AUTOMATIC PARALLELIZATION: AN INCREMENTAL, OPTIMISTIC, PRACTICAL APPROACH

by

Naftali Schwartz

a dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
New York University
September, 1999

Zvi Meir Kedem

Acknowledgements

I have had the good fortune to be surrounded by family, friends and colleagues who supplied the ongoing encouragement without which I could not have brought this latest chapter of my life to a close.

Zvi Kedem has been a strong force in my life for the past five years. I had had the misfortune of previously tackling and abandoning several other research areas; Zvi found a problem which matched my interests well, and to which I could totally commit myself all these years. Along the way, he taught me the critical importance of precision and clarity, both in conception and communication of ideas—even going out of his way to teach me the finer points of writing English where necessary. He has taken a strong personal interest both in my health and well being and that of my family. I will always recall fondly his encouragement of my personal development, which he never limited to the areas of our common academic interests. Indeed, he was always prepared to support my ambitions even when these took us very far afield. His consistently level-headed, critical approach has forever changed the way I view the world.

Ever since my thesis topic crystallized, Zvi Kedem, Davi Geiger and Eric Freudenthal were at my side on an ongoing basis, giving me personal attention and guidance way beyond the call of duty. Though my topic was difficult, every one of them believed I would ultimately see it through.

I have been privileged to work with many esteemed faculty members over the course of my stay at NYU. I will always be grateful for the loving patience of Martin Davis, Bud Mishra, Ed Schonberg and Allan Gottlieb who imbued me with a sense of professional responsibility each in his own caring way. Alan Siegel has been a special friend, always ready to humor me. Vadim Kotov, who mentored me at Hewlett-Packard Labs, taught me a great deal about the real world of computing outside of my academic cocoon.

My friends at NYU have always served me in good stead: Fangzhe Chang, Martin Garcia Keller, Ayal Itzkovitz, Fabian Monrose, Peter Wyckoff and Roman Yangarber; but especially Arash Baratloo, who advised me to seek my summer employment, but also knew when I just needed to take it easy and told me so. Anina Karmen has had the patience of a saint in handling my academic affairs.

I have benefited from the wisdom of Richard Cole, Partha Dasgupta, Art Goldberg, Ben Goldberg, Bob Hummel, Krishna Palem, Jack Schwartz and many others here; but I am most grateful to Vijay Karamcheti, who I could always count on for extensive, reasoned and thoughtful feedback on my papers, which he was always prepared to read.

The members of my committee, Zvi, Jack, Ben, Vijay and Art have all contributed to the style and content of this thesis. The keen insight of Michah Lerner of Columbia has also improved it. All remaining errors are mine.

I am grateful for all the support, financial and otherwise, that my parents and in-laws have provided over the course of this work. My father's high expectations have been a source of continuing inspiration, and I am thankful that he could witness my ultimate triumph; I know that my mother, z"l, is watching and beaming as well.

Most of all, I thank my wife, whose kindness, patience, support and wisdom granted me the strength and time to finish.

This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-96-1-0320 and F30602-99-1-0517; by the National Science Foundation under grant number CCR-9411590; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

Abstract

The historic focus of Automatic Parallelization efforts has been limited in two ways. First, parallelization has generally been attempted only on codes which can be proven to be parallelizeable. Unfortunately, the requisite dependence analysis is undecidable, and today's applications demonstrate that this restriction is more than just theoretical. Second, parallel program generation has generally been geared to custom multiprocessing hardware. Although a network of workstations (NOW) could in principle be harnessed to serve as a multiprocessing platform, the NOW has characteristics which are at odds with effective utilization.

This thesis shows that by restricting our attention to the important domain of "embarrassingly parallel" applications, leveraging existing scalable and efficient network services, and carefully orchestrating a synergy between compile-time transformations and a small runtime system, we can achieve a parallelization that not only works in the face of inconclusive program analysis, but is also efficient for the NOW. We optimistically parallelize loops whose memory access behavior is unknown, relying on the runtime system to provide efficient detection and recovery in the case of an overly optimistic transformation. Unlike previous work in speculative parallelization, we provide a methodology which is not tied to the Fortran language, making it feasible as a generally useful approach. Our runtime system implements Two-Phase Idempotent Eager Scheduling (TIES) for efficient network execution, providing an Automatic Parallelization platform with performance scalability for the NOW.

Our transformation divides the original program into a server and zero or more clients. The server program is a specialization of the original application with each parallel loop replaced with a scheduling call to the client which comprises the body of that parallel loop. The scheduler remotely executes the appropriate instances of this client on available machines.

We describe the transformation and runtime system in detail, and report on the automatic transformation achieved by our implementation prototype in two case studies. In each of these cases, we were able to automatically locate the important loops, construct a shared-memory layout, and generate appropriate server and client code. Furthermore, we show that our generated parallel programs achieve near-linear speedups for sufficiently large problem sizes.

Contents

List of Figures

Chapter 1

Introduction

Parallel programming is not currently a matter for the uninitiated or faint of heart. Application design, tuning, debugging and execution in the parallel world are today orders of magnitude more painful and costly than their sequential counterparts, even in application domains which are known to be "easy." This chapter reviews current practice, challenges it, and introduces a cheap, flexible, practical and fully automatic method for extracting parallelism from "embarrassingly parallel" applications.

1.1 Parallel Programming Today

There have been a great many attempts to create languages and tools which facilitate the ease with which parallelism in programs can be detected and expressed. Some have ambitiously sought to build parallelizing compilers for automatic detection and exploitation of parallelism in "dusty deck" Fortran programs. Although the best of these efforts have borne some fruit, the techniques developed for this controlled environment have often been found not to generalize to more modern languages.

Others, finding this vision wildly optimistic, have instead opted to design languages and interfaces which help programmers communicate the parallelism in their applications to the compiler. These take the form of parallel looping constructs, shared data annotations, data distribution annotations, assertions, partial evaluation, and even very elaborate environments which enable a dialog of sorts to take place between the user and the compiler. The inherent problem with this approach is that it depends more or less on the user to have a deep understanding of both the general principles of parallelization (such as Data Dependence) and the specific characteristics of his application in this regard.

In either case, the target platform for the parallelized program is not often within the budget of the average consumer. The generated parallel program is usually targeted to specialty multiprocessing computers, as these are held to be the only choice which can offer significant speedups on automatically parallelized applications.

What is truly unfortunate about all of this is that *we know that many of the programs we want to parallelize are inherently parallel.* That is to say, we know that the algorithms which these programs implement lend themselves to straightforward parallel implementation. For example, a ray tracing program computes the value of each pixel independently of the value of any other pixel. In this and similar cases, the specialty multiprocessor internode communication hardware becomes so much baggage because processors do not stall waiting for data from other processors. Therefore, even the modest communications infrastructure of the lowly NOW should prove adequate to this domain. A more flexible solution using the NOW, then, while not necessarily straightforward, is both practical and desirable.

1.2 A Different Set of Rules

Parallelization should be approached somewhat differently today than it has been in the past. Traditionally, the goal of parallelization has been the optimal utilization of multiprocessor resources acquired at great expense. More recently, it has become possible to dynamically construct parallel platforms out of collections of machines with idle cycles[ACP95]. This kind of platform can potentially provide vast computing power at essentially no cost. The difficulty is in harnessing this power, which requires special programming techniques for overcoming the special challenges which the NOW presents.

In this context, optimality is no longer the main issue. If we could easily harness NOW power at even suboptimal efficiency, this would be a significant improvement.

Effective parallelization under these new assumptions entails the exploitation of highly complex coarse-grained loops which are not in general amenable to static analysis. This requires moving beyond traditional analyses to runtime parallelism verification, where even a potentially expensive runtime technique can be justified under the new rules.

1.3 Challenges of the NOW

The difficulty of parallelizing programs for the NOW is threefold: communication is not cheap, faults and slowdowns are the norm and machines may join and leave the computation at any time. Two-Phase Idempotent Eager Scheduling (TIES)[KPS90] has laid the foundation for a program transformation methodology which enables efficient utilization of the NOW despite these handicaps.

1.3.1 Distributed Shared Memory

The coordination of a parallel program running on a collection of processors requires the communication of data values among the processors. It is generally agreed that this communication takes place most straightforwardly through the use of shared memory. This is easy to arrange on a multiprocessor because every memory location has a unique id in the global address space.

On a NOW, however, a software layer called a Distributed Shared Memory (DSM) must create the illusion of a physical global address space. This involves keeping track of which pages are dirty, which processors hold a writable copy, and various other details. Unfortunately, the high cost of communications on a NOW keeps the performance low as compared with that obtained on a dedicated multiprocessor[JSS97].

1.3.2 Two Phase Idempotent Eager Scheduling

Traditionally, program parallelization begins with the division of a sequential application into a series of alternating serial and parallel steps. Each parallel step is composed of a number of independent threads of control which can be executed simultaneously; for simplicity, we assume these are individual iterations of a parallelizeable loop. While serial steps are executed locally, individual threads of parallel steps are scheduled on available processors. When the last of these threads terminates, the parallel step is itself finished.

This naive but popular approach is sufficient for dedicated multiprocessors where individual processors are uniform and reliable. However, the widely disparate power and load of different nodes on a network coupled with unpredictable slowdowns and failures make this impractical for the NOW, as the running time for a parallel step would always be dictated by the slowest processor. Worse, this might be a machine which has completely failed, in which case the computation would have to time out and be restarted.

The Two Phase Idempotent Eager Scheduling [KPS90] technique was created to address these issues. Instead of assigning each thread to a unique machine, the Eager Scheduling approach simultaneously assigns several copies of individual threads to several machines to guarantee termination as long as at least one continues to progress. The first of these to successfully complete the job has the results committed, while the other machines running the same job are immediately freed up for other work. However, running a job several times can produce incorrect results unless the starting memory state is identical for each copy. To keep the execution of each copy idempotent, or repeatable, this original memory state is preserved until the parallel step successfully concludes.

Therefore, a parallel step is divided into two phases. The first phase involves the division and scheduling of individual threads of the parallel step on the network. All threads receive an identical starting memory image. Only after all threads have run to completion is the global memory state updated in the second phase of the parallel step. This scheme also minimizes communications within individual parallel steps because the complete memory image for each thread is available at the start of a parallel step.

1.3.3 Embarrassingly Parallel Programs

Embarrassingly parallel programs are programs whose execution time is dominated by a small number of coarse-grained parallel loops. Such applications arise in virtually every domain of science, particularly in the context of simulations[Zom96].

It is critical to note that the lack of internode communication requirements within the main loops of these programs is perfectly matched to the execution style of the TIES methodology, in which no inter-iteration communication is offered. Although this loop property is strictly speaking undecidable[Rep99], high-level information about the application may still lead us to suspect that certain loops are both coarse-grained and parallel. We will see that an optimistic approach can help us to extract effective parallelism from these loops.

1.3.4 Calypso

Calypso[Bar99] is a language and a runtime system which implements the TIES methodology. It extends C++ to include keywords which the programmer can use to specify parallel program steps and annotate shared memory. The Calypso preprocessor then transforms the program into valid C++ with library calls for shared data handling and remote execution.

Although the Calypso system goes a long way in easing the development of parallel programs for the NOW, it is not a parallelizing compiler, and as such requires significant effort on the part of the user. It is easy enough to insert annotating keywords, but understanding which parts of an application lend themselves to parallel execution frequently requires a deep understanding of the application as well as an appreciation of the general principles of parallel programming. A loop should not be parallelized unless it represents a coarse grain of the computation, but a misunderstanding in this case merely causes system inefficiency. More critical is the need to understand which loops are legal to parallelize, for a misapplied parallel directive can cause unpredictable failures. Even after a correct parallelization has been chosen, it can be quite difficult to track down every variable which must consequently reside in the shared data segment. Leaving out a needed shared annotation can also cause failure. Similar onerous programmer responsibilities have traditionally kept debugging costs high and public interest low[CPW94].

Moreover, even if a programmer understands precisely which annotations must be made, the mechanics of the transformation process may still prove quite challenging. Since the Calypso system requires that all shared data declarations be contiguous, some declarations will clearly need to be moved around. But arbitrary shuffling of data declarations among source files introduces a host of problems, including:

- Data declarations may depend on type declarations which are scattered throughout header files and/or recursively dependent on other declarations. It can be difficult to localize a complete type definition for a variable, and, having done this, transplant it to a different source file.
- Data initializations that depend on a particular order of variable definitions within a source file may need significant recoding if variable definitions need to be transferred to alternative source files.
- Name clashes may prevent the arbitrary moving of variables between different scopes.

Therefore, although this option can produce efficient programs for widely available multiprocessing platforms, such as the Network of Workstations (NOW), it is not an easy road to travel. Furthermore, the work is never done, as updates to the serial version will have to be periodically reflected in the parallel version. Because this too is time consuming, it is not performed very often except for perhaps the most important and popular of parallel programs. Consequently, it is common to find the parallel version of an application drifting slowly into obsolescence.

1.4 Automatic Parallelization

Since we seek to apply the techniques of Automatic Parallelization to the domain of Distributed Systems, we first discuss the general transformational issues which have been found to arise in the more traditional domain of dedicated multiprocessors.

1.4.1 Parallelizing Compilers

Some of the important automatic parallelizing compiler prototypes which have been built include: McCAT[HDE+93], Panorama[NGL96], Parafrase[Leu90], PFC[AK87], PIPS[CI95], Polaris[PEH+93], PTRAN[ABC+87] and SUIF[TWL+91]. In identifying and exploiting parallelism, these systems generally rely almost exclusively on the static analysis of data dependence relationships present in the original programs.

1.4.2 Data Dependence

Allen et al.[ACK87] describe the general conditions under which particular loops can be automatically transformed to execute individual iterations in parallel. They show that the memory access patterns of individual loop iterations must satisfy certain conditions to admit a parallelization. In particular, if two iterations access the same memory location and at least one of these accesses is a write, we say that a *data dependence* exists, and some synchronization must be performed.

Data dependences fall into three categories, depending on the chronological relationship of the write access:

Flow Dependence An earlier iteration writes a value which is read by a later iteration.

Anti Dependence An earlier iteration reads a value which is written by a later iteration.

Output Dependence An earlier iteration writes a value which is also written by a later iteration.

In fact, only flow dependence represents a real transfer of values between iterations and it is for this reason sometimes referred to as a *true dependence*. The other two dependences are classified as "memory related," and can be satisfied through a technique called *privatization*. This involves allocating individual copies of each

```
int i, sum = 0;
for ( i = 0; i < 5; i++ )
        if ( P() )
                sum++;
```

Figure 1.1: Induction/Reduction Variables

variable which exhibits this dependence behavior to each loop iteration (or to each processor if a single processor may execute several iterations). Tu and Padua[TP93] describe a technique for automating this transformation.

A Dependence can be further classified as either *loop-independent* or *loop-carried*, depending on whether it exists independently of any loop inside of which it is nested. A loop-independent flow dependence does not inhibit any parallelization of the outer loops because it will still be satisfied. Loop-carried dependences may inhibit parallelization because the simultaneous execution of different iterations may leave them unsatisfied.

1.4.3 Special Case Dependences

There are certain cases of dependence which arise even within nominally parallelizeable loops. An *induction variable* is updated by a predictable amount on each loop iteration. In this case, a closed form expression for the value on each loop iteration can often be generated. A *reduction variable*, on the other hand, is updated by an unpredictable value on every loop iteration; these variables are generally used for statistical activity summaries. We can transform the way in which this statistic is accumulated to prevent this from inducing a serial semantics on the loop.

Figure 1.1 shows examples of these two dependences. In the figure, `i` is an induction variable because it depends only on the loop iteration number. In contrast, `sum` is a reduction variable because it depends as well on the value of an (unanalyzable) predicate. In general, for reductions we must implement some kind of non-serial combining strategy of the results computed at each iteration. The precise details will depend on the nature of the combining operator and the stability of intermediate results. For example, it may be unwise to arbitrarily reorder reductions to a floating point variable.

