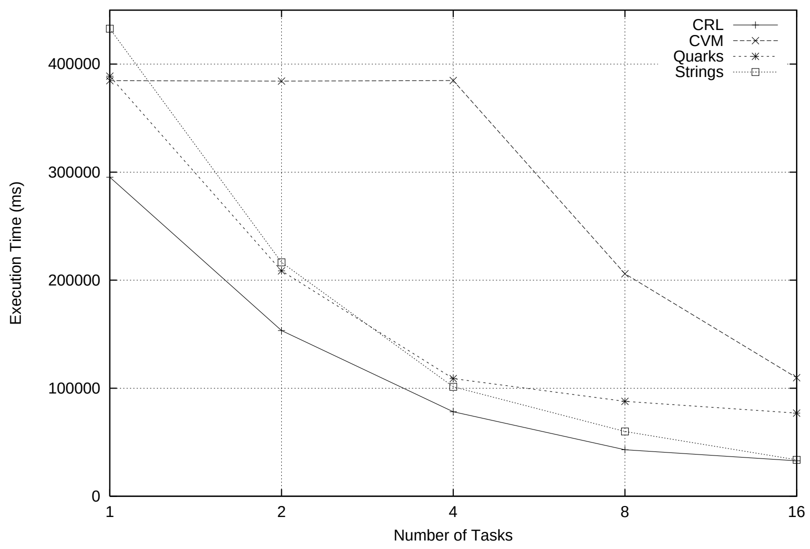

Figure 6.15: MATMULT results for Different DSMs

other researchers that object based DSMs can perform well for such applications. However there seems to be considerable overhead in processing region accesses for *CRL* as seen for FFT, LU-c and OCEAN-c. Though the cost per operation is low when caching is possible, this is more than offset by the large number of calls made to the region access routines. More work needs to be done to identify the ideal object size and the duration for which objects can be safely locked.

Strings has a far smaller page fault cost when compared to *Quarks* and *CVM*, since it has a concurrent server and supports migratory page ownership. For *CVM*, the default 'master process is owner' policy was used. This causes the master process to be interrupted whenever there is a page fault at any node, and increases the event count and overhead.

From the current results, *Strings* is a good system for a broad range of applications. It performs particularly well for applications with a high computation to communication ratio. However the time spent on synchronization events like locks and barriers needs to be improved when compared to *Quarks* and *CVM*.

6.5 Effect of Network Hardware

The final evaluation compares the performance of the *Strings* system on two different networks, a 155 Mbps ATM network with a ForeRunnerLE 155 ATM switch, and a 100 Mbps FastEthernet network with a BayStack FastEthernet Hub.

6.5.1 Program Parameters

The problem sizes used in this evaluation are shown in Table 6.6.

Program	Parameters
FFT	1048576 complex doubles
LU-c	2048 $\times$ 2048 doubles, block size 128
LU-n	1024 $\times$ 1024 doubles, block size 32
RADIX	1048576 integers, radix 1024
OCEAN-c	258 $\times$ 258 grid
WATER-n2	4096 molecules, 3 steps
WATER-sp	4096 molecules, 3 steps
MATMULT	1024 $\times$ 1024 doubles, 16 blocks
SOR	2000 $\times$ 2000 doubles, 20 iterations
MRI	14 frequency points, PHANTOM image

Table 6.6: Problem Sizes for Network Evaluation

6.5.2 Experimental Results

The bandwidth required, the average packet size and the roundtrip latency for the 8 tasks case are shown in Table 6.7. The bandwidth requirement was calculated as the ratio of the total bytes transmitted per task to the lifetime of the task. Though FFT, LU-n and RADIX have the same average bandwidth requirements, and the same average packet size, they have widely differing latencies. This is due to the burstiness of traffic which causes contention at the network interface by multiple threads. For WATER-n2, the packet size is very small, and the setup overhead accumulates, resulting in a large latency. MRI has

Program	100 Mbps Ethernet			155 Mbps ATM		
	BW Mbps	Pkt. Sz. Bytes	Latency ms	BW Mbps	Pkt. Sz. Bytes	Latency ms
FFT	11.95	4916.51	101.65	10.25	4035.91	64.02
LU-c	4.64	5975.64	249.15	4.60	5975.64	241.87
LU-n	10.80	5250.85	45.39	11.98	5224.45	36.41
RADIX	11.85	4666.69	396.29	12.56	4683.39	463.23
OCEAN-c	6.11	3023.52	103.05	6.64	3035.95	76.37
WATER-n2	0.77	197.19	170.82	0.71	198.35	313.30
WATER-sp	0.80	908.15	172.56	0.74	907.17	174.04
MATMULT	2.58	4347.17	8.56	2.52	4346.59	11.67
SOR	6.34	5012.15	275.41	6.26	5012.15	277.62
MRI	4.39	3431.55	1.69	4.39	3416.37	1.62

Table 6.7: Communication Requirements Per Node

a very high computation to communication ratio, and the final result is updated under the protection of a lock, hence it shows the least latency in both cases. MATMULT has similar behavior, but at termination all threads try to update their part of the resultant matrix, which causes the latency to increase due to collisions at the receiving node.

The raw latencies and bandwidth obtainable over UDP using typical packet sizes in the *Strings* runtime are shown in Figures 6.16 and 6.17. The results were obtained using two nodes on an unloaded network, using *netperf-2.1* [66]. Though these values are likely to be optimistic, since they do not include network collisions and congestion at the receiving ports, there appears to be a large overhead introduced by the *Strings* communication subsystem. Further work could be done in this area.

The performance comparisons for the different programs on ATM and FastEthernet are shown in Figures 6.18 – 6.27. Since multiple nodes are used only for the 8 and 16 tasks case, the performance differs only for those runs. WATER-sp and WATER-n2 have a very small average packet size, hence the higher ATM latency causes the execution time for

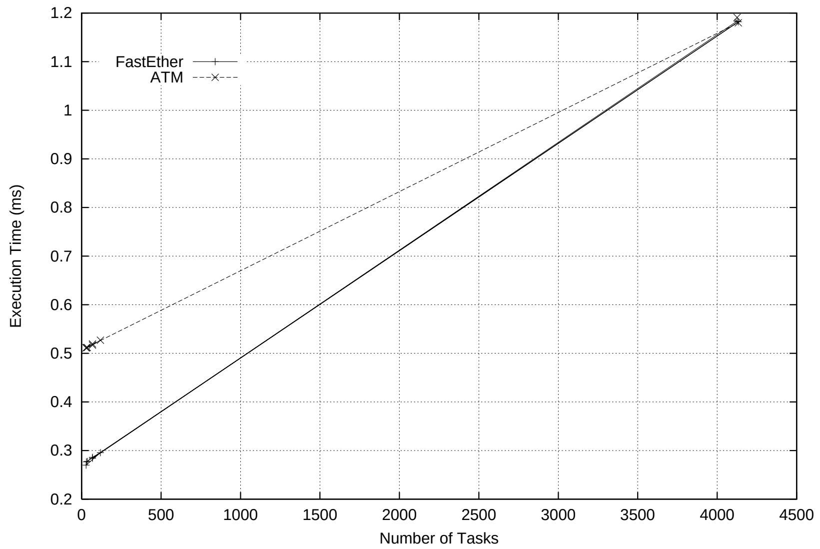

Figure 6.16: Roundtrip times for UDP packets for Different Networks

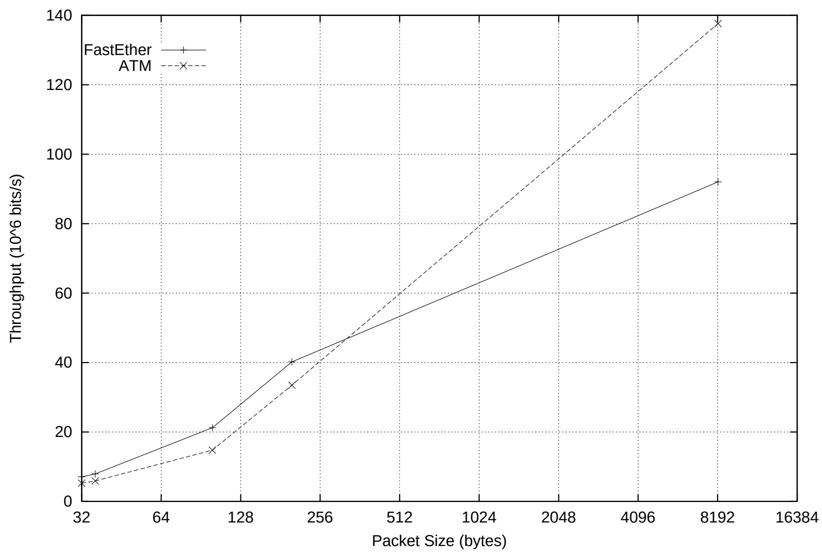

Figure 6.17: Throughput for UDP packets for Different Networks

FastEthernet to be better. The ATM performance is much better for programs that have a high bandwidth requirement, like FFT, OCEAN-c, LU-n, and RADIX. For the other programs there is not much difference in the performance of the two networks.

