Chapter 1
An Overview of the SUIF Compiler for Scalable Parallel Machines *

Saman Amarasinghe † Jennifer M. Anderson † Monica S. Lam †
Chau-Wen Tseng †

Abstract

We are building a compiler that automatically translates sequential scientific programs into parallel code for scalable parallel machines. Many of the compiler techniques needed to generate correct and efficient code are common across all scalable machines, regardless of whether its address space is shared or distributed. This paper describes the structure of the compiler, emphasizing the common analyses and optimizations. We focus on the three major phases of the compiler: parallelism and locality analysis, communication and synchronization analysis, and code generation.

1 Introduction

We are currently developing the SUIF (Stanford University Intermediate Format)[9] compiler system for researching compiler techniques for high-performance architectures. One of the major goals of this compiler is to automatically translate sequential dense matrix computations into efficient parallel code for large-scale parallel machines. We are targeting distributed address space (DAS) machines, such as the Intel Paragon and the IBM SP2, as well as shared address space (SAS) machines, such as the Stanford DASH multiprocessor[7] and the Kendall Square Research KSR-1. On SAS machines, the shared address-space is maintained in hardware using coherent caches.

At first glance, it seems much easier to compile to SAS machines as compared to DAS machines. On SAS machines, the programmer need not explicitly manage communication for non-local data. However, while it is relatively simple to get a program to run correctly, it is nontrivial to get scalable performance. As scalable parallel machines tend to have non-uniform memory access times, SAS machines can also benefit from the analyses and optimizations performed to minimize communication on DAS machines.

This paper discusses the fundamental issues in optimization and code generation needed for all scalable parallel machines. We give an overview of the algorithms used in the SUIF compiler to address these issues. The details of the individual algorithms can be found elsewhere[1, 3, 6, 8, 10]. We have currently implemented a complete compiler that generates code for SAS machines, and our DAS code generator is in progress. We are now experimenting with the compiler to validate the design and test its effectiveness.

*This research was supported in part by ARPA contracts N00039-91-C-0138 and DABT63-91-K-0003, an NSF Young Investigator Award, an NSF CISE Postdoctoral Fellowship in Experimental Science, and fellowships from Digital Equipment Corporation's Western Research Laboratory and Intel Corporation.

†Computer Systems Laboratory, Stanford University, Stanford, CA 94305-4070

The inputs to our compiler are sequential FORTRAN-77 or C programs. The source programs are first translated into the SUIF compiler's intermediate representation. Various program analysis and optimization passes have been implemented as independent programs that operate on the SUIF representation. The optimized and parallelized SUIF program is then converted to C and is compiled by the native compiler on the target machine. The C program contains calls to a portable run-time library, which is linked in by the native compiler. The analysis and optimization passes are divided into four categories, as follows.

Symbolic Analysis. The compiler first performs a series of symbolic analyses on the SUIF program to extract information necessary for subsequent parallelization and optimization passes. These analyses include scalar variable analysis such as constant propagation, induction variable identification and forward propagation, as well as data dependence and data flow analysis on array variables. We have also implemented a pass that recognizes reductions to both scalar and array variables. Our intraprocedural version of these optimizations are fully functional, and we are experimenting with our interprocedural version[4].

Parallelism and Locality Analysis. The parallelism and locality analysis phase first identifies and optimizes the loop-level parallelism in the program. The compiler then maps the computation to the processors and determines a data layout for the array variables such that parallelism is maximized while minimizing communication.

Communication and Synchronization Analysis. The communication and synchronization analysis phase uses the data mapping information to identify accesses to non-local data. This information is used to generate *send* and *receive* messages for accessing the data on DAS machines, and to reduce synchronization costs on SAS machines.

Code Generation. The code-generator phase is machine-dependent — it is responsible for carrying out the transformations required by the previous passes. The code generator first schedules the parallel loops so that each processor executes its allocated iterations, then inserts the necessary synchronization and communication code. It also changes the array accesses using the data mappings calculated in the parallelism and locality phase.

All but the final code generation phase are mainly analysis phases. They do not alter the code in such a way as to obscure or destroy the analysis performed by subsequent passes. Information is propagated between passes by annotations on the compiler's internal representation of the code. The symbolic analysis passes are not discussed in the remainder of the paper — instead we focus on the last three categories described above.

2 Parallelism and Locality Analysis

To achieve good performance on scalable parallel systems, programs must make effective use of the computer's memory hierarchy as well as its ability to perform computation in parallel. As a result, reducing interprocessor communication by increasing the locality of data references is an important optimization for achieving high performance on these machines. The compiler determines the processor location and the order in which the computation is executed, as well as the layout of the data arrays on each processor so that temporal and spatial locality are optimized at both the processor and cache levels.