1.4.4 Dependence Analysis

Dependences which involve only scalar variables are trivial to detect simply by observing whether a loop contains one or more writes to any scalar. More complicated are dependences which involve array subscripts. The traditional way of detecting these dependences has been to compare each possibly overlapping array reference with every other, within the context of the loop bounds and their respective subscript expressions[GKT91].

This approach has been problematic for a number of reasons:

- The number of references to each array tends to increase with application size, and the quadratic asymptotic complexity of comparing every pair of references scales poorly.
- Non-affine subscripts cannot be easily compared.
- Procedure boundaries can make reference pairs difficult to relate, sometimes requiring elaborate value propagation techniques[Mas95].

To cope with these and other problems, array summarization techniques which can represent the aggregate activity for a set of accesses have been proposed. To find potential dependences, these summary sets can be much more quickly intersected with one another than the corresponding individual dependence tests between each pair of references can be performed. The following summarization techniques have been proposed:

Bounded Regular Sections[CK87] Represent only rectangular, triangular and diagonal array sections precisely, but support efficient intersections.

Data Access Descriptors[Bal90] Represent only convex polyhedra whose boundaries are either parallel to a coordinate axis or at a 45^o angle to a pair of axes.

While this is a crude approximation to convex polyhedra, it does support an efficient intersection.

Convex Polyhedra[AI91] Represent any affine constraints, but intersection operation (integer programming) is expensive.

Presburger Formulae[Pug94] Represent some non-affine constraints. Intersection (Omega Test) can usually be done efficiently.

Lists of Polyhedra[CI97] Represent unions of convex polyhedra without introducing approximations. Intersection operation is even less efficient than for individual polyhedra.

Guarded Array Regions[GLL97] Represent high-level constraints on accesses explicitly to ensure as little approximation as possible. Efficient intersection is promoted through aggressive simplification of the high level representation.

Access Region Descriptors[PHP98] Represent extremely precise array region information through specification of *strides* and *spans*. Efficient intersection is promoted through aggressive simplification of the high level representation; loop analysis is kept efficient through the avoidance of dataflow iteration.

Thus we have steadily progressed to extremely effective array summarization techniques based on ever more elaborate symbolic analysis. However, there are theoretical and practical limits to the utility of symbolic analysis. The general flow-dependence problem has been shown to be undecidable[Rep99], and in fact real-world programs often fail to yield to our best symbolic analysis efforts. This is especially true of applications written in modern computer languages such as C and C++ where pointer usage greatly complicates the analysis.

1.4.5 Insufficiency of Static Techniques

Static analysis, however useful, cannot completely address the analysis issues which arise in complex, real-world applications. The brief program in Figure 1.2 demonstrates why this is so. The main loop of this program may contain several different

```
main( int argc, char *argv )
{
        extern int *A;
        for ( int i = 0; i < 5; i += argc )
                A[i] = A[i+atoi(argv[1])];
}
```

Figure 1.2: Necessity of Runtime Analysis

types of dependences depending on the value of the runtime value of `argv[1]`:

- If it is between -5 and -1, there is a loop-carried flow dependence.
- If it is between 1 and 5, there is a loop-carried anti dependence.
- If it is 0, there is a loop-independent flow dependence.

Otherwise, there is no dependence. It is clear from this example that the sole use of static analysis may not be able to provide sufficient information to permit parallelization in all cases.

1.4.6 Runtime Parallelization

There are several different approaches to runtime parallelization. The most popular is to derive conditions from the original program under which loop parallelization can be proven legal. For example, in Figure 1.2 this would involve a runtime test on the value of `argv[1]`. A serial or parallel version of the loop is then selected based on the outcome of this runtime test.

However, it is not always easy to derive sufficient conditions on loop parallelization from the source program. Therefore, some have used the strategy of deriving "inspector loops" from the original program. These are just side-effect-free skeletons of the original loop in which only address calculations are performed. If the inspector loop is considerably faster to execute than the original, it can be executed first, and the pattern of memory accesses computed can then be used to allow or disallow parallelization of the original loop.

Unfortunately, this technique has its own drawbacks. For one thing, it may not be possible to extract the necessary inspector loop in cases where a cycle exists in the dependence graph for the loop[KM90]. Furthermore, even if no cycle exists, inspector extraction for loops spread over multiple procedures can be impractical.

1.4.7 Speculative Parallelization

In still another approach, Rauchwerger and Padua[RP94] suggest that we go ahead and parallelize loops with possible dependences and check for actual existence of these dependences at runtime. Their technique, which they call "speculative parallelization," calls for the allocation of a shadow array for every array which might contain a flow dependence. As elements in the original array are accessed, the shadow array records the access type. When the parallel step is over, the access patterns of all iterations can be compared to determine the existence of an actual flow dependence. If the parallel execution turns out to be illegal, a backup copy of all shared memory is used to re-execute the loop sequentially.

To prevent slowdown in the case where serial execution is necessitated, they further suggest that a serial version of the loop be run simultaneously whose results can be used in the case that the parallelization is found to be illegal; the remaining processors are to be used for the parallel version.

However, this technique is formulated in a Fortran setting, where expressions which access memory are relatively rigid. In C and C++, however, memory expressions may be arbitrarily complicated and include autoincrement and autodecrement side effects. The use of shadow arrays in this environment would cause significant mangling of the code, potentially requiring the breaking up of expressions and the introduction of temporaries. This reduces programmer recognition of and confidence in the generated code[Ram98].

Also, because the technique is intended to apply to fine grained parallel loops running on multiprocessors, it is realistic to perform the speculation on every program run—after all, the cost of speculation is low. However, we target coarse-grained loops for the NOW, where the high overhead of a parallel step is only justifiable when we are "reasonably certain" that there are in fact no unsatisfied dependences.

1.5 Introducing Practical Parallelization

This thesis presents a scalable, automated, network friendly technique, which provides a practical approach to building parallel applications from embarrassingly parallel serial programs. We avoid substantial overheads in the requisite DSM subsystem by harnessing the efficiency and scalability of the Network File System (NFS)[PJS^{+}94] as our DSM subsystem. We sidestep the more stringent consistency requirements which NFS does not fulfill by supporting only inherently parallel programs.

1.5.1 Incremental, Optimistic Approach

We first identify the application loops with significant potential parallelism. Out of these, we disqualify loops which we can show statically *not* to be parallel. We then transform the program into a parallel form in which the other loops are optimistically parallelized. Simple source code transformations segregate variables that need to be shared into a particular section of memory, from which they are easily communicated to all participating computers through the facilities of a small runtime system. Our transformation is robust because it preserves substantially all original program structure, introducing only additional initializations and pointer indirections.

We have constructed a complex (10,000 lines) prototype parallelizing compiler and a corresponding simple (100 lines) runtime system. The compiler performs the above analysis and inserts calls to the runtime system for scheduling and shared memory allocation and setup. We have used this system to fully automatically parallelize two real-world embarrassingly parallel applications. Furthermore, we demonstrate the scalability of our approach by showing that machine utilization improves with increasing problem size.

Because our approach is optimistic, we may generate incorrect parallel code if the semantics of the original program were not truly parallel. The current implementation requires awareness of this possible misuse on the part of the user. However, we describe extensions to our technique which, with some additional runtime overhead, will allow the system to detect at runtime whether a parallel model violation has in fact occurred. Although the extent of this overhead has not been ascertained through actual implementation, we argue that it should be minimal for our domain of coarse-grained loops.

Although the actual occurrence of a violation will depend both on the input data and on the configuration of the NOW on which the application is running, a particular parallelization may be "tested" over a number of varying inputs and machine configurations, the size and breadth of which can ensure that the possibility of future failures is minimized. Our transformation utilizes the underlying protection mechanisms of the operating system rather than relying on particular language features to detect runtime parallel model violations, ensuring language independence.

This incremental and optimistic approach is unique in that it respects the fact that different users may use the same application in substantially differing ways, and that the identical parallelization may not be optimal for all users. It views parallelism as a simple application customization, done for particular users upon demand and in the context of their particular needs. It supports fully automatic parallel program generation which can proceed independently in several directions, because all parallelizations are generated from the identical sequential source code, merely respecting different constraints.

1.5.2 Client/Server Architecture

A sequential application is divided up into a server application, which contains all the code outside of any parallel loop, and some number of client applications, which contain the respective loop bodies of the parallelized loops. Calls to the scheduling subsystem are inserted into the server application to replace each parallelized loop to execute the appropriate iteration space of the relevant client application. Each client application is constructed to accept a consecutive range of the iteration space as well as the iteration step as input parameters. The comparison operator of the loop termination condition need not be a parameter, as it is statically known from the structure of the original loop.

1.5.3 Shared Memory Segregation

Once the parallel loops have been chosen, a conservative estimate of all program variables which must be shared, including dynamic memory allocation sites, is generated. All declarations and uses of these variables are modified to include an additional level of indirection, and each declaration is initialized to point into the appropriate location within the shared memory segment. This gives us complete freedom in shared memory placement while at the same time respects the variable scoping of the original program.

At runtime, this shared memory segment is mapped both to the server and all clients through OS-level file-mapping primitives and NFS. At the server, the mapping is with `MAP_SHARED`, so that changes are immediately reflected in the global copy. At each client, the mapping is `MAP_PRIVATE` so that none of the changes to shared memory are reflected in the master copy until the client concludes and explicitly returns its changes to shared memory to the server.

1.5.4 A Small Example

Consider the program in Figure 1.3. The first order of business is locating the coarsest-grained program loops. In our example, there is only one loop, so we consider this for parallelization. We note that the loop body, line 8, references 3 variables. There is obviously no flow dependence in the loop because the only variable written to, `a`, has a different array element accessed on each iteration. Although `g` is referenced on every iteration, the reference is only a read, which by itself does not create a flow dependence. Also, note that the induction variable `i` is incremented linearly and is not modified in the body of the loop. Therefore, this loop is parallelizeable in our model. This program would be transformed to the version in Figure 1.4.

Figure 1.4 (a) shows the generated server program. We see on lines 1s-2s that both `g` and `a` have been transformed to shared variables. They both have one additional level of indirection and will both be initialized to point into the shared data segment,

```
1:      int g = 10;
2:      int a[5];
3:
4:      int main()
4:      {
5:              int i;
6:              for ( i = 1; i < 4; i++ )
7:              {
8:                      a[i] = g * i;
9:              }
10:             printf( "Series sum is %d.\n", a[1]+a[2]+a[3]+a[4] );
11:     }
```

Figure 1.3: Input Program.

although at different offsets. (The parameters to the call to `shcreate` are the offset and total size of the shared segment, respectively.) An additional complication in the case of `g` is that it already had an initialization. Therefore, a dummy variable `__1` is created to provide a slot where an initialization expression can be inserted into the code immediately following the allocation of the shared memory for `g`, exactly as would have taken place in the original sequential application.

On line 6s, the original loop has been replaced by a scheduling call which includes the loop bounds, test, step and client application as parameters. Because `i` is an induction variable, is is not treated as shared, but is specially handled within the client.

Now we turn to the `client1` code in Figure 1.4 (b). As in the server, the variables `g` and `a` are initialized with a call to allocate shared memory, but since this shared memory already exists at the server, it is not actually created, but the proper offset is merely returned. This is why unlike in the case of the server, no shared data segment size is given as an argument. Notice also that the original initialization of `g`, while reconstructed for the server, is left out of the client. This is because the value of the shared memory has already been initialized by the server.

The client program consists of a main function with initializations of the loop

```
int (*g) = (int *)shcreate( 0, 36 ), __1 = (*g) = 10;
int (*a)[5] = (int (*)[5])shcreate( 16, 36 );
int main()
{
        int i;
        i = remotely_spawn( 1, "<", 4, 1, "client1" );
}
```

(a) Server Code

```
int (*g) = (int *)shopen( 0 );
int (*a)[5] = (int (*)[5])shopen( 16 );
int main( int argc, char *argv[] )
{
        int i = atoi( argv[1] );
        int __limit = atoi( argv[2] );
        int __step = atoi( argv[3] );
        for ( ; i < __limit; i += __step )
        {
                (*a)[i] = (*g) * i;
        }
        write_pagediffs( argv[4] );
}
```

(b) Client Code

Figure 1.4: Output Client/Server Program.

bounds and step (lines 5c–7c), a loop header (line 8c), the transformed original loop body (line 10c), and a concluding memory update operation (line 12c). The actual arguments to the client are determined by the runtime system, depending on the overall number of jobs and the number of available machines. At the conclusion of the assigned iterations, the program determines which memory locations have changed, and sends this information back to the server, which can then update the master memory image for those iterations.

1.6 Achievements

Using a wide variety of techniques we have created a practical approach to the effective utilization of the NOW as a general parallel programming platform:

- We devise a technique for quickly zeroing in on likely coarse grain loops.
- We develop an automatable transformation methodology for the co-location of shared variables which preserves original declaration scope.
- We show that the DSM needs of our model are fully supported by the popular and scalable NFS.
- We show how to generate customized server and client applications for the parallelization. This should scale better than the popular monolithic approach of using the same program for the server and all clients, as used, e.g., in Treadmarks[KDCZ94].
- We construct an implementation of our transformation and show that it provides speedups of two real-world applications on the NOW, *fully automatically.*
- We devise a language-independent technique for the runtime verification of parallelization correctness which is efficient within our domain of coarse-grain parallel loops.

- We show that this potentially sequential verification step can be done efficiently through the appropriate division of the loop iteration space among available machines.
- We propose a strategy for successive parallel program refinement to any arbitrary confidence level in lieu of perfect data dependence information.
- We advocate parallelism as a simple program customization which is informed by the manner in which an application is actually used.

Although our transformed applications are targeted towards the low cost and flexibility of the NOW platform, the parallelism we expose could equally as well be exploited on more sophisticated hardware, for those who have it. Custom versions of the server and client operations could even be developed to take advantage of the specific features offered by particular platforms.

1.7 Outline of the Thesis

In Chapter 2 we explain the special significance of coarse-grained loops for parallelization on the NOW. We then show how to quickly locate coarse grain parallel loops through the construction of the Interprocedural Loop Level Graph, which makes the interprocedural looping structure explicit. By examining this graph, we can easily discover the outermost program loops, which are the most likely loops to provide a coarse parallel grain. These loops are then cheerfully parallelized unless there is obvious indication that this is illegal. Finally, Chapter 2 shows how the set of shared variables can be conservatively estimated.

Chapter 3 describes our automatable methodology for the transformation of shared variables. It shows how the runtime initialization feature of C++ makes the dynamic allocation and initialization of shared memory possible.

Chapter 4 describes how we customize the generated program for NOW execution. Unlike other parallelization systems which depend on various Distributed Shared Memory or Message Passing libraries, we use only popular and scalable network services of NFS. We discuss the relevant runtime issues and explain the benefits of our approach.

In Chapter 5, we report on performance improvements achieved in two actual case studies of the use of our techniques. The first of these applications is a junction detector, which is used to locate sharp features such as corners in images. The second is a ray tracer, used to generate high quality graphical images. Both of these applications are inherently parallel because the information computed at one pixel does not use the information computed at any other pixel. We benchmark the automatically generated parallel versions of these applications on a variety of input data sets and interpret the range of performance levels achieved.

The following Chapter 6 develops the final ingredient for a fully automatable parallelization. We show how speculative parallelization can be performed in a language-independent manner based on the MultiView[IS99] fine-grain variable access protection technique. Because we need to audit data accesses at the element level, we propose a more elaborate variable transformation which exposes every element to access protection. We further show that while this technique may induce secondary effects on the transformed program, it will terminate. Although this sophisticated protection scheme engenders additional overhead, the cost of this overhead recedes with increasing grain size. Moreover, we show how the complexity our final verification step is a function of the number of machines rather than the number of loop iterations, avoiding a potentially devastating sequential bottleneck.