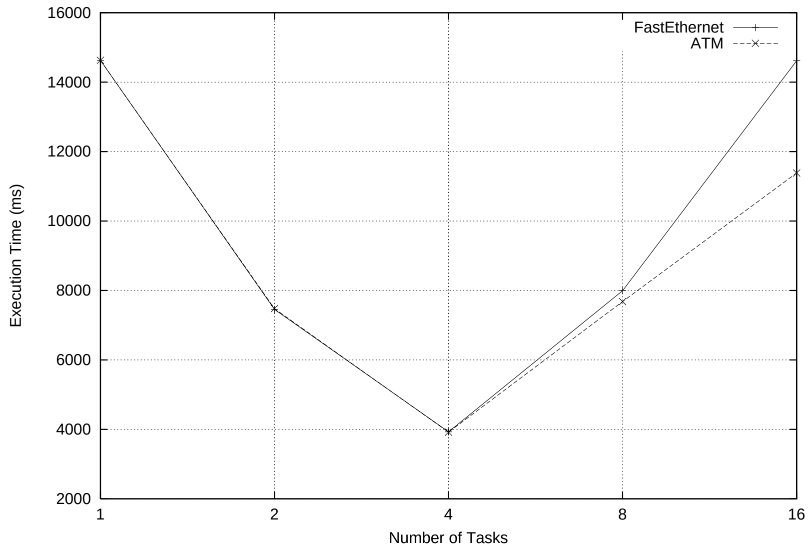

Figure 6.18: FFT results for Different Networks

6.5.3 Summary

Overall the communication results show that bandwidth utilization can be quite high for some applications. In the worst case, if all nodes communicate at the same time, on the same link, the total bandwidth requirement is still within the theoretical limits for both networks. High latencies for small packets, and contention at the network interface are other issues that need to be investigated for improving the performance.