The need to optimize the data layout for DAS machines is well understood. For example, the HPF language[5] is designed so that the user can explicitly guide this optimization process. While it is not necessary for the compiler to distribute data across the processors explicitly on SAS machines, changing the data layout can greatly improve the cache

```
      DOUBLE PRECISION X(M,M), Y(M,M)
      DO 10 J1 = 2, M
        DO 10 I1 = 1, M
10        X(I1,J1) = X(I1,J1-1) + Y(I1,J1)
      DO 20 J2 = 1, M
        DO 20 I2 = 2, M
20        X(I2,J2) = X(I2,J2) + Y(I2-1,J2)
```

FIG. 1. *Example Code.*

performance. Data in caches are transferred in units known as cache lines. *False sharing* occurs whenever two processors use different data that are co-located on the same cache line. Caches typically have a small set associativity; that is, each memory can only be cached in a small number of cache locations. *Interference* occurs whenever different memory locations contend for the same cache location. One simple way to enhance spatial locality, and to minimize false sharing and cache interference, is to restructure the array so that the major regions of data accessed by a processor are contiguous in memory. Both SAS and DAS machines thus benefit from optimizing the data layout for locality.

Our compiler algorithm to maximize interprocessor parallelism and locality is divided into two phases. First a local analysis phase optimizes parallelism at the loop level[10]. It reorders the computation to discover the largest granularity of parallelism using unimodular code transformations (e.g. loop interchange, skewing and reversal). A global analysis phase then examines all loop nests together to determine the best overall mapping of data and computation across the processors of the machine[3].

The global analysis phase represents the mappings of computation and data onto the processors as the composition of two functions: an affine function that maps onto a virtual processor space, and a folding function that maps the virtual space onto the physical processors of the target machine. This representation is able to capture a wide range of computation and data assignments. For example, this model represents a superset of the data mappings available to HPF programmers. The data and computation mappings are represented as systems of linear inequalities, which are then read as input by the subsequent compiler passes.

The compiler tries to find data and computation mappings such that the amount of interprocessor communication is as small as possible, while maintaining sufficient parallelism to keep the processors busy. The compiler tries to align the arrays and loop iterations to decrease non-local data accesses, while exploiting all the parallelism available in the computation. If that is not possible without major data reorganization, the compiler will trade off some degrees of parallelism to choose a parallelization that incurs no such data reorganization communication. Finally, when communication is needed, the compiler uses a greedy algorithm to insert communication into the least frequently executed parts of the program.

For example, consider the FORTRAN code shown in Figure 1. Because of a data dependence between the references $X(I_1, J_1)$ and $X(I_1, J_1 - 1)$, the J_1 loop in the first loop nest is sequential. The remaining loops can all be executed in parallel. Communication is minimized when the two arrays are distributed by row. The corresponding computation mapping is to distribute iterations of the I_1 loop across the processors in the first loop nest, and iterations of the I_2 loop in the second loop nest. The compiler aligns arrays X and Y, and groups blocks of rows together to fit the number of processors. Communication is incurred due to the access $Y(I_2 - 1, J_2)$ in the second loop nest; however, the communication

Loop nest 1	$2 \leq J_1 \leq M$ $1 \leq I_1 \leq M$ $p * b + 1 \leq I_1 \leq (p+1) * b$
Loop nest 2	$1 \leq J_2 \leq M$ $2 \leq I_2 \leq M$ $p * b + 1 \leq I_2 \leq (p+1) * b$

FIG. 2. *The computation mapping for the example code*

is inexpensive since it only occurs at block boundaries.

The compiler represents the computation mappings for the two loop nests by the systems of linear inequalities shown in Figure 2. In the figure, p is the processor id, n is the number of processors, and the block size $b = \lceil \frac{M}{n} \rceil$.

3 Communication and Synchronization Analysis

Compilers for DAS machines produce *single-program, multiple-data* (SPMD) programs with explicit interprocessor communication in the form of calls to *send* and *receive* library routines. Since *receive* routines stall a processor until the matching message arrives, messages also serve as synchronization. In comparison, traditional compilers for SAS machines adopt a *fork-join* model, inserting global barriers at the end of each parallel loop to prevent potential data races. No explicit communication code is necessary. Despite these differences, as shown below, both SAS and DAS machines benefit from similar analysis and optimization in the SUIF compiler.