Chapter 7 compares our work to that of others, gives direction for the future, and summarizes our contributions.

Chapter 2

Interprocedural Loop Analysis

The most fundamental issue in parallelizing a program is deciding what to parallelize. We begin with a discussion of the special properties of coarse-grained loops which makes their parallelization of critical importance for the NOW, and go on to examine why exploiting this parallelism has traditionally been found to be difficult.

2.1 Significance of Coarse-Grained Loops

In general, parallelization of a sequential program is not possible without the introduction of at least some overhead. In particular, the overhead in the setup of a parallel step implies that the sum of the execution times of all processors participating in a parallel execution will exceed the time spent in an equivalent sequential execution, ignoring performance gains from memory hierarchy effects. The advantage of using a specialty hardware multiprocessor is the minimization of this overhead through resort to hardware communication functions optimized for speed. In this context, although real execution time savings are possible even for individual executions of fine-grained loops, coarse-grained loops are still preferred because they ensure a high ratio of work to overhead.

On the NOW, however, this overhead is greatly pronounced, and completely overwhelms the benefit of parallelizing a fine-grained loop. Therefore, the identification and exploitation of coarse-grained loops is the only way of actually achieving real speedups.

2.2 Difficulty of Coarse-Grained Loop Identification

Unfortunately, analysis difficulty increases with loop size. There are two reasons for this: the analysis must cope with more information and therefore demands careful optimization, and the analysis is likely to cross procedure bounds, which brings a host of new issues into the analysis such as aliasing of parameters, recursive functions, and more.

Although there has been significant success in intraprocedural analysis[Kuc96], the field of interprocedural analysis is still in progress, with mixed results reported across a wide variety of academic prototype efforts.

Therefore, our approach recognizes that some guesswork is inevitable in any practical parallelization scheme, and we focus our efforts on what we can do well: identifying potentially parallel coarse-grained loops. Locating "correct" parallel loop candidates requires an analysis which can assign some measure of computational cost to each loop. We assume for simplicity that the entire program is available for analysis and no recursive cycles are present in the Program Call Graph (PCG); Section 3.1 will reveal the justification for the second of these assumptions.

2.3 Global Recursion Level Analysis

Constant Propagation[ASU86] increases program efficiency by replacing the runtime evaluation of constant variables and expressions with the values of the constants themselves. It is a useful and well-understood technique, but limited to the optimization of expressions whose value never changes. Some program variables, while not completely constant, tend to change very slowly, and updating the program text each time the value changes can be more efficient than repeatedly fetching the value from memory.

Glacial Variable Analysis[AW97] is a staging analysis for computing the *glacialness*, or infrequency of modification, of each variable. It begins with Global Recursion Level Analysis (GRLA), which is an interprocedural analysis using the PCG and the CFG (Control Flow Graph) of each procedure. The goal of GRLA is to assign staging levels which indicate relative frequency of execution to each block of code. Once these levels are known, Glacial Variable Propagation uses the Static Single Assignment graph to determine how frequently each variable changes.

GRLA uses the loop and procedure structure of the program to assign staging levels. Reducible CFGs give rise to natural loops[ASU86], which are by definition either disjoint or nested. Call graph analysis[CCHK87] reveals which procedures are called from each loop; for programs that use function pointers, call graph construction may proceed in tandem with alias analysis[BCCH97]. For simplicity, we assume that this analysis succeeds in generating an unambiguous PCG.

We define the *Loop Tree* of a procedure as a root node plus one node for each loop, with edges connecting each node to all loops which it directly contains. We also define the *Program Loop Graph* as the graph obtained by adding a connecting edge from each Loop Tree node to the root of the Loop Tree for each function called from that node.

We construct a monotone dataflow analysis framework as a tuple $D = < G, L, F >$ of our Program Loop Graph G, lattice L whose elements are drawn from the set of nonnegative integers with MAX meet, and monotone function space $F \subseteq f : L \rightarrow L$ which increments the lattice value. GRLA begins the staging level assignment by assigning zero at the root of the Loop Tree of the main procedure. The entire Program Loop Graph is then traversed with every loop assigned a staging level one greater than the loop in which it is nested, and with individual Loop Tree roots assigned the meet of each of the loops from which it is called. The loops which are assigned the highest lattice values are the most likely places in which the Value Specific Optimization will be helpful.

2.3.1 Inverse GRLA

We modify the above formulation of GRLA by starting the assignment of staging levels at the leaves of the Program Loop Graph, which are each assigned level 0. The stage level of each inner loop is one more than the meet of each of its constituent loops; in the case of a root Loop Tree node, it is precisely the meet of the staging levels of all inner loops. When the staging levels have been assigned, the coarsest grained loops will be the ones with the highest lattice values.

In general, this analysis will discover a number of "main" functions if there are several unused functions included in the program, which is usually the case. Although it is possible to distinguish the real one by matching to the `main` identifier, this would prevent application of the system to the parallelization of libraries. A truly general solution simply takes the highest-labeled function as the main function. This works as long as the loop depth of the unused code is lower than the loop depth of the main program—probably a safe assumption.

The structure of the chosen main function is then examined. In general, it will consist of a sequence of statements, function calls and loops. Individual statements and low-numbered function calls and loops (below a parallel-practical threshold) are ignored. Loops with high level are considered for parallelization, and functions with high level are recursively examined. In this manner the main function is (recursively) divided up into an alternating sequence of sequential code and parallelizeable loops.

2.4 Program Loop Level Graph

The Program Loop Level Graph is a graphical representation supporting visualization of the preceding dataflow analysis. It extends the Program Call Graph to make loop nesting explicit. By traversing this graph we can efficiently assign a loop-nesting level to each loop of each function, and we assume that the highest of these are the coarsest grain loops. Although this is merely a heuristic, experience has shown that this technique works well in practice.

Finding the most expensive program loops involves two major steps: assigning

```
SetNests( Loop loop, bool nest )
{
        int funcs = 0, nests = 0;
        for ( i in loop.calls )
                funcs = max( i.loop.nesting, funcs );
        for ( i in loop.inner )
        {
                SetNests( i, true );
                nests = max( i.nesting, nests );
        }
        loop.nesting = max( funcs, nests ) + nest;
}
```

Figure 2.1: Nesting Level Algorithm.

loop levels to each loop, and finding the maximum of these over the entire program. The following subsections explore the algorithms for each of these tasks in detail.

2.4.1 Assigning Nesting Levels

Figure 2.1 shows the interprocedural nesting level algorithm. It assumes that each loop is initialized with the following data elements:

- A list of inner loops, generated using perhaps the dominator-based techniques of [ASU86].
- A list of defined functions called from within the loop (but not subloops). (We assume that undefined library functions do not contribute significant loop nesting to the program. If this is not the case, a dummy function of the same name can be defined to mimic the expected loop nesting of the library call.)

Every function contains a nominal "outer loop" with nesting level of 0, which does not imply looping semantics, but is just a container for the function body. The algorithm begins by taking the maximum loop nesting level of all functions called within the loop. Then for each inner loop, it calls itself recursively to find the

```
FindCostly( Loop loop, int threshold, Loops costly )
{
        for ( i in loop.lcalls )
                if ( i not a recursive call )
                        FindCostly( i.loop, threshold, costly );
        for ( i in loop.inner )
                if ( i.nesting > threshold )
                        costly += i;
}
```

Figure 2.2: Costly Loop Identification Algorithm.

maximum loop nesting level over all inner loops. Finally it assigns the loop nesting level as the maximum of the function nesting level and the inner loop nesting level, with one added if this is a recursive call (signifying an actual inner loop as opposed to a "nominal" outer function loop). The SetNests function is applied to each function separately in reverse topological order based on the call graph, with leaves done first.

2.4.2 Locating the Costliest Loops

The costliest loops will naturally be called from the main function—the function with the highest nesting level for its nominal "outer loop." Finding the costliest loops, then, requires merely locating all top-level loops called from this main function, with perhaps some selection of the "better," more deeply nested ones. The algorithm is given in Figure 2.2.

The algorithm checks for recursive cycles in the call graph and does not follow them. This is fine because our model does not consider loops in recursive functions for parallelization, as mentioned above. For every other loop call, it calls itself recursively on the function's outer loop. At the bottom level of the recursion, all 0-level outer function loops have been explored, and the algorithm simply checks every inner loop of every function loop and returns the ones which exceed some nesting threshold; a good heuristic based on empirical evidence seems to be to consider all outer loops with nesting level within 2 units of the maximum level found at the outer loop of the main function.

It is easy to see that each of these algorithms visits each program loop and function call no more than once, and therefore are both linear in the size of the program.

2.5 Nominal Parallelizeability Test

Each coarse grained loop is then analyzed for parallelizeability. This involves minimal testing for this non-exhaustive list of disqualifying conditions:

- Input operations or other operating system calls
- Termination conditions which are inherently sequential (often true of `while` loops)
- Premature loop exit conditions
- Non-local transfers of control such as `longjmp` or C++ exceptions
- Obvious loop-carried flow-dependence, using perhaps the scheme of [GKT91]
- Obvious allocation of memory without corresponding deallocation

Notice that loop-carried output- and anti-dependences do not disqualify a loop, because these are memory-related and each iteration will use a private copy of the shared memory. Notice also that as special cases of reductions, file appends are permitted within a parallel loop.

Loop-carried flow dependence can not in the general case be checked statically, but in Chapter 6 we devise a technique for performing this check efficiently at runtime. Non-matching allocation/deallocation pairs are easily checked by keeping records of each of these operations at the client and making sure they match up at client termination.

If the program depends on library functions only available in object format, we must find an alternative method of communicating pertinent details of their operation to the parallelizer. This is done through the compilation of lists of standard library functions and noting whether or not they maintain state. If so, then their presence in a loop may invalidate any parallelization.

As an aid to the user, we can also indicate which loop was a desirable parallelism candidate and why it had to be disqualified. This is a very effective strategy because it quickly focuses attention to the most important loops in the program and their problems, instead of making the user decide without any guidance which loops should be parallelized. It should then be a fairly easy task for the user to remove or modify the offending code.

At the end, we arrive at the maximal set of parallelizeable coarse-grained loops present in the input program. We accomplish this in a simple two-phase algorithm: first bottom up, assigning labels to loops and functions, and then top-down, finding the largest-grain parallelizeable loops. This procedure is very efficient, and focuses attention quickly on the loops of interest.

2.6 An Example

Consider the standard matrix multiplication program in Figure 2.3. This program contains several loops, the most important of which is contained in `mmult()`. Our Interprocedural Loop Level Analysis applied to this program yields the Loop Level Graph displayed in Figure 2.4.

The nodes `srand48`, `drand48` and `printf` represent undefined library functions, and as such are not assigned a level or considered in the analysis. For all the other nodes, the number in parenthesis represents the loop level. The nodes `main`, `mmult`, `print` and `randomFill` represent the nominal "outer loops" of their respective functions. All other nodes represent actual loops, and the reader will observe that every additional loop nesting raises the level by one. Each of the functions `mmult`, `print` and `randomFill` are assigned the same level as the (single) loop they contain; `main` is assigned the maximum level of all loop levels which it encloses.

Now that we have shown how to identify the loops of special interest for parallelization, we turn to finding a conservative estimate of the set of shared variables which are associated with the parallelization of individual loops.

```
void randomFill( float M[SIZE][SIZE] )
{
        for ( int i = 0; i < SIZE; i++ )
                for ( int j = 0; j < SIZE; j++ )
                        M[i][j] = (drand48() - 0.5) * 1000.;
}
void print( float M[SIZE][SIZE] )
{
        for ( int i = 0; i < SIZE; i++ )
        {
                for ( int j = 0; j < SIZE; j++ )
                        printf( "%f\t", M[i][j] );
                printf( "\n" );
        }
}
void mmult( float A[SIZE][SIZE], float B[SIZE][SIZE], float C[SIZE][SIZE] )
{
        for ( int i = 0; i < SIZE; i++ )
                for ( int j = 0; j < SIZE; j++ )
                {
                        C[i][j] = 0.;
                        for ( int k = 0; k < SIZE; k++ )
                                C[i][j] += (A[i][k] * B[k][j]);
                }
}
void main( int argc, char *argv[] )
{
        float A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];
        srand48( 0 );
        randomFill( A );
        randomFill( B );
        mmult( A, B, C );
        printf( "Matrix A:\n" );
        print( A );
        printf( "Matrix B:\n" );
        print( B );
        printf( "Matrix AxB:\n" );
        print( C );
}
```

Figure 2.3: Matrix Multiplication Program.

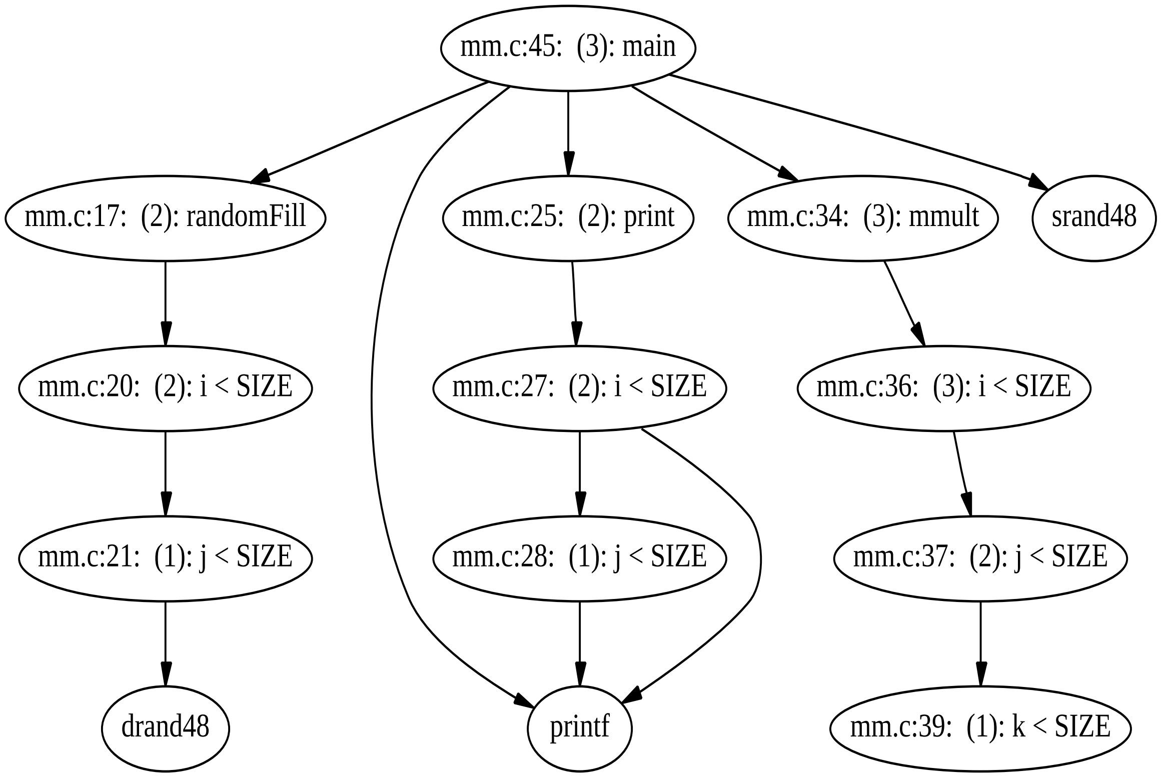


Figure 2.4: Interprocedural Loop Level Graph.

```
LoopRefs( Loop loop, Set refs )
{
        refs += loop.lrefs;
        for ( i in loop.inner )
                LoopRefs( i, refs );
        for ( i in loop.lcalls )
                LoopRefs( i.loop, refs );
}
```

Figure 2.5: Finding Variables Referenced in a Loop.