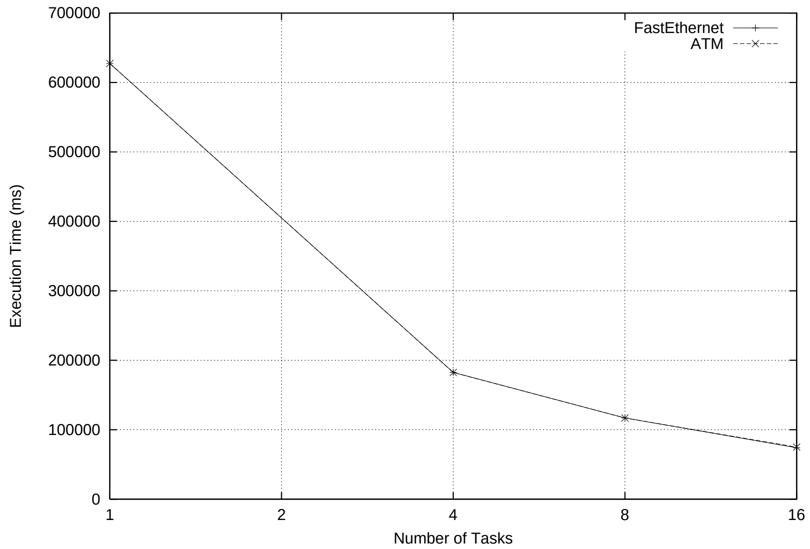

Figure 6.19: LU-c results for Different Networks

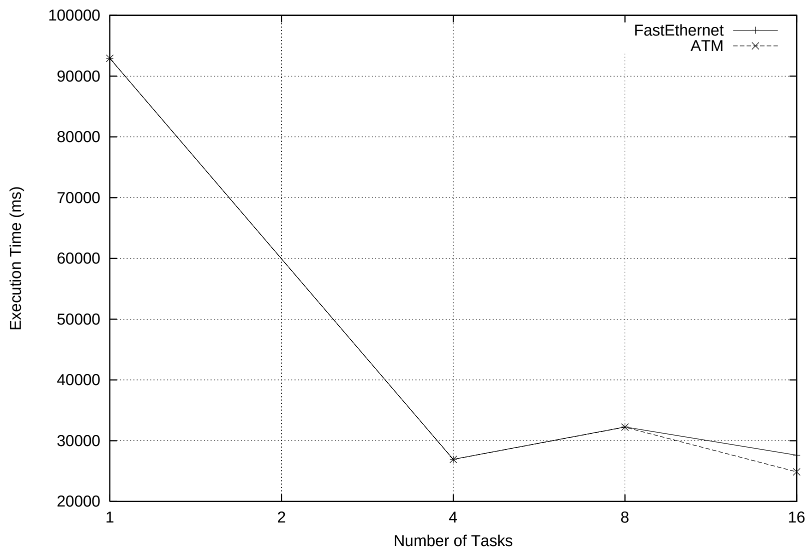

Figure 6.20: LU-n results for Different Networks

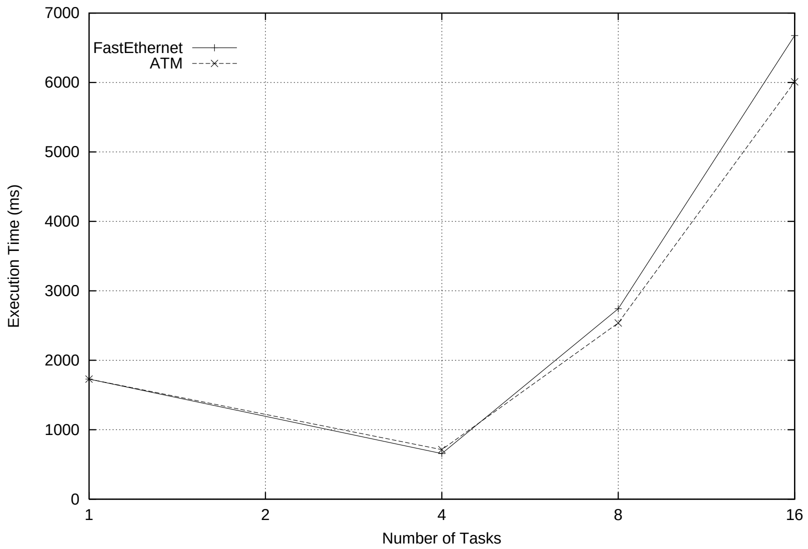

Figure 6.21: RADIX results for Different Networks

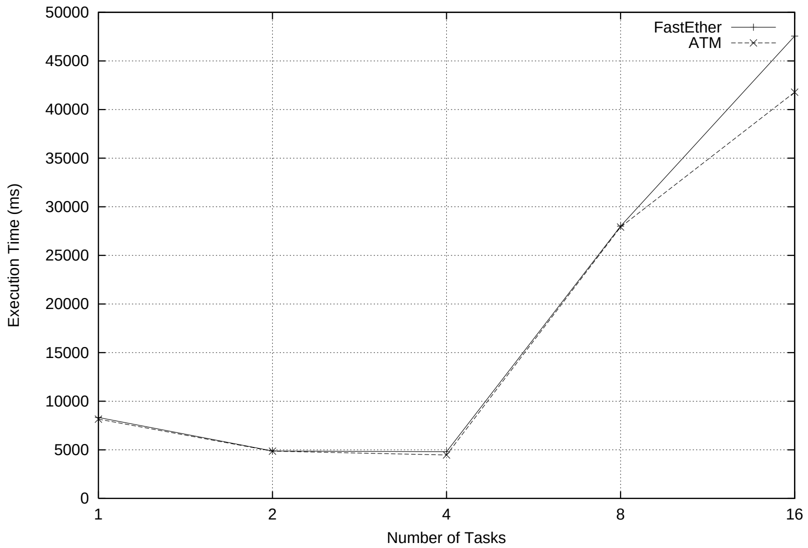

Figure 6.22: OCEAN-c results for Different Networks

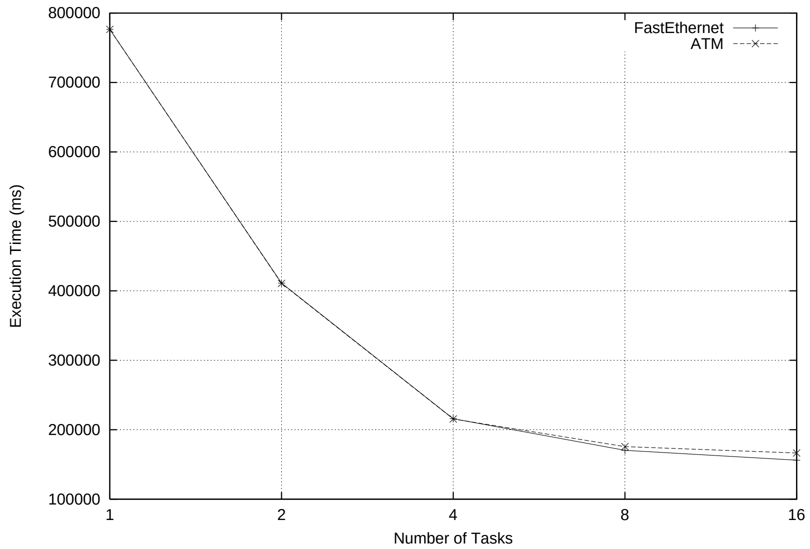

Figure 6.23: WATER-n2 results for Different Networks

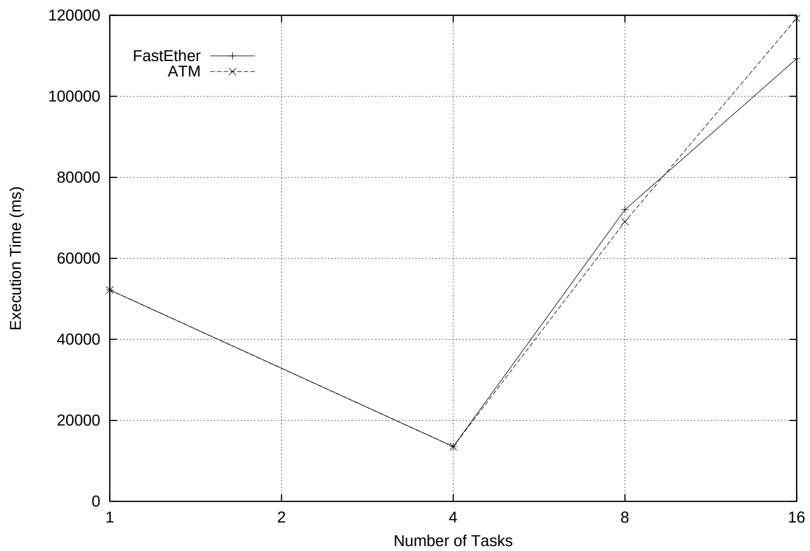

Figure 6.24: WATER-sp results for Different Networks

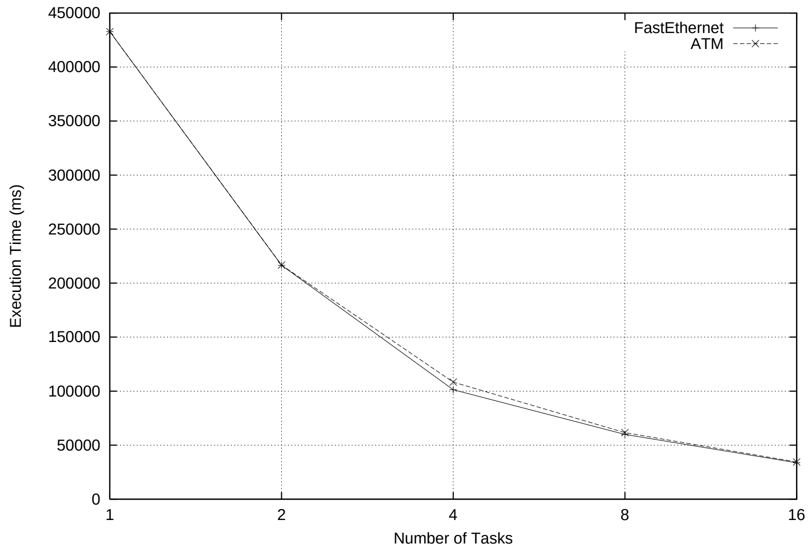

Figure 6.25: MATMULT results for Different Networks

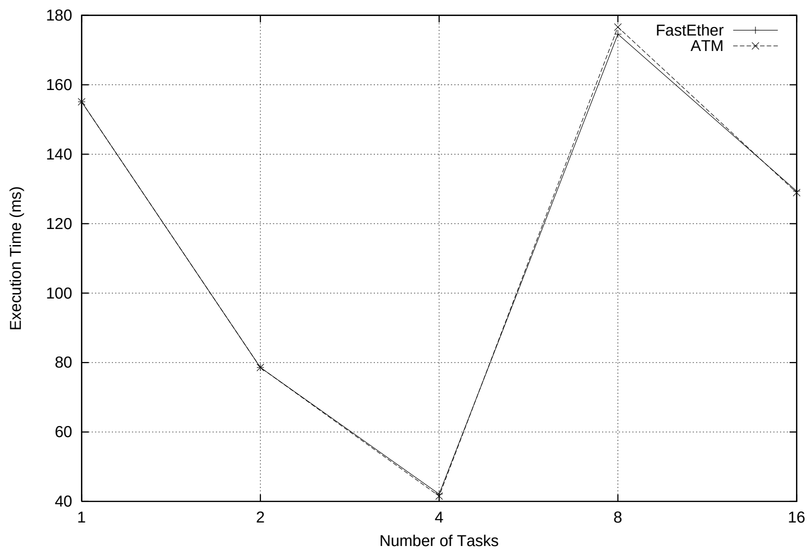

Figure 6.26: SOR results for Different Networks

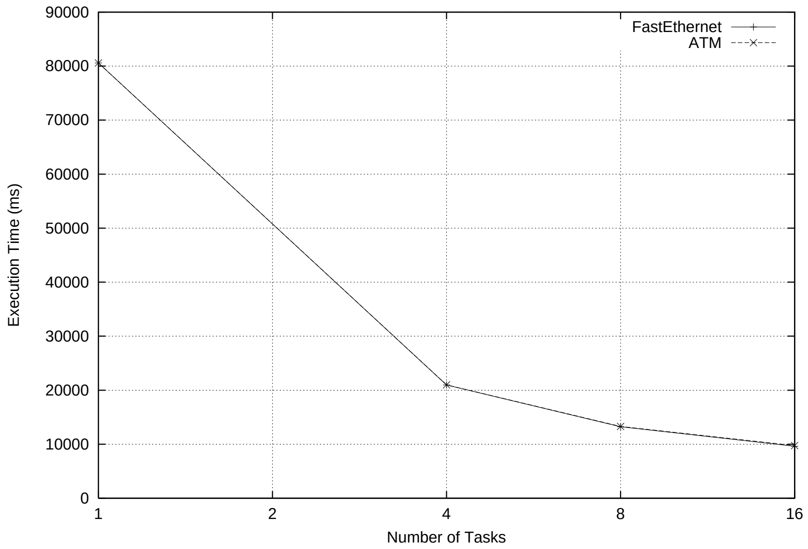

Figure 6.27: MRI results for Different Networks

7. Compiler Interface

This chapter describes the additional pass created so that sequential programs could be directly transformed into *Strings* compliant code, and the changes made to the SUIF runtime.

7.1 Shortcomings of Existing DSM Systems

The SUIF compiler produces multithreaded parallel code for a single Shared Memory Multiprocessor. Hence one should be able to use this code unchanged on a network of workstations if a shared virtual memory or distributed shared memory can be provided completely transparent to the compiler. The present evaluation of CRL, CVM, Quarks, and *Strings* shows that current software DSMs cannot provide such an environment as explained below.

7.1.1 Explicit Shared Memory

Hardware based DSMs and shared memory machines implicitly share all global data, though the implementation may cause non-uniform memory access times for different processes. The current software DSMs require the programmer to explicitly initialize shared data regions. CVM, Quarks, *Strings* and TreadMarks do this by using wrapped calls to the system memory allocator. SAM uses special data types for shared variables. CRL controls access to the shared data areas through the concept of regions. Systems that allow more relaxed memory models also require explicit demarcation of accesses to shared data in the code eg. CRL uses the `rgn_start_write()` and `rgn_end_write()` calls to ensure consistency.

Hence, to integrate SUIF with a DSM system it needs to be extended to include such information in the final output code.

7.1.2 Execution Model

SUIF assumes an initial master thread that first initializes variables and then creates slave threads which handle the parallelized sections of code. The slave threads require a pointer to the parallel function for execution, hence a DSM must provide some method of not only sharing data, but also pointers to functions.

Existing DSMs usually require that all nodes be initialized before creating shared memory regions. Hence the DSM needs to be adapted to the SUIF execution model, or the SUIF output must be modified, to ensure that only the master thread initializes shared regions.

7.1.3 Multithreading

Since the target architecture consists of multiprocessor based nodes, it would be desirable to have multiple threads run concurrently to exploit available processing power. Out of the systems surveyed, only CVM, Quarks and *Strings* provide a multithreaded library. However, CVM and Quarks implement a user level package, which cannot utilize multiple processors on an SMP node. SAM and CRL use PVM for the communication and process control layer. PVM is not thread safe. Multithreaded extensions to PVM exist in the form of TPVM (a threaded PVM) [52] and LPVM (a light-weight PVM for SMPs) [53] but neither can be used on multiple SMPs.

It has also been proposed that message passing applications can benefit from multithreading, by relieving the programmer from having to overlap communication and computation by hand [51].

7.1.4 Lightweight Implementation

Since this project assumes a DSM as a compilation target, it may be possible to remove parts of some existing system that are essentially provided for the shared memory programmer. This could eventually lead to a more efficient implementation and improve performance. SAM already implements such a scheme by encouraging the use of the JADE language and compiler for its use, rather than direct library calls [49]. The SUIF compiler does not use condition variables, which can hence be removed from the target DSM. A possible performance improvement used by CVM is to add specific support for commonly used reduction operations, which can also be detected by the compiler.

7.2 *suif2strings* Pass

The problem in the SUIF runtime model is that it assumes that all global data is implicitly available to all the threads in a program. Some SUIF programs also pass pointers to automatic variables to the parallel routines, which have to be converted to global variables. Hence a two phase *suif2strings* pass was written to solve these problems.

7.2.1 Promoting Automatic Variables

The first phase of the pass is executed after the code generation stage of the parallelizer, ie. *pgen*. The pass iterates through the functions in the file, looking for automatic variables that need to be promoted. If an automatic variable is found that is passed by reference to another function, it is promoted to file scope. SUIF does not support nested parallelism, which means that a function that is executed in parallel can never call another parallel function. Hence analysis of these parallel functions is skipped.

7.2.2 Conversion to Pointers

The second phase converts every user defined global variable, including the ones promoted in phase one, to an explicitly allocated variable. SUIF has built-in mechanisms to prevent name space collisions at this stage. The new and old variables are then entered into a *replacement* list. Array variables are converted into a pointer to the base element that they contain, other variables are matched to pointers to their respective type. The pass then goes through each non-parallel function in the program and replaces the old symbol with the appropriately dereferenced new symbol in every instruction, by looking at the *replacement* list.

A new function, *_suif_memory_init()*, is then created. This function does the actual memory allocation for each of the new global elements. If some of the old symbols were statically initialized, the definitions are copied into the new variables. The *_suif_memory_init()* is then written out to the file with the transformed code.

7.2.3 Example Code

Figure 7.1 shows parts of the original SUIF parallelized code for a Fortran Matrix Multiplication Program. The problematic variables are the index variable `i` and the three arrays `a`, `b`, and `c`, whose addresses are passed via the `_MAIN___1_struct_ptr`. Since these are automatic variables in the scope of function `MAIN__()`, their addresses cannot typically be resolved on remote nodes in a DSM system.

Figure 7.2 shows part of the output of the matrix multiplication program when passed through the additional *suif2strings* pass. The variables `i`, `a`, `b`, and `c`, were first promoted to global file scope by phase one. Phase two created new pointer variables `_ptr_i`, `_ptr_a`, `_ptr_b`, and `_ptr_c`. These are then used as replacements in the original `MAIN__()` function. The replacement must be done in the original sequential code and the assignment to the pointer `_MAIN___1_struct_ptr` assignment. The actual memory for these variables is then allocated by a call to `_suif_memory_init()`.

The current pass is conservative since it promotes all referenced automatics, and converts all user defined global variables to pointers. A possible improvement of this pass

```
extern int *volatile _suif_aligned_my_nprocs;
extern void *volatile _suif_aligned_args;
extern void (**volatile _suif_aligned_task_f)(int);
struct _BLDR_struct_000 { ... };
struct _BLDR_struct_001 { int *_field_0;
  float ((*_field_1)[1024])[1024];
  float ((*_field_2)[1024])[1024];
  float ((*_field_3)[1024])[1024]; };

void _MAIN___0_func(int _my_id){ ... }

void _MAIN___1_func(int _my_id)
{
  int *i, _my_nprocs, j, __priv_i0, k, j_tile4, k_tile5;
  float ((*c)[1024])[1024], ((*a)[1024])[1024], ((*b)[1024])[1024];
  struct _BLDR_struct_001 *_my_struct_ptr;

  _my_struct_ptr = _suif_aligned_args;
  i = _my_struct_ptr->_field_0;
  c = _my_struct_ptr->_field_1;
  a = _my_struct_ptr->_field_2;
  b = _my_struct_ptr->_field_3;
  _my_nprocs = *_suif_aligned_my_nprocs;
  ...
  if (_my_id == _my_nprocs - 1)
    *i = __priv_i0;

}

int MAIN__()
{
  float (a[1024])[1024], (b[1024])[1024], (c[1024])[1024];
  int i,j, k,
  int _doall_level_result = suif_doall_level();

  if (0 < _doall_level_result) {
    for (i = 0; i < 1024; i++){
      *_ptr_i = i;
      for (j = 0; j < 1024; j++)
        for (k = 0; k < 1024; k++)
          c[j][i] = c[j][i] + a[k][i] * b[j][k];
    }
  }else{
     *_suif_aligned_task_f = _MAIN___1_func;
     _MAIN___1_struct_ptr = _suif_aligned_args;
     _MAIN___1_struct_ptr->_field_0 = &i;
     _MAIN___1_struct_ptr->_field_1 = (float ((*)[1024])[1024])c;
     _MAIN___1_struct_ptr->_field_2 = (float ((*)[1024])[1024])a;
     _MAIN___1_struct_ptr->_field_3 = (float ((*)[1024])[1024])b;
     suif_named_doall(_MAIN___1_func, "_MAIN___1_func", 1, 1024, 1);
  }
}
```