2.7 Interprocedural Shared Variable Identification

Once the parallelizeable loops have been nominally identified, it may or may not be simple to identify the variables which must be shared. In particular, variables which are found to occur textually both within a parallel loop body (and its callees) and elsewhere are obviously shared. More subtle are the cases of addressed variables and heap memory, which could be accessed in ways which the source code does not clearly reveal. A simple, conservative approach is to label all instances of these memory genres as shared memory, unless proven otherwise.

The algorithm in Figure 2.5 shows how to find the nominal set of variables referenced within a parallel loop candidate. It assumes the loop-level analog of classical REF sets for functions—the set of variables referenced within particular loops. It also assumes precomputation of the set of functions called from within each loop.

This algorithm is called on each parallel loop candidate. For each of these loops, it adds locally referenced variables, plus the results of calling itself recursively on each of its inner loops and the nominal "outer loop" of each of the functions which it calls. Where recursion is present, the algorithm does not follow the backedge of the cycle, although this detail is omitted from the figure for simplicity.

To the set of all variables which are explicitly mentioned inside parallel loops we can add all variables which are addressed anywhere in the program, unless it can be proven that the address never crosses a parallel loop boundary.

Of these variables, we select only the ones which are live outside of the loop body.

This includes variables from global scope, variables which are static (even though they may be local to a function called only inside of the loop), and stack variables which are declared in the same function as the loop, but lexically scoped externally to it. All of these shared data elements are collected and placed in a single shared memory area.

The running time for this algorithm is at most linear in the size of the program, because each program loop is visited at most once.

With a conservative estimate of the set of potentially shared variables in hand, we turn in Chapter 3 to a demonstration of how we can use this information to transform these variables in a uniformly automatable manner, providing the semantic means to implement shared memory efficiently on the NOW.

Chapter 3

Shared Memory Segregation

Because it is difficult to keep a DSM system simple and efficient if it must manage an unlimited number of tiny shared memory segments, it is customary to aggregate all shared data into a single place. Unless a programmer has designed his application from the ground up with parallelism in mind, this shared memory segment will be composed of shared variables which are arbitrarily scattered across the application.

There are three important memory categories:

1. **Application Local** This memory is not accessed within any parallel step.
2. **Shared** This memory is accessed both within and outside of one or more parallel steps.
3. **Thread Local** This memory is accessed only within a single thread of a parallel step.

The second category, shared, represents the memory state which must be preserved to support idempotence of individual jobs.

These shared variables are of several general classifications:

- Global to all source files.
- Local to a particular file.

- Local to a particular function (to use C terminology, these are generally `static`, but may be `auto` if they appear within the text of a parallelizeable loop, but are declared outside of the loop).

To comply with DSM semantics, then, it is necessary to collect all of these variables into a single contiguous shared data segment. There are generally two ways in which this segment is assembled.

The first scheme, typical of most DSM systems, is to have the program request allocations from a shared memory segment at runtime. These can naturally come from whatever address the DSM finds convenient. Unfortunately, this is problematic in that it calls for the programmer to use shared memory in ways which are unnatural. In particular, the memory may not be able to be initialized in file scope together with ordinary program variables because the program must first make a call to initialize the DSM subsystem. Furthermore, gratuitous indirection syntax may be necessary at every variable usage. However, this method does give the programmer complete control over where and when the shared allocations are made.

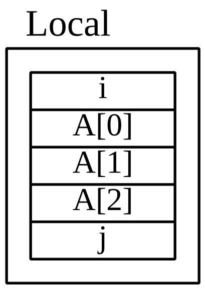
The second scheme is to force the programmer to manually relocate all shared variables to a single defining block in the source program. In Chapter 1 we pointed out some drawbacks of this approach.

This chapter describes a technique that combines the advantages of both of these schemes. We devise a program transformation which adds an extra level of indirection to each shared variable declaration and usage, and in this manner is able to maintain the scoping semantics of the original program while providing freedom for the DSM to place the shared variables all together in a single memory segment.

3.1 Shared Memory Layout Scheme

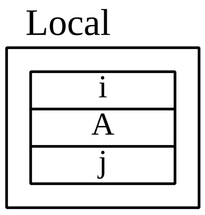
A completely general scheme for shared memory layout cannot entail any variable reshuffling at the source code level. A simple way of avoiding this reshuffling is to add one level of indirection to each shared variable declaration. Consider, for example, the declarations in Figure 3.1(a). Suppose the program analysis finds that the array

```
int i;
int A[3]; // shared var
int j;
```



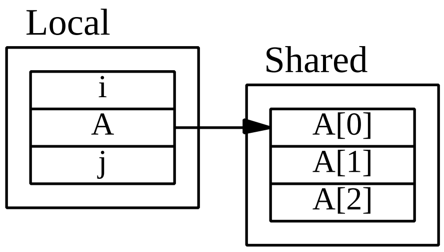
(a) Original Program

```
int i;
int (*A)[3];
int j;
```



(b) Indirection Transformation

```
int i;
int (*A)[3] = (int(*)[3])shcreate();
int j;
```



(c) Initialization to Shared Memory

Figure 3.1: Shared Variable Transformation

`A` needs to be shared. In order to keep the variable in its original scope while at the same time maintain the freedom to position the memory somewhere else, we simply add one level of indirection to the declaration, as shown in Figure 3.1(b).

However, there are several issues which must be addressed for this to be a general solution. For one thing, how is this variable-cum-pointer initialized? And what if the variable already had its own initialization, how is the ultimate target in shared memory initialized?

We handle the first of these problems by turning to an especially opportune feature of the C++ language, runtime initialization. Classical C places restrictions on the content of variable initialization expressions for all global (or `static`) data. Specifically, initialization expressions must be compile-time constants. This is presumably because in classical C these initializations were actually performed on the executable file image of initialized memory at compile time.

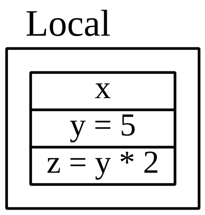
C++, on the other hand, provides a great deal more flexibility by deferring these initializations until runtime, and thereby being able to support arbitrary expressions as initializers. Capitalizing on this, we immediately initialize all shared variables at declaration to point to the area of shared memory which is returned by our static shared memory allocation function, `shcreate()`, as shown in Figure 3.1(c). (The inner workings of `shcreate()` are explored below.)

At this point, we know how shared memory is allocated and initialized. Now we turn to variable initializations. Consider the initialized variable `y` in Figure 3.2(a). Notice that the declaration of `z` uses the initialized value of `y`, so if we are to completely follow the original program semantics, we must provide a way of initializing this variable at declaration, just as in the untransformed code.

We do this by introducing dummy variables, such as `__1` in Figure 3.2(b). Because the initialization syntax supports arbitrarily complex expressions, we can chain the shared memory location `(*y)` in a series of assignments which terminates with the dummy variable. Although this does introduce extra baggage into the source program, in practice most compilers will probably notice the uselessness of the garbage declaration and ignore it, other than to perform the visible side effects we intend.

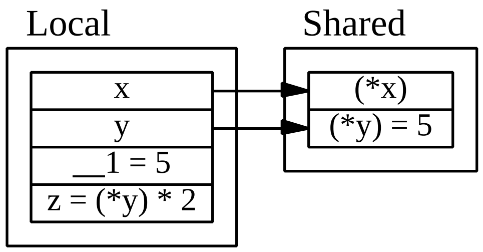
This initialization trick works just as well for aggregates, as shown in Figure 3.3.

```
int x; // shared var
int y = 5; // shared var
int z = y * 2;
```



(a) Initialized Shared Scalar

```
int (*x) = (int*)shcreate();
int (*y) = (int*)shcreate(),
    __1 = (*y) = 5;
int z = (*y) * 2;
```



(b) Initialization after Sharing Transformation

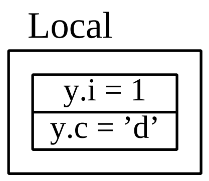
Figure 3.2: Scalar Initializations

Even C++ classes with constructors can be supported under this scheme, with judicious use of the placement `new` operator, which is designed exactly for this purpose: to direct the allocation to a particular *shared arena*[Str97].

Every shared variable can be initialized consistently in this manner except for procedure parameters. Since there is no parameter initialization, shared parameters must be augmented with an additional indirect declaration within the function body, which can then be initialized normally and assigned the actual parameter value. The indirect version is then substituted for all uses of the parameter.

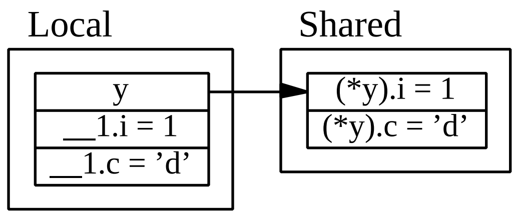
Stack variables may be shared, but the only stack variables for which this this is possible are the ones in functions which contain parallel loops. By replacing these shared stack variables with indirect references to a particular address in shared memory, we are implicitly changing the semantics to static. This will not affect program correctness unless the containing function is called recursively. For this reason, loops which are contained in a function which participates in a call graph cycle are not considered for parallelization, and call graph cycles were ignored in the loop level

```
struct agg { int i; char c; };
struct agg y = { 1, 'd' }; // shared aggregate
```



(a) Initialized Shared Aggregate

```
struct agg { int i; char c; };
struct agg (*y) = (agg*)shcreate(),
  __1 = { (*y).i = 1, (*y).c = 'd' };
```



(b) Initialization after Sharing Transformation

Figure 3.3: Aggregate Initializations

analysis of Chapter 2.

As shown in the initialization examples, an indirection operator is inserted into every use of a shared variable. This scheme even works when shared variables are addressed as the address operator and dereference operator simply cancel each other out, and the pointer value itself is returned.

We now have the tools we need to develop an automated transformation methodology for shared variables, unburdening the programmer from one of the headaches which must otherwise be endured before a parallel program can be generated. The next section shows an allocation scheme for the shared memory which is network-friendly.

3.2 Shared Memory Management

We capitalize on widely available network services by modeling the capabilities of our system on available system services, instead of the other way round. NFS is a widely available protocol for file sharing which, due to its popularity, is likely to be among the best tuned and most scalable network services available (although in Chapter 5 we will point out a notable exception to this premise). Likewise, file-mapping kernel primitives are likely to be fast. We combine these two technologies to obtain a runtime system which is a good fit for our application domain and our chosen platform, and which capitalizes on the shared variable transformation we have developed above.

As described more fully in Chapter 4, we are developing a client/server model of parallelism where the server is the transformed application program, and each parallelizeable loop has been encapsulated within a client "mini-app." As this mini-app is parameterized by loop iteration count, as many independent copies of it as is desirable may be spawned at runtime on all available machines. To provide a common shared memory image between the server and all clients within a given parallel step, we encapsulate all shared data into an NFS-mounted file and map it at both the server and all clients. Through this simple technique, we obtain a highly efficient and extremely portable implementation of Distributed Shared Memory.

3.3 Static Shared Memory

We have seen that given a set of parallelizeable loops, we can compute a conservative upper bound of which variables may actually be used within the loop. Knowing this, we can precompute a shared memory layout, and use it to determine the precise offset in the shared memory segment where each shared variable will ultimately reside.

In all of our examples of shared memory initialization above, we have omitted for simplicity the parameters to the `shcreate()` call. We supply an offset parameter as argument to each of these allocations. The first time this functions is called, it allocates the entire segment; subsequent calls simply dole out the various pieces of it to initialize the appropriate variables.

Since the first call must know the size of the entire shared memory segment, we must include this as a parameter to the first call. Unfortunately, we cannot in general know which is the first call, because there may be global initializations in separate source files for which no platform-independent initialization order exists. Therefore, we simply tack on the data segment size to every call to `shcreate()`.

On the client side, the corresponding `shopen()` function performs a similar task, returning the appropriate offset into the same shared memory file (through the magic of NFS and memory-mapped files) for each shared variable.

If memory allocations are present which cannot be proven not to allocate memory which must be shared, these will be redirected to call `shmalloc()`, which returns available space from a preallocated heap area in the shared data segment. Only programs which actually use this feature will pay the (slight) overhead which the reservation of large amounts of shared memory space implies. Furthermore, because of the simplicity and efficiency of the runtime system, "over-engineering" and using shared memory where not absolutely necessary should produce only a negligible performance hit.

Calls in the server application that free memory are replaced with calls to `shfree()`, which can perform the appropriate action within the shared memory segment if the argument is determined to refer to the shared data segment, else to pass the call right on to `free()`.

On the client side, there is no special processing required for dynamic memory management, other than to ensure that there is no attempt to retain pointers to memory allocated inside a client on return to the server. This entails disqualifying the loop from further parallelization attempts if some mismatch is found between the `malloc` and `free` call records at client termination.

3.4 Memory Mapping

On the server side, the shared memory file is mapped using `MAP_SHARED`, because all clients initialize their shared memory segments by mapping in the same memory file and need to be able to see all updates to the shared memory which precede the parallelized loop invocation.

In order to perform the identical mapping on the client side as on the server side, the client must know the name of the shared memory file and the address to which it was mapped. Moreover, these pieces of information cannot be passed as arguments to the client application, because data definitions may be global—indeed may occur in an entirely different source file than the client's `main()`.

Therefore, we use the two environment variables `$MEM_FILE` and `$MEM_ADDRESS` to communicate these values. On the first call to `shopen()` the filename contained in `$MEM_FILE` is mapped to the address in `$MEM_ADDRESS`. Since this address was available for mapping on the server side, it is likely to be available to the client as well, and our experience has borne this out.

At the client, the shared memory is mapped using `MAP_PRIVATE`, because, as we will explain in Chapter 4, client execution must remain idempotent and can therefore not be allowed to write directly into the shared memory space. Furthermore, the memory is write-protected, and a handler unprotects the particular pages which are written to, while storing the page addresses in a dirty page list.

3.5 Static Sizing of Shared Memory

Our approach to shared memory services is simple, automatic and potentially widely applicable. While there are many obvious benefits to this approach, here we mention one which is more subtle.

As shared memory is requested, the size of the shared data segment increases. When the sizes of all requests are unknown in advance, there is a need for dynamic resizing of the entire segment, which involves unmapping and remapping to the larger size. One source of difficulty is that the operating system may choose to remap the memory at a different address than before, which would invalidate all current references to it. This problem has driven many DSM systems to preallocate a large, fixed size data segment in spite of the attendant loss of flexibility. This is why users of CVM[Kel95], for example, find 40 megabytes of swap space consumed by each application, regardless of size.

Because we can determine the size of static shared data at compile time, however, we can cap the size of the shared data segment when no calls to `shmalloc()` are necessary, and preallocate a segment of only that size. This cannot be done in systems such as CVM in which all allocations to shared memory, whether static or dynamic, are treated identically.

The next chapter will present a runtime system that uses these techniques on the

NOW.

Chapter 4

Runtime Support

As we mentioned briefly in Chapter 1, we generate one server program and one or more client programs. The server contains all application code except for parallelized loops, for each of which one client application is constructed.

In place of each parallel loop, the server has a call to a scheduling module with the loop bounds, step and client application name as arguments. The scheduler eagerly schedules some number of instances of the client on available machines, updates the memory with the computation results, and returns control to the main application.

4.1 NFS as DSM Server

Interestingly enough, we have found that native NFS facilities in conjunction with kernel-level memory-mapped files can supply all our shared data handling needs. This departs from some earlier studies of the suitability of NFS as a DSM server. In fact, Minnich has even developed his own souped up version of NFS, Mether-NFS[Min93a], which fills the gaps in NFS to make it a powerful communication and synchronization subsystem. Minnich points out that NFS was not designed to support shared memory, and presents several compelling arguments for the necessity of a more powerful drop-in replacement:

- NFS client code does not differentiate between read and write faults.