Figure 7.1: SUIF output of Matrix Multiplication (edited for clarity)

```
struct _BLDR_struct_001 { ... };
int i;
float (*_ptr_a)[1024], (*_ptr_b)[1024], (*_ptr_c)[1024];
int *_ptr_i;
void _MAIN___1_func(int _my_id)
{
  int *i, _my_nprocs, j, __priv_i0, k, j_tile4, k_tile5;
  float ((*c)[1024])[1024], ((*a)[1024])[1024], ((*b)[1024])[1024];
  struct _BLDR_struct_001 *_my_struct_ptr;

  _my_struct_ptr = _suif_aligned_args;
  i = _my_struct_ptr->_field_0;
  c = _my_struct_ptr->_field_1;
  a = _my_struct_ptr->_field_2;
  b = _my_struct_ptr->_field_3;
  _my_nprocs = *_suif_aligned_my_nprocs;
  ...
  if (_my_id == _my_nprocs - 1)
    *i = __priv_i0;
}

extern int MAIN__()
{
  int j, k;
  int _doall_level_result = suif_doall_level();
  if (0 < _doall_level_result){
    for (i = 1; i <= 1024; i++){
      *_ptr_i = i;
      for (j = 1; j <= 1024; j++){
        for (k = 1; k <= 1024; k++){
          _ptr_c[j - 1][*_ptr_i - 1] = _ptr_c[j - 1][*_ptr_i - 1]
            + _ptr_a[k - 1][*_ptr_i - 1] * _ptr_b[j - 1][k - 1];
        }
      }
    }
  }else{
    struct _BLDR_struct_001 *_MAIN___1_struct_ptr;
    *_suif_aligned_task_f = _MAIN___1_func;
    _MAIN___1_struct_ptr = _suif_aligned_args;
    _MAIN___1_struct_ptr->_field_0 = _ptr_i;
    _MAIN___1_struct_ptr->_field_1 = _ptr_c;
    _MAIN___1_struct_ptr->_field_2 = _ptr_a;
    _MAIN___1_struct_ptr->_field_3 = _ptr_b;
    suif_named_doall(_MAIN___1_func, "_MAIN___1_func", 1, 1024, 1);
  }
}

extern void _suif_memory_init(void)
{
  Qk_create_region(0, 4, &_ptr_i);
  Qk_create_region(0, 4194304, &_ptr_c);
  Qk_create_region(0, 4194304, &_ptr_b);
  Qk_create_region(0, 4194304, &_ptr_a);
}
```

Figure 7.2: *suif2strings* Matrix Multiplication (edited for clarity)

would be to ignore variables that are not used by the parallel routines in the program. This would reduce the amount of globally shared memory required. However, such variables could never cause access misses on remote nodes, since they are not shared, so there would not be a direct benefit in execution time.

7.3 Runtime Enhancements

The SUIF runtime was changed to use the *Strings* API for all process creation and synchronization. The data structures that specify the number of processors, and the parallel task data arena were converted to explicitly allocated memory. A call to the explicit memory allocator *_suif_memory_init()* was inserted before the thread creation call.

7.4 Performance Results

The performance results shown in Table 7.1 are obtained with a dummy version of the runtime. This dummy version allocates the globally shared regions using *mmap()* and then uses threads for the execution. The objective was to remove effects due to communication and isolate the cost of doing explicit allocation.

The programs used for this evaluation are members of the NAS Parallel Benchmarks 2-serial suite [27]. These programs are sequential versions of programs representing Computational Fluid Dynamics code. Each of the parallel runs was carried out on a 4 processor UltraEnterprise 3000. For comparison purposes, the results with the native Fortran compiler, and the parallelizing Fortran compiler are also provided. The *-parallel* flag increases the optimization level of the Fortan compiler, hence the same optimization level was specified to the backend C compiler for the *SUIF* and *suif2strings* versions of the code. The SUIF compilation system is able to obtain a speedup in all cases, and is better than the native compiler in almost 50 % of the programs. The *suif2strings* version also shows some

Benchmark	F77 (s)	F77 -parallel (s)	SUIF (s)	*suif2strings* (s)
BT	25.54	4.39	9.98	13.26
CG	7.84	0.70	0.64	0.72
EP	53.26	45.83	25.28	41.86
FT	27.77	7.37	15.78	18.23
LU	17.18	8.63	4.62	5.63
MG	3.03	0.23	0.66	0.70
SP	14.16	1.86	2.04	2.88

Table 7.1: Compiler Performance on NAS Parallel Benchmarks-2.3

performance improvement, however it suffers from a less optimal barrier implementation in the runtime compared to *SUIF*.

Figure 7.3 shows the results of the compiler applied to the CFD kernels from the Army Research Labs, using a varying number of tasks. The barrier cost is seen to be a dominant factor in the overall execution time, though the actual computational component reduces almost linearly with an increasing number of tasks. Similar results were observed with the hand converted shared memory code from the SPLASH-2 benchmark suites.

For the NAS benchmarks, Figure 7.4 shows the results for two program using a different number of tasks in each run. The compiler generated code causes a large number of synchronization calls, as shown in Table 7.2, for 16 tasks. Since the current performance for these primitives using *Strings* is poor, the overall execution time increases for the parallel case.

Figure 7.5 shows the results for some computational kernels: gaussian elimination of a dense 2048×2048 matrix (GAUSS), cholskey factorization of a 2048×2048 matrix (CHOLSKEY), successive over relaxation of a 4096×2048 grid (SOR), and matrix multiplication of two 1024×1024 matrices (MATMULT). All programs show speed-ups when

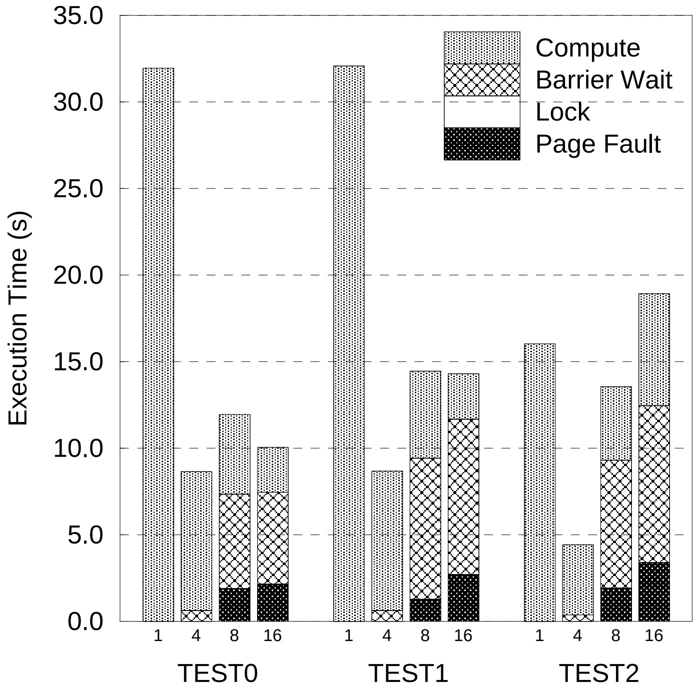

Figure 7.3: Performance for CFD Kernels

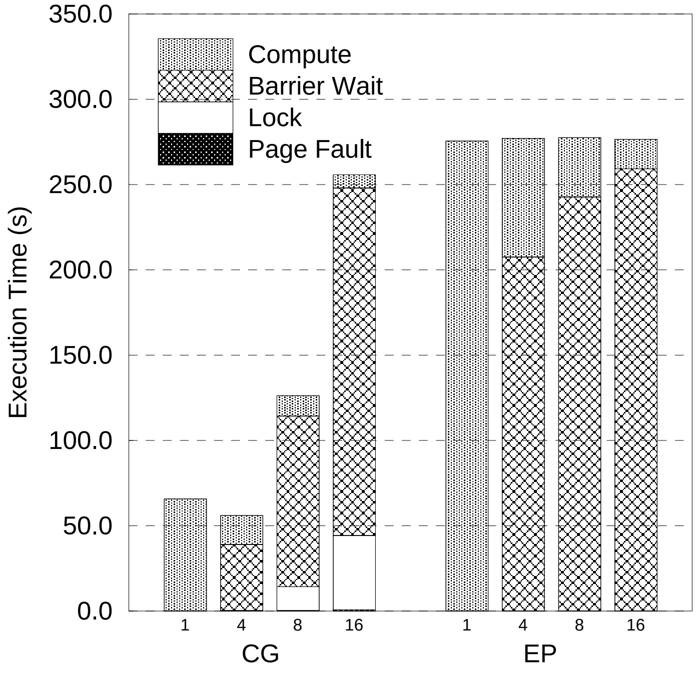

Figure 7.4: Performance for CG and EP

Benchmark		Page	Lock	Barrier
CG	Event Count	120.00	433.00	5837.00
	Time (ms)	759.54	43667.45	203634.96
EP	Event Count	9.25	1.00	11.00
	Time (ms)	97.23	37.51	259009.07

Table 7.2: DSM Overhead Per Thread for Compiler Generated Code

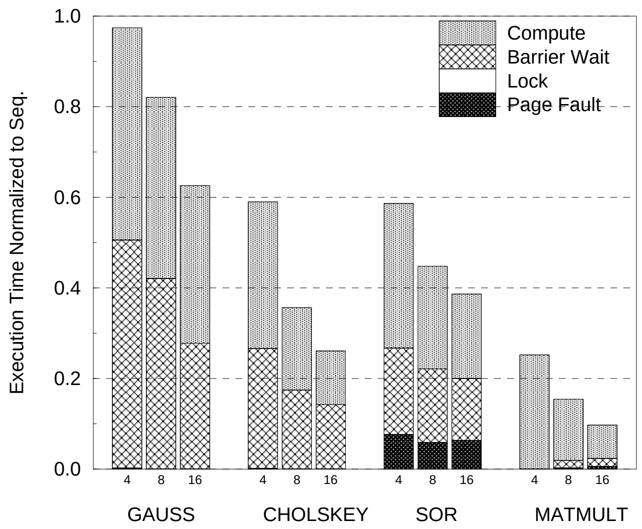

Figure 7.5: Performance for Computational Kernels

compared to the sequential code. The synchronization cost still dominates the overall execution time in the case of GAUSS and CHOLSKEY.