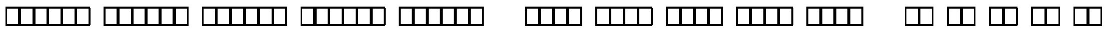


Figure 4.1: Splitting the Iteration Space

- NFS server code does not ensure that only one process is writing a page at a time.
- No coherence is available.
- More than one page size is needed.
- A mechanism for delivering a writable copy of a page to holders of read-only copies is needed.
- A mechanism for direct application-to-application synchronization is needed.

While these points cannot be denied, they come from an application-centric mindset: This is what my program needs. How can we provide it?

DSM subsystems are constructed for the purpose of bringing shared memory multiprocessor semantics into the network setting, whatever the cost. The NFS protocol was, naturally enough, designed to do precisely what can be done well on a network. To get the best performance out of network parallelization, we have taken this lesson to heart and said instead: This is the style of parallel programming the NOW can support efficiently. How can we use it?

4.2 Iteration Space Partitioning

We partition the iteration space to ensure a number of different grain sizes are present, and that the larger grains cover earlier iterations than the later grains. This scheme is graphically depicted in Figure 4.1. A range of grain sizes ensures a proper balance between the conflicting goals of minimizing overhead and achieving a proper load balancing. Minimizing overhead calls for large grains, while a smaller grain size supports better load balancing. Henceforth, a computational grain will also be referred to as a *job*.

We start every machine with a good sized chunk of work which is estimated to keep it busy for some fraction of the ultimate duration of the parallel step. Some machines will finish faster, and can be given the remaining smaller chunks in the remaining time of the parallel step. Others will remain occupied, and perhaps be overtaken once all of the smaller jobs have been finished.

We specify that the larger grains be comprised of earlier iterations in the iteration space so that we can subsequently overlap the incorporation of the results of these iterations with the computation of later iterations. This is particularly important in circumstances in which the updates to global state must be performed in a particular order, such as when the standard output of each iteration must be collected into an overall output stream for the loop.

To support file appends within a parallelizeable loop, we store these appends in temporary files until one of the client copies successfully completes, and then only after all earlier grains of the loop have also been completed and the corresponding file appends performed can we perform the file append of the most recently completed grain. Task copies are prevented from interfering with the files of other copies by including the copy number in all temporary file names.

4.3 Page Diffs

The `write_pagediffs` function is called at the client right before exit. It loops through the list of dirty pages populated by the write-protect handler and compares the values byte by byte with the values in the corresponding shared memory file pages. The new values, along with corresponding addresses, are written to a temporary diff file, which is specified to each client on the command line. This file, of course, is also named by job and copy number, so that different copies of the same job do not interfere with each other.

4.4 Job Flushing

We have seen that larger chunks of work must be scheduled before smaller ones. We further insist that larger chunks be associated with earlier iterations. This lends efficiency to the system by allowing us to commit successful jobs concurrent with scheduling, rather than delaying all updates until the successful completion of the last loop job.

Upon successful completion of each job, then, we loop through the entire job list. Jobs may be classified as “done” and “flushed.” All jobs which have already been flushed are skipped over. Jobs which are not done break the loop. All other jobs encountered in job list are marked flushed and the corresponding commitments are performed. This entails concatenating temporary output files to the appropriate output stream, updating a second copy of the shared memory with the changes recorded in the client-side diff, and deleting the corresponding temporary files.

Besides the efficiency advantages, this scheme has the practical benefit of preserving file system space by not requiring simultaneous storage of all intermediate results. This is important for programs whose output cannot fit into available file system space, such as those writing to a pipe.

We have constructed a prototypical implementation of the system as we have described it here, and tested it on two example programs. The next chapter, Chapter 5, describes these case studies in detail, reports on performance achieved, and interprets the results.

Chapter 5

Case Studies

The previous chapters of this thesis have described an integrated system for the transformation, scheduling and execution of embarrassingly parallel programs on networks of workstations. In this chapter we present two applications which have been transformed using these techniques, and report on observed performance. Only optimistic parallelization has been implemented, so for this study we have chosen applications whose main, coarse-grained loops are known to be parallel.

The first application is Kona[PGH97], an image-processing application written by Laxmi Parida which detects junctions. The second is Gk, a ray-tracing application written by George Kyriazis. Each of these presented particular challenges, as described below.

5.1 Methodology

Tests were run on a 17 machine network of 200 Mhz Pentium Pros running Linux 2.0.3, interconnected with a 100 Mbps Ethernet through a non-switched hub. The network was isolated to eliminate outside effects.

One of the machines was reserved for execution of the server application, while from 1 to 16 machines were used for the "mini-apps" of each parallel loop. All our performance timings represent the average value of five identical runs. The server machine is not considered in the graphs of our performance results.

For each application, we show the effects of problem size on utilization by using a range of different problem sizes. Because our applications read or write pixels, we accomplish this by varying the number of pixels read or written.

5.2 Case 1: Kona

Kona processes a given rectangular portion of an image, dividing the local neighborhood of each pixel into a particular number of pie slices, or spokes. It computes the average intensity of each spoke, and finds a best-fit set of line segments to match to these intensity levels. These lines are then matched to precomputed junction templates to locate image features.

The Kona application consists of 5 source files:

`main.c`, 7 lines This file contains the `main` function, but all it does is call a workhorse driver function.

`read_raster.c`, 141 lines This file contains image raster reading and allocation functions.

`save_raster.c`, 57 lines This file contains image raster saving and freeing functions.

`permute.c`, 120 lines This file contains functions which are used within the main program loop.

`Y.c`, 1136 lines This file contains the coarsest-grain program loop.

This program is particularly difficult to analyze manually because it uses macros heavily, and does not uphold the elements of data-hiding and encapsulation which are the hallmark of good programming style. Indeed, the present author invested over a month in the original hand-parallelization of this application for the Calypso[Bar99] system.

5.2.1 Preliminary Analysis

The main loop called the following library functions: `fabs`, `free`, `malloc`, `printf`, `sprintf`. Of these, `fabs` and `sprintf` present no special difficulties, and `printf` is allowed under general file append support. `free` and `malloc` were nominally permitted, but the calls were checked to ensure they are complementary. The program call graph contained one recursive cycle called from the main loop.

5.2.2 Program Transformation

The first step in transforming the program involved determining precisely which source files needed be changed to generate the server application. The file which contained the admissible loop nest, `Y.c`, clearly did. Figure 5.1 lists all static variables which were determined to be shared with respect to the parallelizeable loop, along with their places of definition and use. Each of the files which use or define any of these also needed to be rewritten. The figure also lists the locations in which memory was allocated which could have been used in the parallel step. Since the allocation function for these calls needed to be changed to `shmalloc()`, these files needed to be rewritten as well.

The first 13 variables in Figure 5.1 are scalars which are Read-Only within the parallel loop; they are initialized once (or twice if set through the command line) at the start of the program. The image itself, stored in dynamic memory pointed to by `image`, is also Read-Only within the parallel loop. The other variables in the figure are written as well as read within the loop.

Each generated file is written in C++ and therefore has the `.cc` extension. Three new source files were created for the server:

- `Y.rewrite.cc`, 2545 lines
- `permute.rewrite.cc`, 1378 lines
- `read_raster.rewrite.cc`, 783 lines

Note that these files are generally larger than the files from which they were generated because they include all header file declarations. However, they generally

```
shared variables:
        Y.c:9: static double RAD_threshold
        accessed in:   Y.c
        Y.c:10: static int odd_dim
        accessed in:   Y.c
        Y.c:11: static double rad_hole_fraction
        accessed in:   Y.c
        Y.c:12: static int no_of_spokes
        accessed in:   Y.c
        Y.c:13: static double homogen
        accessed in:   Y.c
        Y.c:18: static int XMIN
        accessed in:   Y.c
        Y.c:19: static int XMAX
        accessed in:   Y.c
        Y.c:20: static int YMIN
        accessed in:   Y.c
        Y.c:21: static int YMAX
        accessed in:   Y.c
        Y.c:22: static int no_of_linefits
        accessed in:   Y.c
        Y.c:23: static double alpha1
        accessed in:   Y.c
        Y.c:24: static double alpha2
        accessed in:   Y.c
        Y.c:25: static double cornerity
        accessed in:   Y.c
        Y.c:1042: static double theta[361]
        accessed in:   Y.c
        Y.c:1043: static int gl_no[100]
        accessed in:   Y.c
        Y.c:1044: static int gl_ptr[100]
        accessed in:   Y.c
        Y.c:1045: static radial_line_type gl_rad[100 * 100]
        accessed in:   Y.c
        Y.c:1050: unsigned char **image
        accessed in:   Y.c
        permute.c:64: static double lasterr1
        accessed in:   permute.c
        permute.c:64: static double lasterr2
        accessed in:   permute.c
shared heap:
        read_raster.c:17: 1
        read_raster.c:18: 1
        permute.c:111: 1
```

Figure 5.1: Kona Shared Variables.

compile to smaller sized object files than the original files because these declarations do not take up space in the program image, and only relevant code is included in the generated files.

Special attention in the rewriting was required to change certain identifiers, particularly `this` and `new`, to prevent conflicts with reserved C++ keywords. These files were then compiled and linked with the other application files and the server library to form the parallel server application.

For the client, `Y.c`, which contains the client loop body and functions it calls, and `permute.c` which contains other functions called from the loop body needed to be generated. Note that these functions would have needed to be custom-written for the client in any case because they contain definitions of static shared data.

The following client files were generated:

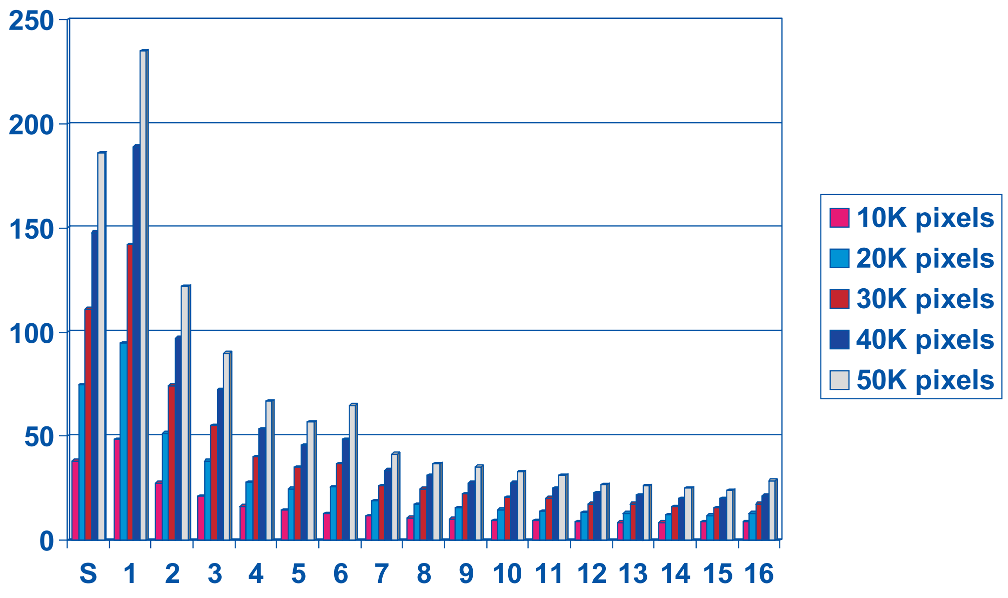
- `client1.cc`, 1350 lines
- `client1.permute.cc`, 918 lines

Each of these included all data type and function prototype declarations, but only the relevant data and function definitions, with the shared variables declared using, as usual, the specially initialized indirect syntax. These files were compiled and linked with the client library.

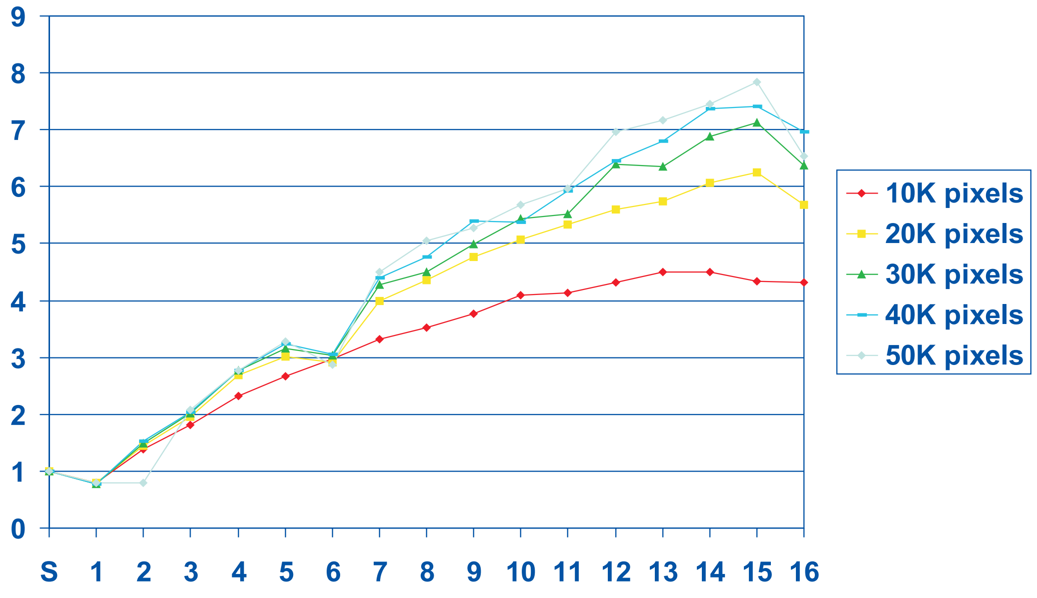
5.2.3 Performance Results

In Figure 5.2(a) we show the execution time for each of the runs. The column labeled "S" represents sequential application timing. The figure reveals a steady speed improvement except at 6 and 16 machines. Figure 5.2(b) graphs this speedup, where each point is calculated as the ratio of the sequential running time to the running time for each number of machines. We leave out the server from our graphs because the load on the server will be minimal for small numbers of machines.

Figure 5.2 shows that as the problem size increases from 10K to 50K pixels, the network utilization becomes more consistent and predictable. At 10K pixels, this speedup increases almost linearly with the number of client machines until it levels



(a) Execution Time



(b) Sequential versus Parallel Time

Figure 5.2: Kona Performance Results

off at 13 machines, where additional machines do not improve the running time; at 50K pixels this limitation is not observed.

5.3 Case 2: Gk

Gk is a ray-tracing program. It reads a scene definition file and writes an output image at a user-supplied resolution. To make the analysis and transformation easier, file writes were replaced by writes to standard output (although the underlying techniques apply straightforwardly to either). Furthermore, several reduction variables in the form of operation counts were discovered. Although we shall describe ways of handling them, they were removed in our experiments for simplicity.

5.3.1 Preliminary Analysis

Gk consists of 8 files:

`vector.c`, 82 lines This file contains various vector operations.

`readfile.c`, 154 lines This file contains file operations.

`main.c`, 52 lines This file contains the `main` program function, which calls a workhorse function `raytrace` after performing some initializations.

`trace.c`, 237 lines This file contains the `raytrace` function. This is the function which contains the coarsest-grain program loop.

`intersect.c`, 182 lines This file contains ray intersection facilities.

`initialize.c`, 21 lines This file contains code to initialize various counters, which we disabled to break dependences.

`shade.c`, 175 lines This file contains shading functions.

`bg.c`, 44 lines This file contains some color functions.