7.5 Related Work

Environments to convert sequential programs for running on distributed memory systems have been proposed earlier eg. Polaris [67] and PARADIGM [68]. Research has tried to improve the performance of data parallel Fortran D programs on the TreadMarks DSM system via compiler involvement [13]. It was found that locks and barriers were a significant source of communication overhead and performance degradation. The key compiler optimizations include optimized section transfers, broadcast, reductions and sparse matrix primitives for incomplete reductions. It was suggested to use the compiler to divide shared pages into those that can be handled via an invalidate protocol and those that require an update protocol. Regular section descriptors were used to introduce prefetching, and thus reduce the overhead due to interrupts at the remote node. Since *Strings* uses polled I/O and dedicated *DSM_server* thread, this overhead is not incurred.

Compiler involvement was also used to solve irregular problems with TreadMarks [11]. In this approach the indirection array was inspected by the compiler to determine the list of pages that would be accessed, and then those pages were prefetched. The FORGE SPF restructuring compiler was also used to convert annotated F77 code for execution on TreadMarks [65]. Compiler parallelized DSM code was found to outperform compiler generated message passing code by 38 % – 89 % for irregular programs. For regular codes, the message passing code performed better by 5.5 % – 49 %. The compiler generated DSM code had redundant synchronizations and showed poor locality in some programs. A major improvement was observed by converting the traditional all-to-all barrier to a master-slave style. Data elements, like loop control variables were sent together with the barrier release message, instead of being faulted in.

Compile time analysis of the data access pattern was also used to aggregate communication, combine data and synchronization messages, and replace global synchronization by point-to-point synchronization where possible [12]. The communication aggregation was found to provide the most benefit, together with synchronous data fetching.

Other research has used the SUIF compiler system on CVM programs to combine synchronization and application data with task invocation [60]. Reduction operations were improved by eliminating synchronization and piggyback messages. Pre-sending data at barriers was found to be another source of major performance improvement.

Thus most of the previous work has addressed the problem of improving the performance of DSM code using a compiler. The current work solves a somewhat different problem, converting parallel code to actual DSM code. It would be beneficial to incorporate some of the above approaches as additional passes to improve the overall performance of the system.

8. Conclusion

This work has successfully integrated the SUIF compiler with a Distributed Shared Memory (DSM) system. It forms a portable parallelization environment, that can convert sequential Fortran or C programs to a parallel version that runs on a cluster of Symmetrical Multiprocessors (SMPs), as shown in Figure 8.1. This system can also serve as a test-bed for future work on the Automatic Parallelization Environment (APE).

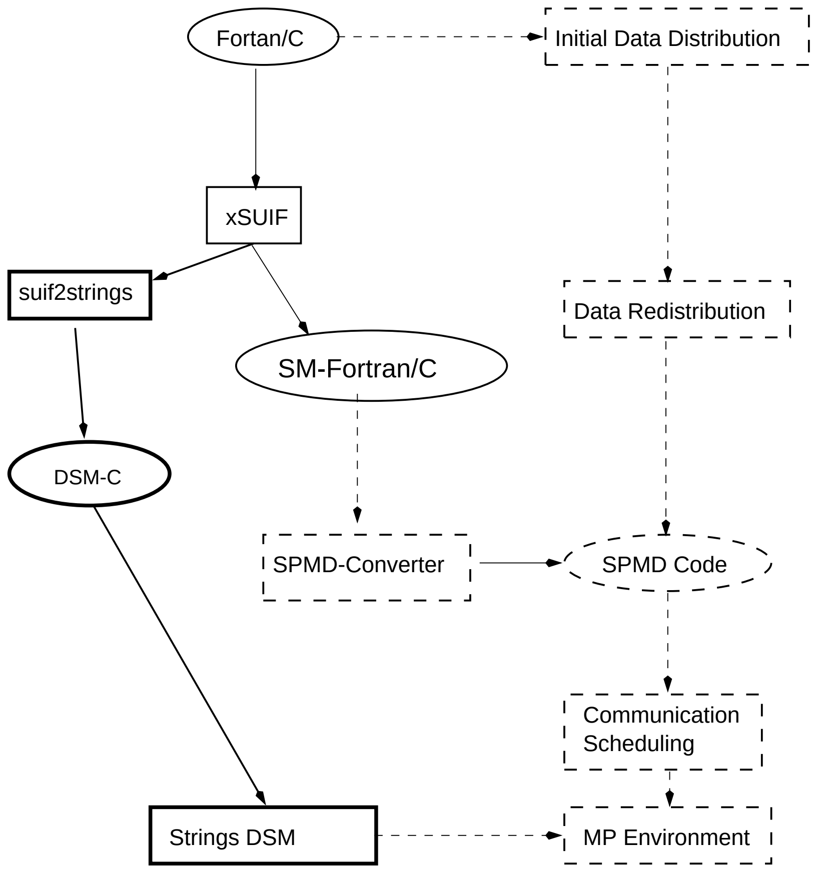

Figure 8.1: Current Status of APE

The newly developed DSM system *Strings* uses multithreading at the kernel level to exploit the computational resources of a cluster of SMPs. The performance of the system was evaluated for the effects of different design decisions, ie. multiple application threads, multithreaded runtime, use of kernel threads and polled I/O. In each case it was found that the current design provides the best performance. The system was also compared to other existing DSMs, namely CRL, CVM, and Quarks, using benchmark programs, computational kernels as well as complete applications. For many of these test programs *Strings* can be seen to provide better performance, particularly for those programs that have a high computation to communication ratio. Lastly *Strings* was also evaluated on two different networks and it was shown that the programs tested are more bandwidth tolerant than they are latency tolerant.

The SUIF parallelizing compiler was extended using the *suif2strings* pass. This pass converts implicit shared memory code to explicit shared memory code for execution on a DSM environment. Preliminary results show that this system works well for computational kernels.

Future work in this area can proceed in two orthogonal directions. One part could address performance issues in the current runtime system. Experiences with other DSM systems have suggested that the Release consistency model can be inefficient and generate excess traffic. Hence other consistency models could be tested. The communication subsystem also requires improvement, since the round-trip latencies are very high for most of the programs tested. A possible approach would be to reorganize the message buffer management system and use a zero-copy approach. For some larger programs, load imbalance across the nodes leads to high wait times at barriers. This could be avoided by using thread migration to move computations to lightly loaded nodes.

The other part could investigate issues in the compiler performance, and the problem with incorrect code generation. The current compiler interface to the DSM is somewhat simplistic, but functional. In particular, it does not use any capabilities of the compiler to actively improve the performance of a program parallelized on a DSM. It may be possible to reduce the amount of synchronization required in a parallelized program by adding more detailed analysis to the compiler suite. Another interesting aspect would be to look at different data layout patterns using the compiler. The current minimum allocation unit in the DSM is very large, and it may be beneficial to implement a memory management scheme for better utilization and performance.

Bibliography

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C.-W. Tseng, "The SUIF Compiler for Scalable Parallel Machines," in *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, February 1995.

[2] Kuck & Associates, Inc., Champaign, IL 61820, *KAP User's Guide*, 1988.

[3] C. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten, "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors," in *Proceedings of the International Conference on Parallel Processing*, (St. Charles), pp. 39–48, August 1989.

[4] B. Appelbe and K. Smith, "Determining Transformation Sequences for Loop Parallelization," in *Fifth Workshop on Languages and Compilers for Parallel Computing*, (Yale), August 1992.

[5] High Performance Fortran Forum, http://www.crpc.rice.edu/HPFF/hpf1/hpf-v11/hpf-report.html, *High Performance Fortran Language Specification*, version 1.1 ed., November 10 1994.

[6] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu, "FORTRAN D Language Specification," Technical Report TR90-141, Department of Computer Science, Rice University, Houston, Texas, 1990.

[7] I. Foster and M. Chandy, "Fortran M: A Language for Modular Parallel Programming," preprint mcs-p327-0992, Argonne National Lab., June 1992.

[8] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," in *Proceedings of the International Symposium on Computer Architecture*, (Los Alamitos), pp. 148–159, IEEE, IEEE CS Press, 1990.

[9] S. Frank, "The KSR1: Bridging the Gap Between Shared Memory and MPPs," in *COMPCON 93*, pp. 285–294, CS Press, 1993.

[10] J. A. Clarke, "Emulating Shared Memory to Simplify Distributed-Memory Programming," *IEEE Computational Science and Engineering*, pp. 55 – 62, January–March 1997.

[11] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and

W. Zwaeonopoel, “TreadMarks: Shared Memory Computing on Networks of Workstations,” *IEEE Computer*, pp. 18–28, February 1996.

[12] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel, “An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System,” in *ASPLOS-VII Proceedings*, vol. 24, (Cambridge, Massachusetts), pp. 186–197, ACM, October 1996.

[13] R. Mirchandaney, S. Hiranandani, and A. Sethi, “Improving the Performance of DSM Systems via Compiler Involvement,” in *Proceedings of Supercomputing 1994*, 1994.

[14] V. Chaudhary, C. Xu, S. Roy, J. Ju, V. Sinha, and L. Luo, “Design and Evaluation of an Environment APE for Automatic Parallelization of Programs,” in *Proceedings of the 1996 2nd International Symposium on Parallel Architectures, Algorithms and Networks*, (Beijing, China), pp. 77 – 83, IEEE, June 1996.

[15] V. Chaudhary, J. Ju, L. Luo, S. Roy, V. Sinha, C. Z. Xu, and V. Konda, “Automatic Parallelization of Non-Uniform Dependences,” in *Proceedings of the First SUIF Compiler Workshop*, (Stanford), pp. 148–152, January 1996.

[16] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy, “SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory,” in *ASPLOS-VII Proceedings*, vol. 24, (Cambridge, Massachusetts), pp. 210–220, ACM, October 1996.

[17] J. Holm, A. Lain, and P. Banerjee, “Compilation of Scientific Programs into Multithreaded and Message Driven Computation,” in *Proceedings of the 1994 Scalable High Performance Computing Conference*, 1994.

[18] T. Fahringer, M. Haines, and P. Mehrotra, “On the Utility of Threads for Data Parallel Programming,” in *Proceedings of the International Conference on Supercomputing*, (Barcelona), pp. 51–59, July 1995.

[19] M. Haines and W. Böhm, “An Evaluation of Software Multithreading in a Conventional Distributed Memory Multiprocessor,” in *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*, (Dallas), pp. 106–113, IEEE, December 1993.

[20] T. Brandes and F. Zimmermann, *ADAPTOR - A Transformation Tool for HPF Programs*, ch. Programming Environments for Massively Parallel Distributed System, pp. 91–96. Birkhuser Verlag, April 1994.

[21] V. Sunderam, “PVM: A Framework for Parallel Distributed Computing,” *Concurrency: Practice and Experience*, vol. 3, December 1990.

[22] R. Butler and E. Lusk, "User's Guide to the P4 Parallel Programming System," Tech. Rep. ANL-92/17, Argonne National Laboratory, October 1992.

[23] R. Butler and E. Lusk, "Monitors, Messages, and Clusters: the P4 Parallel Programming System," *Journal of Parallel Computing*, 1993.

[24] R. Hempel, H. C. Hoppe, U. Keller, and W. Krotz, *PARMACS V6.0 Specification. Interface Description*. Pallas GmbH, November 1993.

[25] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott, "New User Interface for Petit and Other Extensions," tech. rep., University of Maryland, 1996.

[26] M. Wolfe, "The Tiny Loop Restructuring Research Tool," in *Proceedings of the International Conference on Parallel Processing*, vol. II, pp. 46–53, August 1991.

[27] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," Tech. Rep. NAS-95-020, NASA Ames Research Center, http://science.nas.nasa.gov/Software/NPB/, December 1995.

[28] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Transactions on Computer Systems*, vol. 7, pp. 321–359, November 1989.

[29] B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE Computer*, pp. 52–60, August 1991.

[30] J. Protić, M. Tomašević, and V. Milutinovoć, "Distributed Shared Memory: Concepts and Systems," *IEEE Parallel and Distributed Technology*, vol. 4, pp. 63–79, Summer 1996.

[31] M. Stumm and S. Zhou, "Algorithms Implementing Distributed Shared Memory," *IEEE Computer*, pp. 54–64, May 1990.

[32] G. Delp, *The Architecture and Implementation of Memnet: A High-Speed Shared Memory Computer Communication Network*. PhD thesis, University of Delaware, Newark, Delaware, 1988.

[33] D. James, "The Scalable Coherent Interface: Scaling to High Performance Systems," in *COMPCON 94*, (Los Alamitos), pp. 64–71, CS Press, 1994.

[34] R. Campbell, N. Islam, P. Madany, and D. Raila, "Designing and Implementing

Choices:an Object-Oriented System in C++," *Communications of the ACM*, September 1993.

[35] U. Ramachandran and M. Khalidi, "An Implementation of Distributed Shared Memory," in *First Workshop Experiences with Building Distributed and Multiprocessor Systems*, (Berkeley), pp. 21–38, USENIX Association, 1989.

[36] B. Fleisch and G. Popek, "Mirage: A Coherent Distributed Shared Memory Design," in *Proceedings of the ACM Symposium on Operating System Principles*, (New York), pp. 211–223, ACM, 1989.

[37] R. Bisiani and M. Ravishankar, "Plus: A Distributed Shared-Memory System," in *Proceedings of the International Symposium on Computer Architecture*, (Los Alamitos), pp. 115–124, IEEE, IEEE CS Press, 1990.

[38] H. E. Bal and A. S. Tannenbaum, "Distributed Programming with Shared Data," in *International Conference on Computer Languages*, pp. 82–91, CS Press, 1988.

[39] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," in *Proceedings of the International Symposium on Computer Architecture*, pp. 15–26, May 1990.

[40] L. Lamport, "How To Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Computer*, vol. 28, pp. 690 – 691, September 1979.

[41] J. Bennett, J. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," in *Proceedings of the ACM Symposium on the Principles and Practice of Parallel Programming*, (New York), pp. 168–176, ACM, ACM Press, 1990.

[42] A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel, "Software vs. Hardware Shared Memory Implementation: A Case Study," in *Proceedings of the International Symposium on Computer Architecture*, pp. 106 – 117, April 1994.

[43] B. N. Bershad, M. J. Zekauskas, and W. Sawdon, "The Midway Distributed Shared Memory System," in *COMPCON 93*, pp. 528–537, CS Press, 1993.

[44] L. Iftode, J. P. Singh, and K. Li, "Scope Consistency: A Bridge between Release Consistency and Entry Consistency," in *The 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1996.

[45] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Adaptive Software Cache Management for Distributed Shared Memory Architectures," in *Proceedings of the International Symposium on Computer Architecture*, pp. 125–134, May 1990.

[46] P. Keleher, *CVM: The Coherent Virtual Machine*. University of Maryland, CVM Version 2.0 ed., July 1997.

[47] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott, "CASHMERE-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network," in *Proceedings of the ACM Symposium on Operating System Principles*, (Saint Manlo, France), October 1997.

[48] D. Khandekar, *Quarks: Portable Distributed Shared Memory on Unix*. Computer Systems Laboratory, University of Utah, beta ed., 1995.

[49] D. J. Scales and M. S. Lam, "An Efficient Shared Memory Layer for Distributed Memory Machines," Tech. Rep. CSL-TR-94-627, Computer Systems Laboratory, Stanford University, 1994.