The program contains one recursive cycle, composed of the functions `trace_a_ray`, `trace` and `shade`. One deeply nested loop was discovered in the file `trace.c` in function `raytrace`. It calls the library functions `pow`, `printf`, `random` and `sqrt`. None of these present a problem except for `random` pseudo-random number generator. The random numbers generated within the parallelized iterations would obviously not be the same as would occur in a sequential execution, depending, as random numbers do, on an initial seed and all subsequent random numbers.

Whether or not this is problematic is really up to the application developer; we proceeded assuming that this was acceptable. (If this had been determined to be insufficient the techniques set forth in [Bre98] could be used to ensure better randomness.)

5.3.2 Program Transformation

The largest shared data structure is the list of scene objects, pointed to by `obj`. This memory is both read and written within the parallel step. Instead of being written to a section of memory, generated pixel color values are simply written sequentially to standard output. All of the variables in Figures 5.3-5.4 are read-only within the parallel step with the exception of `Time`, `col`, `color`, `ray`, `r2`, `r`, `g`, `b` and `i`.

The shared variable reference and use statistics listed in Figures 5.3-5.4 indicated that the following source files needed to be created for the server:

- `trace.rewrite.cc`, 1210 lines
- `readfile.rewrite.cc`, 1169 lines
- `main.rewrite.cc`, 428 lines
- `intersect.rewrite.cc`, 936 lines
- `shade.rewrite.cc`, 926 lines
- `bg.rewrite.cc`, 148 lines

The following client files were generated:

```
shared variables:
        main.c:30: struct light_t light
        accessed in:   readfile.c, shade.c
        main.c:33: int noo
        accessed in:   readfile.c, intersect.c
        main.c:36: int tries
        accessed in:   trace.c, readfile.c
        main.c:39: struct vector hor
        accessed in:   trace.c
        main.c:39: struct vector ver
        accessed in:   trace.c
        main.c:39: struct vector eye_dir
        accessed in:   trace.c, readfile.c
        main.c:40: double fov
        accessed in:   trace.c, readfile.c
        main.c:43: double time1
        accessed in:   trace.c, readfile.c
        main.c:43: double time2
        accessed in:   trace.c, readfile.c
        main.c:44: double Time
        accessed in:   trace.c, intersect.c
        main.c:46: int bgflag
        accessed in:   readfile.c, bg.c
        main.c:49: struct obj_t *obj
        accessed in:   readfile.c, intersect.c
        main.c:52: int xres
        accessed in:   trace.c, main.c
        main.c:52: int yres
        accessed in:   trace.c, main.c
        trace.c:165: int x
        accessed in:   trace.c
        trace.c:166: struct color_t col
        accessed in:   trace.c
        trace.c:166: struct color_t color
        accessed in:   trace.c
        trace.c:167: struct ray_t ray
        accessed in:   trace.c
        trace.c:167: struct ray_t r2
        accessed in:   trace.c
        trace.c:168: int r
        accessed in:   trace.c
```

Figure 5.3: Gk Shared Variables.

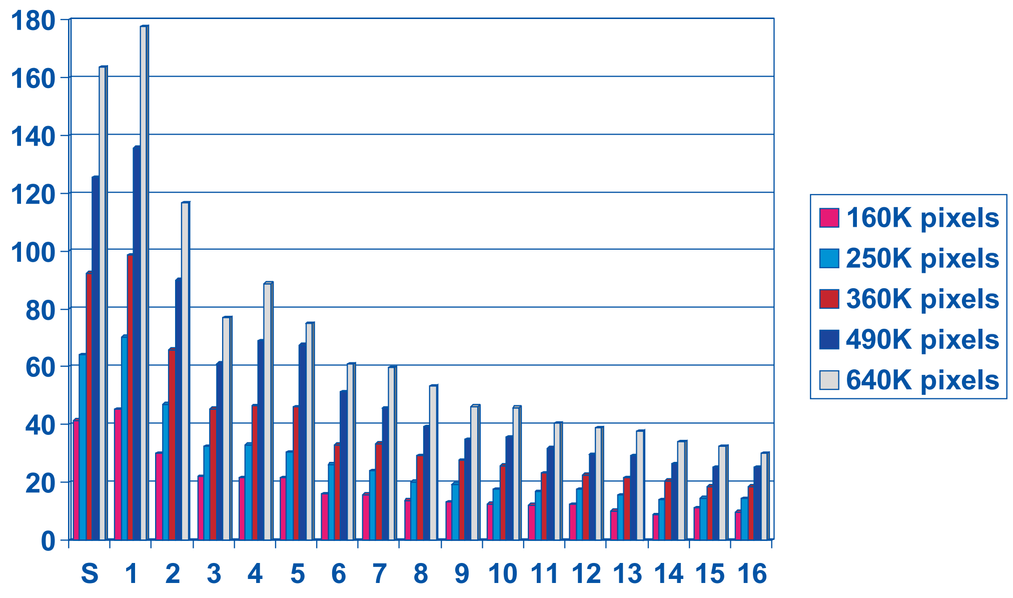
```
trace.c:168: int g
accessed in:   trace.c
trace.c:168: int b
accessed in:   trace.c
trace.c:171: int i
accessed in:   trace.c
trace.c:172: double p_w
accessed in:   trace.c
```

Figure 5.4: Gk Shared Variables (cont).

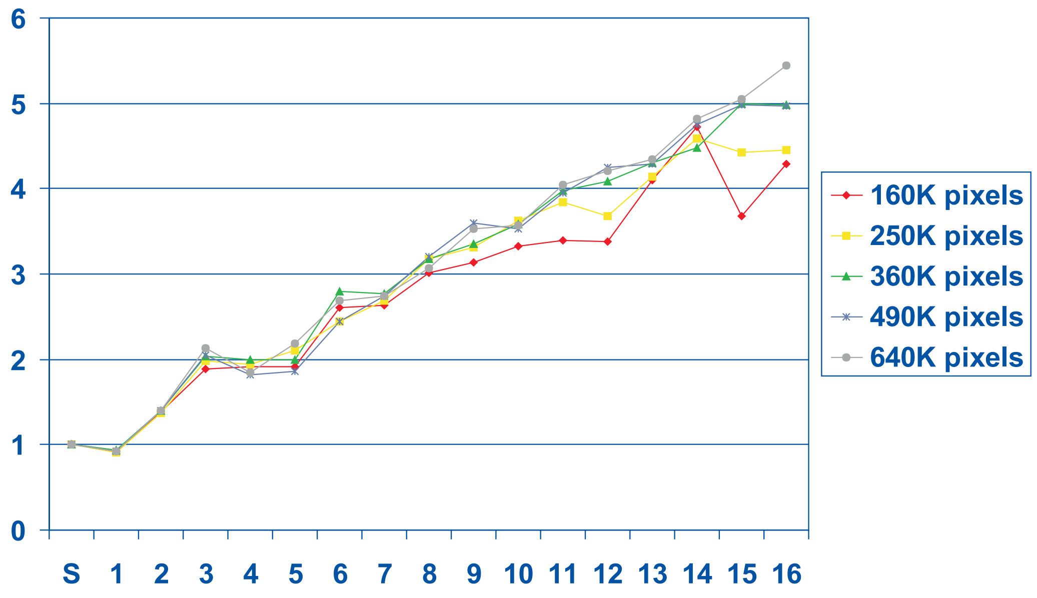
- `client1.cc`, 890 lines
- `client1.intersect.cc`, 619 lines
- `client1.bg.cc`, 141 lines
- `client1.main.cc`, 355 lines
- `client1.shade.cc`, 609 lines
- `client1.vector.cc`, 525 lines

Each of these included all data type and function prototype declarations, but only relevant functions. Notice that some files, such as `main.c` needed to be regenerated for the client not because they contained code relevant to the parallel step, but because of the relevant data declarations they included.

The initially generated program refused to compile because of multiple definitions of several of the shared variables. It turned out that one of the header files which was included in several of the program units contained several uninitialized data definitions (as opposed to simple declarations). This is a C anachronism which was permitted as long as no more than one initialization was present; it is not permitted in C++. To remedy this, we simply removed these definitions to one of the program source files, replacing them by `extern` declarations.



(a) Execution Time



(b) Sequential versus Parallel Time

Figure 5.5: Gk Performance Results

5.3.3 Performance Results

In Figure 5.5(a) we show the execution times for each of the runs. The column labeled “S” represents sequential execution time. The figure reveals a steady speed improvement except at 4 and 5 machines. Figure 5.5(b) graphs this speedup, where each point is calculated as the ratio of the sequential running time to the running time for each number of machines.

Figure 5.5 shows that as the problem size increases from 160K to 640K pixels, the network utilization becomes more consistent and predictable. At 160K pixels, this speedup increases almost linearly with the number of client machines until it levels off at 14 machines, where additional machines do not improve the running time; at 640K pixels this limitation is not observed.

5.4 Interpretation of Results

Notice that the overhead in Gk is consistently less than that for Kona. This is attributable to the fact that Gk performs relatively little I/O, which is associated with significant cost, during the parallel step.

The speedup discontinuities observed within both applications (6 and 16 machines for Kona; 4 and 5 machines for Gk) indicate a scheduling/granularity bug. It should be observed that this aberration is not only independent of the problem size, but of the structure of the input as well. This is revealed by the consistency of the dips in the graph of Figure 5.2(b), for which the different input sizes are associated with different input sets as well. We may surmise that this performance discontinuity is associated with the manner in which the problem structure interacts with our choice of granularities. In particular, decreasing the number of large grains would easily fix this, but perhaps at the price of slightly more overhead everywhere else.

This situation leads us to conclude that an optimal scheduling policy should include a granularity selection which is truly dynamic and can respond to the needs of a particular parallel loop on the fly, as has been done in the Calypso[Bar99] system.

5.5 A Note on Performance

While our performance has not been stellar, it is competitive with the performance of other vastly more complex DSM subsystems[JSS97]. Moreover, our chosen platform was a set of Linux machines, whose NFS service is known to be suboptimal. There are two reasons for this: the daemon runs entirely in user space, and does not yet implement proper caching. As our system stresses NFS quite heavily, these inefficiencies have obviously been expressed by our system in turn. In particular, our experiments have revealed significant delays between the startup times of concurrently spawned processes. It would be interesting to see how much performance improvement can be offered by a more competitive implementation.

Another optimization which may be significant is the use of Minnich's Vector eXecute (VX)[Min93b] remote process starter tool. He reports that starting identical processes on multiple machines can be done orders of magnitude faster as compared with the identical operations under more popular systems such as PVM[AB95]. As most of our overhead is in process startup, this could provide a substantial performance boost.

5.6 Conclusions

This chapter has reported on two case studies of the step by step transformation of two real-world application programs into client/server form. These applications are typical of many others and demonstrate the power of our technique. We conclude that this technique has important applications to some of the most pressing parallelization problems facing our community today.

Although we present powerful techniques to support a variety of application characteristics, we have demonstrated that this power is paid for only where it is used, and simple parallelizations can still achieve a large percentage of sequential performance.

The next chapter, Chapter 6, describes a planned extension of our system which can efficiently determine if the optimistic parallelization we have devised has violated any of the assumptions on which the parallelization is based.

Chapter 6

Speculation Extension

In Chapters 2, 3 and 4 we developed a methodology for automatic program transformation based on educated guesses of the most profitable loops to parallelize. Because we cannot guarantee that this transformation is legal, every time we run the program on a new input set we must check that the execution does not use memory in ways which violate the parallel model. This chapter describes a current approach to performing this check dynamically and then presents our solution.

6.1 The Polaris Approach

Rauchwerger and Padua[RP94] solve the speculative parallelization problem by allocating special shadow arrays for each array in question. Every access to an array element is reflected in the shadow array, and the order and manner of accesses to the shadow determine whether the parallelization was legal.

Although appropriate to fine-grain Fortran loops, there are a number of difficulties with directly using this approach on the coarse-grained loops of C and C++ programs. One problem is that in the case of dynamic memory, it may be difficult to create an appropriate shadow array. Furthermore, because of the rich complexity of C operators, it may be necessary to pepper the generated code with compiler temporaries when dealing with complicated expressions. Besides the reduction in programmer recognition of and confidence in the generated code which we pointed out in Chapter 1 this entails, it also makes the transformation process itself less robust because of the added complexity. We propose an alternative scheme which operates not at the language level, but at the operating system level.

6.2 Testing Execution Legitimacy

As we saw in Chapter 1, the basic violation which is exhibited through the parallelization of an unparallelizeable loop is the non-satisfaction of a flow dependence, where one iteration reads a value which has been written by an earlier iteration.

This can be checked with memory profiling, keeping statistics for every iteration of which locations were read only, read first, and written before being read. In fact, we do not need to perform this test for every pair of iterations if we use a granularity that includes several consecutive iterations in each grain, because within a particular grain, dependences will automatically be satisfied. Therefore, we can collect the information we need to perform this test on a per-grain basis.

Adapting Hoeflinger's[Hoe98] Region Analysis methodology to our setting, we associate three bits with each memory element: Read-Only, Write-First and Read/Write. For each read access, if no bits are set, we mark the target Read-Only. For each write access, if no bits are set, we mark the target Write-First. Otherwise if the Read-Only bit is set, we mark it Read/Write. This provides all the information we need to detect inadmissible flow dependences.

For example, if we find that one of the memory locations within a particular grain was read before being written, the legitimacy of the execution will depend on the access patterns to this location in all earlier grains. If no earlier grain wrote to the memory, then we know that the proper value for this memory location was indeed the value with which the memory for that grain was initialized—the value at the start of the parallel step. On the other hand, if some earlier grain wrote the memory, then the value which the present grain was using will have turned out to be an obsolete value, and the computation will be invalid.

But suppose a particular memory location was written before being read within a particular grain. In this case, the value at that location cannot depend on the value assigned within any other grain. Also, memory locations not accessed at all obviously cannot be the sources of dependences with other grains of the computation.

Notice that variable privatization happens completely implicitly according to our arrangement of the shared memory. Because the file mapping is private, no interference between the writes of two different grains is possible. Also, because we update the master copy of shared memory according to the order of writes within each grain, writes to the same location will correctly favor the later write.

6.3 Avoiding Sequential Validation Overhead

As Rauchwerger[Rau95] points out, it is very important that the validation test for the parallel execution be performed in parallel, or else we are simply replacing one sequential bottleneck with another.

We saw in Chapter 1 that the aggregation of iterations into a single grain of execution is important in the coarse grain parallelism appropriate to the NOW environment. Because we divide the iteration space initially according to the number of available machines, and the check we need to make is between *grains* and not between *iterations*, the number of tests we need to perform is proportional to the number of machines participating in the computation, far fewer than the number of iterations. Furthermore, the runtime system can overlap earlier checks with later computations if earlier sections of the iteration space generally finish before later ones.

An interesting aspect of this scheme is that even if a flow dependence exists within a parallel loop, whether or not it causes a parallel model violation will depend on the precise division of the iteration space among the available machines. Therefore, a validated program run does not necessarily imply the absence of dependences, only that this *particular execution* did not reveal any.

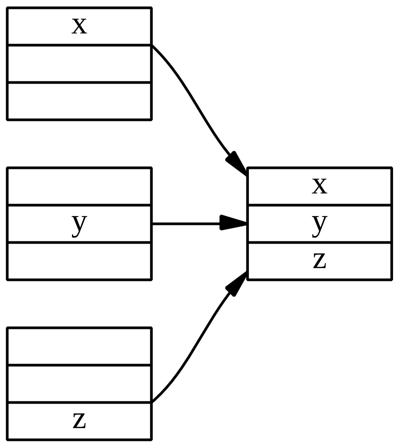


Figure 6.1: Example of MultiView.

6.4 Efficient Memory Profiling

In order to provide an efficient, language-independent, platform-independent methodology for the required element-by-element memory profiling, we turn to recent developments in the Distributed Shared Memory arena.