[50] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach, "CRL: High-Performance All-Software Distributed Shared Memory," in *Proceedings of the ACM Symposium on Operating System Principles*, December 1995.

[51] E. W. Felten and D. McNamee, "Improving the Performance of Message-Passing Applications by Multithreading," in *Proceedings of the Scalable High Performance Computing Conference*, pp. 84–89, April 1992.

[52] A. Ferrari and V. S. Sunderam, "Multiparadigm Distributed Computing with TPVM," Tech. Rep. CSTR-951201, Emory University, Atlanta, Georgia, 1995.

[53] H. Zhou and A. Geist, "LPVM: A Step Towards Multithreaded PVM." http://www.epm.ornl.gov/ zhou/lpvm.ps.

[54] S. Roy and V. Chaudhary, "*Strings:* A High-Performance Distributed Shared Memory for Symmetrical Multiprocessor Clusters," in *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, (Chicago, IL), pp. 90 –97, July 1998.

[55] K. Thitikamol and P. Keleher, "Multi-threading and Remote Latency in Software DSMs," in *Proceedings of the 17th International Conference on Distributed Computing Systems*, 1997.

[56] S.-Y. Park, J. Lee, and S. Hariri, “A Multithreaded Message-Passing System for High Performance Distributed Computing Applications,” in *Proceedings of the IEEE 18th International Conference on Distributed Systems*, 1998.

[57] J. B. Carter, “Design of the Munin Distributed Shared Memory System,” *Journal of Parallel and Distributed Computing*, 1995.

[58] C. Amza, A. Cox, K. Rajamani, and W. Zwaenepoel, “Tradeoffs Between False Sharing and Aggregation in Software Distributed Shared Memory,” in *Proceedings of the ACM Symposium on the Principles and Practice of Parallel Programming*, (Las Vegas), pp. 90 – 99, June 1997.

[59] H. Lu, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepol, “Compiler and Software Distributed Shared Memory Support for Irregular Application,” in *Proceedings of the ACM Symposium on the Principles and Practice of Parallel Programming*, 1997.

[60] P. Keleher and C.-W. Tseng, “Enhancing Software DSM for Compiler-Parallelized Applications,” in *Proceedings of International Parallel Processing Symposium*, August 1997.

[61] S. C. Woo, M. Ohara, E. Torri, J. P. Singh, and A. Gupta, “The SPLASH-2 Programs: Characterization and Methodological Considerations,” in *Proceedings of the International Symposium on Computer Architecture*, pp. 24–36, June 1995.

[62] D. Jiang, H. Shan, and J. P. Singh, “Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-Coherent Multiprocessors,” in *Proceedings of the ACM Symposium on the Principles and Practice of Parallel Programming*, (Las Vegas), pp. 217 – 229, ACM, 1997.

[63] Y. Zhou, L. Iftode, J. P. Singh, K. Li, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood, “Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation,” in *Proceedings of the ACM Symposium on the Principles and Practice of Parallel Programming*, (Las Vegas), pp. 193 – 205, June 1997.

[64] P. Menon, V. Chaudhary, and J. G. Pipe, “Parallel Algorithms for deblurring MR images,” in *Proceedings of ISCA 13th International Conference on Computers and Their Applications*, March 1998.

[65] A. L. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel, “Evaluating the Performance of Software Distributed Shared Memory as a Target for Parallelizing Compilers,” in *Proceedings of International Parallel Processing Symposium*, April 1997.

[66] I. N. Division, *Netperf: A Network Performance Benchmark*. Hewlett-Packard Company, http://www.netperf.org, revision 2.1 ed., February 1996.

[67] D. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford, and K. Faigin, "Polaris: A New-Generation Parallelizing Compiler for MPPs," Tech. Rep. 1306, Univ. of Illinois at Urbana-Champaign, CSRD, June 1993.

[68] P. Banerjee, J. A. Chandy, M. Gupta, E. Hodges, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su., "The PARADIGM Compiler for Distributed-Memory Multicomputers," *IEEE Computer*, vol. 28, pp. 37–47, October 1995.

ABSTRACT

COMPILATION ISSUES FOR DISTRIBUTED SHARED MEMORY ON CLUSTERS OF SYMMETRICAL MULTIPROCESSORS

by

SUMIT ROY

December 1998

Advisor: Dr. Vipin Chaudhary

Major: Computer Engineering

Degree: Doctor of Philosophy

Clusters of Symmetrical Multiprocessor machines are increasingly becoming the norm for high performance computing environments. The Automatic Parallelization Environment (*APE*) has recently been proposed as a new environment for solving large scale scientific and engineering problems on such computing platforms. This work extends current research in parallelizing compilers and runtime environments to convert existing sequential programs for execution on such clusters of Symmetrical Multiprocessors (SMPs).

The first phase of this work addresses the selection of the front-end compiler for *APE*. The SUIF compiler is selected for integration into the system and its performance is evaluated with some benchmark codes, and Computational Fluid Dynamics kernels. The compiler is shown to provide close to linear speedups for some of the kernel code. The compiler currently produces code for a single shared-memory multiprocessor system. The use of a software Distributed Shared Memory (DSM) system is hence suggested in order to provide

a portable target for the compiler.

An evaluation of existing DSMs, SAM, CRL and Quarks shows that they cannot be directly used within the existing framework. A new, multithreaded DSM system, *Strings* is proposed and its performance compared to other existing systems using programs from the SPLASH-2 benchmark suite, some computational kernels and a full application.

The last part focuses on integrating the compiler with the DSM. An additional pass called *suif2strings* is added to the parallelizing SUIF compiler. The overhead added due to this pass is evaluated with the NAS serial benchmarks, and results with some computational kernels are shown. The complete work thus serves as an initial implementation and test-bench for the APE.

Autobiographical Statement

Personal

Born on 10/09/1968 to Debi Prasad Roy and Shibani Roy in Aachen, Germany.

Education

1991	B.Tech. Electrical Engineering	Indian Institute of Technology, Bombay, India
1996	M.S. Computer Engineering	Wayne State University, Detroit, USA
1998	Ph.D. Computer Engineering	Wayne State University, Detroit, USA

Honors and Awards

1993 Thomas C. Rumble Fellowship, Wayne State University.

1994 Graduate Professional Scholarship Award, Wayne State University.

Publications

1. S. Roy and V. Chaudhary, "Strings: A High-Performance Distributed Shared Memory for Symmetrical Multiprocessor Clusters," in *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, July 1998.
2. V. Chaudhary, S. Roy and B. Sabata, "Parallelization of 3-D Range Image Segmentation on a SIMD Multiprocessor," *Proceedings of the ISCA 12th International Conference on Computers and Their Applications*, March 1997.
3. S. Roy and V. Chaudhary, "A New Metric for Processor Allocation Schemes in Multiprocessor Systems," *Proceedings of the International Performance, Computing and Communications Conference*, February 1997.
4. S. Punyamurtula, V. Chaudhary, J. Ju, and S. Roy, "Compile Time Partitioning of Nested Loop Iteration Spaces with Non-Uniform Dependences," in *Journal of Parallel Algorithms and Applications*, December 1996.
5. V. Chaudhary, C. Xu, S. Roy, J. Ju, V. Sinha, and L. Luo, "Design and Evaluation of an Environment APE for Automatic Parallelization of Programs," *Proceedings of the 2nd International Symposium on Parallel Algorithms, Architectures and Networks*, June 1996.
6. V. Chaudhary, J. Ju, L. Luo, S. Roy, V. Sinha, C. Xu, and V. Konda, "Automatic Parallelization of Non-Uniform Dependences," *Proceedings of the First SUIF Compiler Workshop*, California, January 1996.
7. S. Roy, E. Yaprak, J. Liu, E. Chow, and R. Markley, "ATM Switch Modeling and Simulation", *Proceedings of the IASTED International Conference on Networks*, January 1996.
8. E. Yaprak, S. Punyamurtula, S. Roy, and G. Follen, "Performance Study of Clustered Computers," *Proceedings of High Performance Computing 1995*, Phoenix, Arizona, April 1995.

Research Interests

Distributed Shared Memory Systems	Parallelizing Compilers
Interconnection Networks	ATM Network Technology