Itzkovitz and Schuster[IS99] pioneered a methodology called MultiView for fine-grained memory access control of DSM using the standard OS page fault mechanism. MultiView can support access faults at the variable level, even for variables which share pages with other variables.

The way in which this is done is through the construction of separate and independent mappings which map to identical memory areas. For example, Figure 6.1 shows how three variables which reside in the physical page on the right side of the figure are mapped through the three different virtual pages on the left side of the figure. Variables which point to the variable `x` are initialized to point to the version of `x` which appears in the first page, and similarly for `y` and `z`.

We can adapt this technique to provide access statistics for each memory element at every client processor. Instead of simply handing out pointers to the original mapping space of the shared memory, we create a separate mapping for each variable, and return the appropriate offset into it. We begin the grain's execution with no read/write permissions set on any of these mappings, and as page faults are generated, we turn on the appropriate access and set the appropriate bit. Locations which are only read or written before being read will generate only a single page fault, providing, respectively, read access and read/write access. Those that are read before being written will generate two faults: the first turns on read access and the second turns on write access.

While taking 1 or 2 page fault performance hits for each memory location may seem costly, this cost will be amortized over the number of memory accesses to the location. Therefore, as parallel grain size goes up it will have a waning impact and could be an ideal profiling scheme.

However, our solution is not yet complete, because a variable partition such as that supported by MultiView is not the same as an element-level partition, in particular, when the variables in question are aggregates (structures or arrays).

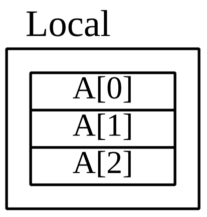
Aggregate values are addressed by a pointer and an offset. If an application can access an arbitrary aggregate member with the same pointer value simply by changing the offset, then access statistics can only be generated for the variable as a whole, or at best, at as fine a granularity as the number of pages it spans.

6.5 Protecting Aggregates at the Element Level

We saw in Chapter 4 that all shared memory accesses are transformed to indirect accesses. The reason for this is that we need the shared memory to be contiguous, and we do not have the freedom to rearrange data declarations within source files. So for example, consider the program fragments and corresponding memory layouts in Figure 6.2(a)–(c). Figure 6.2(a) shows the original; Figure 6.2(b) shows the results of the customary sharing transformation.

In order to arrange the memory such that MultiView can be applied at the element level, we have to alter the manner in which we set up the indirection. The idea is to maintain the structure offset access semantics, but to transform each element into a pointer-to-element. Using the memory layout of Figure 6.2(c), we can directly apply the MultiView technique to the targets `*(*A)[0]`, `*(*A)[1]` and `*(*A)[2]`, and efficiently determine the legality of our parallelization. Although not shown, the

```
int A[3];
for ( int i = 0; i < 3; i++ )
    A[i] = i;
```



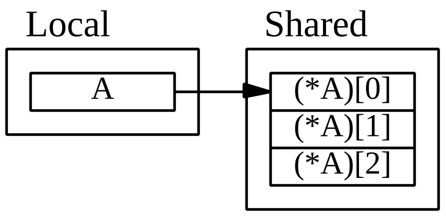
(a) Original Program

Server:

```
int (*A)[3] =
    (int (*)[3])shcreate();
```

Client:

```
(*A)[i] = i;
```



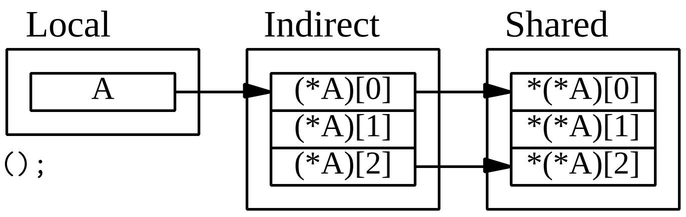
(b) Transformed Program

Server:

```
int *(*A)[3] =
    (int *(*)[3])shcreate();
```

Client:

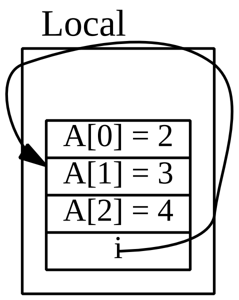
```
*(*A)[i] = i;
```



(c) Access Control at Element Level

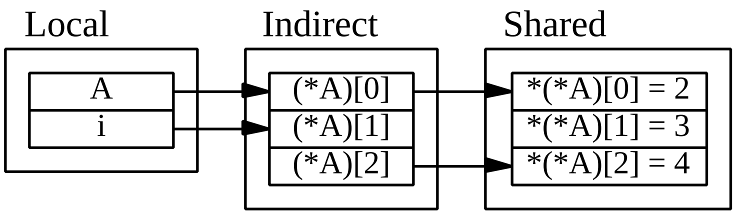
Figure 6.2: Progression of Transformations

```
int A[3]; // shared aggregate
int *i;
A[0] = 2;
i = &A[1];
i[0] = 3;
i[1] = 4;
```



(a) Original Program

```
int *(*A)[3] = ...
int **i;
*(*A)[0] = 2;
i = &A[1];
*i[0] = 3;
*i[1] = 4;
```



(b) Transformed Program

Figure 6.3: Aggregate Access Types.

calls to `shcreate()` will also need to specify the datatype so that the correct unit of actual data will be allocated for each pointer element to point to.

Using this technique, however, requires considerably more sophistication at the transformation level. For the simple sharing transformation described in Chapter 3, we were able to perform a uniform transformation, replacing every variable with a single dereference. For this new transformation, the context of the variable is critical in deciding how the transformation is to proceed.

There are two basic contexts in which an aggregate variable may appear: a value context, where one of its elements is being referenced, and an address context, where the address of some offset inside the aggregate is being taken. For example, consider the code in Figure 6.3(a). The first statement assigns to an element of `A`. The second statement assigns the address of `A[i]` to the variable `i`, through which the other elements of `A` are assigned. The sequence of statements results in the memory layout on the right side of the figure.

In the transformed code of Figure 6.3(b), observe that while the first reference to the array `A` had to be transformed, as it represents an element access, the other reference to `A` needed to be preserved because it is used as an address of a particular subelement.

Notice also that performing this sophisticated transformation for the array `A`, does not merely affect the places where `A` is mentioned. In fact, the transformation must ripple through the code to change every variable which is used to hold an offset of `A`. These ripple sites include assignments and function call sites. For the former, the declaration and all uses of the receiver of the address would have to be changed; for the latter, the function declaration (and all of its prototypes) as well as all uses of the corresponding parameter would have to be changed.

This can get complicated when the variable or function is used in other contexts as well such as being assigned the address of a different aggregate which is not shared. As demonstrated by the transformed code in Figure 6.3(b), in this case we can simply "upgrade" this second aggregate to shared status, and apply the entire transformation once again for this new variable. The same would apply to functions which are called both on shared aggregates and non-shared aggregates.

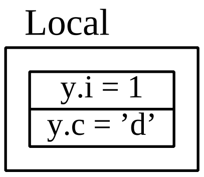
Viewing the transformation as a dataflow analysis on the two-element lattice of sharing-transformed vs. non-sharing-transformed, we can guarantee that this process terminates because the lattice values are monotonically increasing and bounded.

6.6 Structure Aggregates

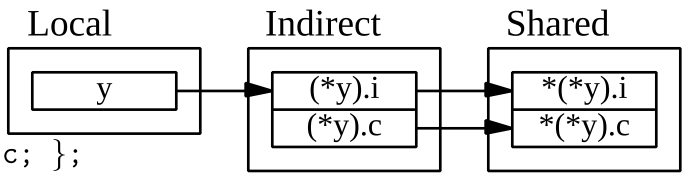
So far we have seen that the speculative transformation for array aggregates involves the inclusion of one additional indirection at the element level, over and above the indirection at the variable level.

The analogous transformation for structure aggregates is somewhat more complex,

```
struct agg { int i; char c; };
struct agg y; // shared aggregate
```



(a) Original Program



```
struct agg { int i; char c; };
struct agg_p { int *i; char *c; };
struct agg_p (*y) = (agg_p*)shcreate( ... );
```

(a) Transformed Program

Figure 6.4: Transforming Structure Aggregates.

because adding indirections at the element level fundamentally changes the data type. Therefore, the original structure declaration would need to be modified to include an additional indirection for each member.

In Figure 6.4(a) we recall our structure example from Chapter 3. Transforming this code to implement speculative sharing for `y` involves actually generating a new datatype `agg_p` from `agg` where each member is given an indirection, as shown in Figure 6.4(b).

The reader will note that this scheme relies on the consistent declaration and use of datatypes, so that when casting is used in unsafe ways, the scheme will fail and loop parallelization will have to be abandoned. Also, the size of each datatype pointer target will need to be communicated to the runtime system so that the appropriate space for each element can be allocated.

Of course, not every aggregate will need such elaborate protection. Some will

```
unsigned char **make_2d_charr( int rows, int cols )
{
    register i;
    unsigned char *arr_ptr;
    unsigned char **arr;

    arr = (unsigned char **)malloc( rows * sizeof(unsigned char *) );
    arr_ptr = (unsigned char*)malloc(rows*cols*sizeof(unsigned char));
    for ( i = 0; i < rows; i++ )
            arr[i] = &arr_ptr[i * cols];
    return arr;
}
```

Figure 6.5: Dynamic Shared Memory Example

clearly cause flow dependences and immediately disqualify the loop from parallelization. Others can be proven not to present any problem. Our method provides a flexible means of dealing with all of the "in-between" cases which would generally inhibit parallelizing compilers from attempting any parallelizing transformation. Moreover, in analogy with ordinary compilers, our system provides application performance commensurate with program analysis effort—a more elaborate analysis achieves better overall application performance by reducing parallel model verification effort at runtime, and reducing the likelihood of parallelization of unparallelizeable loops. Other systems may require a very high initial analysis effort for any parallelization benefit.

6.7 Dynamic Memory

While the transformation for data element-level protection is straightforward for statically declared data, it can be more subtle for dynamically allocated data. It will depend, for example, on the consistent use of allocated memory, and the ability of the translation system to derive a datatype from the manner in which the memory is used.

Consider the example in Figure 6.5, which is taken from one of our case studies, Kona (see Chapter 5). It shows a fairly common method for allocating a multidimensional array in C. First the entire array space is allocated, then a separate array is allocated which holds the address of each row. If these allocations cannot be proven not to be referenced within a parallel step, we must make the allocations from shared memory. As in the static case, the allocation call will need to specify the datatype of the aggregate and initialize all pointers at the element level to point to elements of this size.

In our example, the implied datatype of the memory is clear from the manner in which it is used, so that we know that each of `arr_ptr` and `arr` must be declared and used with an extra indirection at the element level. In the general case, pointer analyses of various complexities, such as [And93], [Ste96], [SH97], [BCCH97], [AFF97], could be employed for memory disambiguation.

6.8 Nominal Sharing Upgrades

When we must update a variable to shared status because of the manner in which it interacts with an actual shared variable, we should not actually include this variable in the shared memory. Besides the additional inefficiency this would engender, it also would be a source of inflexibility: recall, for example, that stack variables of functions which participate in recursive cycles cannot have shared semantics.

As a clever alternative, we can instead use only local declarations in the transformation of these nominally shared variables, as shown in Figure 6.6. In Figure 6.6(a), because aggregate `A` is shared and its address is stored into `i`, `i` must be redeclared to use double indirect syntax, and this ripples to the aggregate `B`, which now must also be redefined because its address is also stored into `i`. In Figure 6.6(b), we demonstrate how we have declared a shadow aggregate `B_sh` for the original `B` and initialized each pointer element of the `B` to point to the corresponding element of `B_sh`. Thus even if there is a domino effect and we are forced to upgrade a great many aggregates, we lose no flexibility and incur negligible overhead because we do not have to involve the runtime system with these nominally shared variables at all.

In the next and final chapter of this thesis, we sum up related work, suggest other

```
int A[5]; // shared aggregate
int B[5];
int *i;
i = &A[2];
i[2] = 5;
i = &B[2];
i[2] = 5;
```

(a) Original Code

```
int *(*A)[5] = (int *(*)[5])shcreate( ... );
int B_sh[5], *B[5] = { &B_sh[0],&B_sh[1],&B_sh[2],&B_sh[3],&B_sh[4] };
int **i;
i = &(*A)[2];
(*i)[2] = 5;
i = &B[2];
(*i)[2] = 5;
```

(b) Transformed Code

Figure 6.6: Transforming Nominally Shared Aggregates.

helpful extensions, and give some direction for the future.

Chapter 7

Conclusions

This thesis has described a framework for automatic incremental parallelization of "embarrassingly parallel" applications using the popular Network of Workstations as the target platform. Despite the theoretical ease with which these programs can be parallelized and their characteristic suitability to execution on Networks of Workstations, no effective automated solution to this problem has yet been offered.

While a good number of parallelizing compiler prototypes have been built, these were meant to cater a broader range of parallelization problems on more sophisticated architectures The unique challenges of efficiently running more banal parallel programs written in banal languages on banal parallel platforms should merit special attention.

7.1 Related Work

The strands of related work are numerous, and span a broad range of fields.

7.1.1 Two-Phase Idempotent Eager Scheduling

This technique for extracting the highest efficiency from a diverse array of sometimes faulty machines[KPS90] has recently found expression in the Calypso[Bar99] system. Although Calypso has proven the general usefulness of the technique, it is mainly a runtime system, leaving the identification of parallelism and shared memory up to the user. The more recent Chime[SD] system brings an interesting twist to this scheme, in that it requires the user to identify only the parallelism; shared data identification is avoided by declaring *all* global data shared. This still leaves some complications for shared stack variables, for which a cactus-stack scheme was devised.

7.1.2 Finding Coarse Grain Parallelism

Hall et al.[HAM$^+$95] report on the effectiveness of their SUIF compiler system at locating coarse grain parallel loops. Since they do not describe how this is done, it is difficult to comment on how their approach differs from ours.

7.1.3 Execution Cost Prediction

Zhou[Zho94] describes a program execution cost estimator. In his model, every instruction is associated with a cost, and these costs are summed up over the expected execution paths for the overall application. Although we also perform a sort of cost estimation, we do not collect information at this level of detail, but assume (correctly, in our experience) that the interprocedural loop nesting depth will be the most important estimator of execution cost. This would not have been sufficient for Zhou, whose goal was to be able to decide which of two applications would perform better on a particular problem—if the loop nesting of each is identical, our technique provides no insight into this problem.

7.1.4 Using NFS as a Scalable DSM Subsystem

Minnich[Min93a] decided not to use standard NFS as a DSM subsystem, opting instead to create his own souped-up Mether-NFS drop-in replacement. By proposing a model of network parallel programming which is well served by NFS, we have shown that the NOW may not have been a target whose properties are best suited to his application domain.

7.1.5 Program Instrumentation

This facility was provided by the PTOPP[EM93] toolset. The point was to zero in on the most time-intensive loops by comparing the profile data of all program loops. This approach is obviously not intended to discover top-level loops in particular, but it could benefit from our Interprocedural Loop Level Analysis to discover which loops might actually be useful to profile.

7.1.6 Partial Evaluation

Goff[Gof97] reports on the benefits of compiling a program together with its input, among other dependence-breaking techniques. She concludes: "of these ... techniques, partial evaluation is the one which should be employed with the most discretion." After all, this partial evaluation would need to be redone for every different set of input data. Goff considers the benefit of program input data only as a boost to the constant propagation subsystem, and the specific constant values read will inevitably change with different inputs.

We, on the other hand, have embraced this technique, although in quite a different form. We have devised a scheme for optimistic parallelization which uses memory profiling information generated from the application at runtime, a form of partial evaluation which can be performed efficiently during every program execution.

7.1.7 Speculative Parallelization

Rauchwerger and Padua[RP94] pioneered speculative loop parallelization for the Polaris parallelizing compiler. In their scheme, loop code is instrumented with shadow arrays which accumulate memory access information at the array-element level. Our scheme not only brings language independence and platform independence to this technique, but adapts it to the specific requirements of coarse-grained parallel loops.

7.1.8 Dynamic Sharing Granularity

Itzkovitz and Schuster[IS99] present a technique for achieving fine-grain page protection under operating systems which nominally provide only a coarser grain of protection. By mapping every variable to a different virtual memory address, they eliminate the classical false sharing problem for page-based Distributed Shared Memory Systems. Our work capitalizes on this concept to apply fine-grain protection at the machine-datatype level so that memory access profiles can be efficiently computed on a per-element basis at runtime.

7.1.9 Automatic Parallelization for the NOW

Using the PIPS parallelizing compiler framework, Coelho[Coe94] experimented with the applicability of automatically compiled HPF programs to the NOW platform. What he found was that the performance obtainable on multiprocessors was not portable to workstation networks, as the network quickly emerged as the bottleneck. We have seen that within our application domain significant speedups are possible provided careful attention is given to the special challenges of parallel computing on the NOW.

7.1.10 Restructuring for Performance Portability

Jiang et al.[JSS97] report on the various restructuring techniques which are helpful in obtaining parallel performance which is portable across a range of hardware and software multiprocessing platforms. They find that the restructuring for the lowest common denominator (NOW) is generally also somewhat beneficial for the higher-end systems, but not significantly. Because they address a range of applications, including irregular ones which exhibit data dependences between loop iterations, their work is more widely applicable than ours. However, for our limited problem domain, automatic techniques provide benefits with respect to essentially manual techniques.

7.1.11 Distributed Parameterized Simulation

The idea of running an identical application with differing parameters across a NOW was exploited in the Nimrod system[ASGH95], which can be used where the goal is to run an identical scientific simulation on a number of different input sets. However, because this approach may involve significant application rewriting to produce the parameterized form, it can prove costly. Although we use this idea, we automatically generate the required parameterization within our client "mini-apps".

7.2 Future Work

Here we describe several additional features which could improve our system. First we discuss reductions, which we saw in Chapter 5 were manually removed from our case studies; we now describe a scheme to support them.

7.2.1 Reductions

To support reduction variables, we can inform the runtime of the reductive nature of particular variables, giving the appropriate reductive operator. For example, Figure 7.1 shows a how a loop with a reduction might be transformed. (We show only the server code in the figure because the client need not be aware of any details of a reduction.) We have added a special "reduction operation" parameter to the `shcreate()` call, which causes the runtime not to disqualify the loop for finding a parallel model violation at this location, but to simply perform the appropriate reductive operation between the values found in the corresponding locations at each conflicting client.

To do this, however, it is also necessary to include the appropriate datatype as an operand to the `shcreate()` call, so that the appropriate arithmetic operator can be applied between the two reductive values. This is indicated in the figure with the final argument to `shcreate()`, `"i"`.

```
int main()
{
    int i;
    int sum = 0;
    for ( i = 1; i < 4; i++ )
    {
            sum += i;
    }
    printf( "Series sum is %d.\n", sum );
}
```

(a) Original Code

```
int main()
{
    int i;
    int (*sum) = (int *)shcreate( 0, 4, '+', "i" ),
            __1 = (*sum) = 0;
    i = remotely_spawn( 1, "<", 4, 1, "client1" );
}
```

(b) Server Code

Figure 7.1: Reduction Transformation.

7.2.2 Heterogeneous Networks

Throughout this thesis we have implicitly assumed a network of workstations with homogeneous architectures. We depend on this feature when we use the kernel memory mapping facilities to map the shared address space created on the server into the address space of each client. Although this simplistic scheme cannot be used across different architectures, an extension to the technique might be for the server to transform the contents of the address space on the fly into a format which can be used at the client side. This translation would only have to be done once per foreign architecture per parallel step; thus if there are a number of machines of the same type, the cost of this translation could be amortized over all of them.

One source of difficulty in this context is the translation of pointer values across architectures and operating systems. This requires detailed information about the locations in memory of all pointer values. This will certainly not be possible to obtain in general, but a sophisticated analysis may be able to generate this information for some applications. Having this information will also ameliorate a problem we encountered in Section 3.5, where we noted that we sometimes have to allocate a huge section of memory at program initialization in order not to have to relocate the shared memory segment. If the precise locations of all pointers in memory are known, every value can simply be updated on relocation.

7.2.3 Finer Grain Parallelism

Another direction for future research is the examination of loops at lower nesting levels for parallelization in the event that parallelism at the higher level fails. This research direction is particularly exciting because it holds out the possibility of extending our techniques even to the realm of nominally non-embarrassingly-parallel applications. An irregular molecular dynamics application, for example, may include data dependences between loop iterations, but each iteration may be expensive enough to be profitably parallelized using our techniques, if not on the NOW platform, then at least on some higher-end platform.

7.2.4 Reporting a Violation

Because we can determine at precisely which memory location a parallel model violation was detected, we can easily inform the user of which variable and even which aggregate offset caused the violation. To present this information in a form which is helpful to the user, we can also include the variable name, file name and line number of the original declaration from which the shared variable was generated as additional arguments to the `shcreate()` call.

This is a helpful alternative to the all-or-nothing loop parallelization scheme presented earlier in this thesis, and may prove useful for loops whose non-parallelizeability can be easily corrected.

7.3 Conclusion

We have explored a collection of techniques which form a unified framework for incremental, optimistic and above all, practical automatic parallelization. While almost all of these have seen some prior application in various settings, they have never before been brought together. We expect this work to have wide applicability because:

- It does not depend on a specific language or programming style. (While we do capitalize on the variable initialization semantics of C++, this would be more properly seen as an implementation detail.)
- It does not depend on a particular parallel platform. In particular, it has been demonstrated[JSS97] that optimizing a program for the NOW will not adversely affect its performance on more sophisticated platforms.
- It does not depend on any kind of user awareness of either the general principles of parallelism or the specific properties of his application, but gradually builds a "parallelism profile" as a local customization of the application through successive application to different inputs on various configurations of virtual machines.

What's more, the framework we describe can be implemented to any desired degree of sophistication. A more elaborate implementation which includes a more advanced dependence or pointer analysis will be less prone to illegal loop parallelization, while generating a parallel program which exhibits greater efficiency through less data sharing and less expended effort in runtime model verification. But even if these capabilities are not present, the generated program will still be safe, parallel, and ever "more correct" and reliable through the "data mining" of parallelism constraints using *all* future program input as a training set.

Bibliography

[AB95] G. Aloisio and M. Bochicchio. The Use of PVM With Workstation Clusters for Distributed SAR Data Processing. *Lecture Notes in Computer Science*, 919:570–581, 1995.

[ABC$^+$87] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An Overview of the PTRAN Analysis System for Multiprocessing. In *Proceedings of the 1st International Conference on Supercomputing*. June 1987.

[ACK87] J. Allen, D. Callahan, and K. Kennedy. Automatic Decomposition of Scientific Programs for Parallel Execution. In *Principles of Programming Languages*, pages 63–76, January 1987.

[ACP95] T. Anderson, D. Culler, and D. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[AFF97] A. Aiken, M. Fahndrich, and J. Foster. Flow-insensitive Points-to Analysis with Term and Set Constraints. Technical Report 97-964, Computer Science Division, University of California, Berkeley, August 1997.

[AI91] C. Ancourt and F. Irigoin. Scanning polyhedre with DO loops. In *Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

[AK87] J. Allen and K. Kennedy. Automatic Translation of Fortran Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[And93] L. Andersen. Binding-Time Analysis and the Taming of C Pointers. In *Programming Language Design and Implementation*, pages 47–58, 1993.

[ASGH95] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A Tool for Performing Parameterised Simulations using Distributed Workstations. In *4th IEEE Symposium on High Performance Distributed Computing*, pages 112–121, August 1995.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[AW97] T. Autrey and M. Wolfe. Initial Results for Glacial Variable Analysis. *Lecture Notes in Computer Science*, 1239, 1997.

[Bal90] V. Balasundaram. A Mechanism for Keeping Useful Internal Information in Parallel Programming Tools: The Data Access Descriptor. *Journal of Parallel and Distributed Computing*, 9(2):154–170, June 1990.

[Bar99] A. Baratloo. *Metacomputing on Commodity Computers*. PhD thesis, New York University, May 1999.

[BCCH97] M. Burke, P. Carini, J. Choi, and M. Hind. Interprocedural Pointer Alias Analysis. Technical Report 21055, IBM Corp., December 1997.

[Bre98] R. Brent. Random Number Generation and Simulation on Vector and Parallel Computers. In *Proceedings of EuroPar '98*, *Lecture Notes in Computer Science*, 1470, pages 1–20, September 1998.

[CCHK87] D. Callahan, A. Carle, M. Hall, and K. Kennedy. Constructing the Procedure Call Multigraph. *IEEE Transactions on Software Engineering*, 16(4):483–487, April 1987.

[CI95] B. Creusillet and F. Irigoin. Interprocedural Array Region Analyses. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'95)*, pages 4–1 to 4–15. Ohio State University, August 1995.

[CI97] B. Creusillet and F. Irigoin. Interprocedural Analyses of Fortran Programs. *Journal on Parallel Computing*, 24(3-4):629–648, June 1997.

[CK87] D. Callahan and K. Kennedy. Analysis of Interprocedural Side Effects in a Parallel Programming Environment. In *Proceedings of the 1st International Conference on Supercomputing*. June 1987.

[Coe94] F. Coelho. Experiments with HPF Compilation for a Network of Workstations. *Lecture Notes in Computer Science*, 797, 1994.

[CPW94] C. Cook, C. Pancake, and R. Walpole. Are Expectations for Parallelism Too High? A Survey of Potential Parallel Users. In *Supercomputing '94*, pages 126–133, November 1994.

[EM93] R. Eigenmann and P. McClaughry. Practical Tools for Optimizing Parallel Programs. In *1993 SCS Multiconference*, April 1993.

[GKT91] G. Goff, K. Kennedy, and C. Tseng. Practical Dependence Testing. In *Programming Language Design and Implementation*, pages 15–29, June 1991.

[GLL97] J. Gu, Z. Li, and G. Lee. Experience With Efficient Array Data Flow Analysis for Array Privatization. *ACM SIGPLAN Notices*, 32(7), July 1997.

[Gof97] G. Goff. *Practical Techniques to Augment Dependence Analysis in the Presence of Symbolic Terms.* PhD thesis, Rice University, May 1997.

[HAM$^+$95] M. Hall, S. Amarasinghe, B. Murphy, S. Liao, and M. Lam. Detecting Coarse-grain Parallelism Using an Interprocedural Parallelizing Compiler. In *Proceedings of Supercomputing '95*, December 1995.

[HDE$^+$93] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In *Workshop on Languages and Compilers for*

Parallel Computing, pages 406–420. *Lecture Notes in Computer Science*, 757, 1993.

[Hoe98] J. Hoeflinger. *Interprocedural Parallelization Using Memory Classificaiton Analysis*. PhD thesis, University of Illinois at Urbana-Champaign, 1998.

[IS99] A. Itzkovitz and A. Schuster. Multiview and Millipage—Fine-grain Sharing in Page-based DSMs. In *Operating Systems Design and Implementation*, February 1999.

[JSS97] D. Jiang, H. Shan, and J. Singh. Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-coherent Multiprocessors. In *Principles and Practice of Parallel Programming*, June 1997.

[KDCZ94] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 94 Usenix Conference*, pages 115–131, January 1994.

[Kel95] P. Keleher. Multiple Writers Considered Harmful. Technical Report CS-TR-3543, Dept. of Computer Science, University of Maryland at College Park, October 1995.

[KM90] K. Kennedy and K. McKinley. Loop Distribution with Arbitrary Control Flow. In *Supercomputing '90*, pages 407–416, New York, November 1990.

[KPS90] Z. Kedem, K. Palem, and P. Spirakis. Efficient Robust Parallel Computations. In *Proceedings of the 22nd ACM Symposium on Theory of Computing*, 1990.

[Kuc96] D. Kuck. *High Performance Computing*. Oxford University Press, 1996.

[Leu90] B. Leung. Issues on the Design of Parallelizing Compilers. Master's thesis, University of Illinois at Urbana-Champaign, 1990.

[Mas95] V. Maslov. Enhancing Array Dataflow Dependence Analysis With On-demand Global Value Propagation. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 265–269, 1995.

[Min93a] R. Minnich. Mether-NFS: A modified NFS which supports virtual shared memory. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 89–107, September 1993.

[Min93b] R. Minnich. Vector execute (vx). Available as `ftp://www.sarnoff.com/pub/mnfs/www/src/vx.tar.Z`, 1993.

[NGL96] T. Nguyen, J. Gu, and Z. Li. An Interprocedural Parallelizing Compiler and Its Support for Memory Hierarchy Research. *Lecture Notes in Computer Science*, 1033, 1996.

[PEH$^+$93] D. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford, and K. Faigin. Polaris: A New-Generation Parallelizing Compiler for MPP's. Technical Report 1306, Center for Supercomputing Research and Development, June 1993.

[PGH97] L. Parida, D. Geiger, and R. Hummel. Kona: A Multi-Junction Detector Using Minimum Description Length Principle. In *International Conference on Energy Minimization and Computer Vision.*, 1997.

[PHP98] Y. Paek, J. Hoeflinger, and D. Padua. Simplification of Array Access Patterns for Compiler Optimizations. *ACM SIGPLAN Notices*, 33(5):60–71, May 1998.

[PJS$^+$94] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *Proceedings of Summer 1994 USENIX Conference*, pages 137–151, Summer 1994.

[Pug94] W. Pugh. Counting Solutions to Presburger Formulas: How and Why. *ACM SIGPLAN Notices*, 29(6):121–134, June 1994.

[Ram98] N. Ramsey. Unparsing Expressions With Prefix and Postfix Operators. *Software—Practice and Experience*, 28(8), August 1998.

[Rau95] L. Rauchwerger. *Run-Time Parallelization: A Framework for Parallel Computation*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[Rep99] T. Reps. Undecidability of Context-sensitive Data-dependence Analysis. Technical Report TR-1397, Computer Sciences Department, University of Wisconsin 1999, March 1999.

[RP94] L. Rauchwerger and D. Padua. The Privatizing DOALL Test: A Run-time Technique for DOALL Loop Identification and Array Privatization. In *1994 ACM International Conference on Supercomputing*, pages 33–43, July 1994.

[SD] S. Sardesai and P. Dasgupta. Chime: A Versatile Distributed Parallel Processing Environment. Available as `http://www.eas.asu.edu/~calypso/frames/research_papers/chime/`.

[SH97] M. Shapiro and S. Horwitz. Fast and Accurate Flow-insensitive Points-to Analysis. In *Principles of Programming Languages*, pages 265–269, January 1997.

[Ste96] B. Steensgaard. Points-to analysis in Almost Linear Time. In *Principles of Programming Languages*, pages 32–41, January 1996.

[Str97] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[TP93] P. Tu and D. Padua. Automatic Array Privatization. In *Workshop on Languages and Compilers for Parallel Computing*, *Lecture Notes in Computer Science*, 768:500–521, August 1993.

[TWL$^+$91] S. Tjiang, M. Wolf, M. Lam, K. Pieper, and J. Hennessy. Integrating Scalar Optimization and Parallelization. In U. Banerjee, D. Gelernter,

A. Nicolau, and D. Padua, editors, *Workshop on Languages and Compilers for Parallel Computing*, August 1991.

[Zho94] M. Zhou. *Statical and Dynamical Analysis of Program Complexity*. PhD thesis, cole des Mines de Paris, September 1994.

[Zom96] A. Zomaya, editor. *Parallel and Distributed Computing Handbook*. McGraw-Hill, 1996.