On DAS machines the communication analysis is required for correctness. Using the data and computation mappings derived from the parallelism and locality phase, the SUIF compiler can determine when remote data are accessed, thus requiring communication. By composing data and computation mappings and array references as a system of linear inequalities, non-local accesses and the identity of the sending and receiving processors may be calculated by applying Fourier-Motzkin elimination[1].

The SUIF compiler also uses array data-flow information to optimize the communication whenever possible. Array data-flow information identifies the processor that produces the values desired, thus data can be sent from the producer to the consumer as soon as the data are produced. This information can also be used to eliminate redundant communication and to aggregate messages so as to amortize the message handling overheads. All such communication optimizations increase the communication buffer space usage. We are currently developing algorithms that trade off between time and space to derive a globally efficient strategy.

On SAS machines communication analysis is not required to ensure correctness, but is used to optimize synchronization. Many parallelized programs consist of a large number of parallel loops, each of which does not contain much computation. The resulting profusion of barriers incurs high overhead and inhibits parallelism. By taking advantage of the fact that data and computation mappings are calculated at compile time, we can use information on when and where communication is necessary to eliminate unnecessary barrier synchronization or replace them with efficient point-to-point synchronization.

Consider Figure 1. Standard shared-memory compilers would insert a barrier between the two loop nests. By applying communication analysis for the data and computation mapping selected, the SUIF compiler determines that none of the data defined in the first loop nest are used by other processors, allowing it to eliminate the unnecessary barrier.

4 Communication and Synchronization Analysis

Compilers for DAS machines produce *single-program, multiple-data* (SPMD) programs with explicit interprocessor communication in the form of calls to *send* and *receive* library routines. Since *receive* routines stall a processor until the matching message arrives, messages also serve as synchronization. In comparison, traditional compilers for SAS machines adopt a *fork-join* model, inserting global barriers at the end of each parallel loop to prevent potential data races. No explicit communication code is necessary. Despite these differences both SAS and DAS machines benefit from similar analysis and optimization in the SUIF compiler, as discussed below.

On DAS machines the communication analysis is required for correctness. Using the data and computation mappings derived from the parallelism and locality phase, the SUIF compiler can determine when remote data are accessed, thus requiring communication. By composing data and computation mappings and array references as a system of linear inequalities, non-local accesses and the identity of the sending and receiving processors may be calculated by applying Fourier-Motzkin elimination[1].

The SUIF compiler also uses array data-flow information to optimize the communication whenever possible. Array data-flow information identifies the processor that produces the values desired, thus data can be sent from the producer to the consumer as soon as the data are produced. This information can also be used to eliminate redundant communication and to aggregate messages so as to amortize the message handling overheads. All such communication optimizations increase the communication buffer space usage. We are currently developing algorithms that trade off between time and space to derive a globally efficient strategy.

On SAS machines communication analysis is not required to ensure correctness, but is used to optimize synchronization. Many parallelized programs consist of a large number of parallel loops, each of which does not contain much computation. The resulting profusion of barriers incurs high overhead and inhibits parallelism. By taking advantage of the fact that data and computation mappings are calculated at compile time, we can use information on when and where communication is necessary to eliminate unnecessary barrier synchronization or replace them with efficient point-to-point synchronization.

Consider Figure 1. Standard shared-memory compilers would insert a barrier between the two loop nests. By applying communication analysis for the data and computation mapping selected, the SUIF compiler determines that none of the data defined in the first loop nest are used by other processors, allowing it to eliminate the unnecessary barrier.

5 Code Generation

The code generator carries out the transformations determined by the previous analysis phases. This phase schedules iterations of parallel loops according to the computation mapping calculated in the parallelism and locality phase. It translates array subscripts in the original code to accesses for the new array layout. It also inserts the necessary communication and synchronization code.

When generating code for a SAS machine, the compiler will first generate SPMD loop nests using the computation mapping calculated for each parallel loop in the parallelism and locality phase (as shown in Figure 2). The compiler generates the bounds of each

```
//  p = processor_id(), n = num_processors()
b = (M+n-1)/n
double X(b, M, n), Y(b, M, n)
for J1 = 2 to M do
  for I1 = p*b+1 to min((p+1)*b, M) do
    X(I1-p*b, J1, p+1) = X(I1-p*b, J1-1, p+1) + Y(I1-p*b, J1, p+1)
for J2 = 1 to M do
  if((p > 0) and (p*b < M))
    X(1, J2, p+1) = X(1, J2, p+1) + Y(b, J2, p)
  for I2 = p*b+2 to min((p+1)*b, M) do
    X(I2-p*b, J2, p+1) = X(I2-p*b, J2, p+1) + Y(I2-p*b-1, J2, p+1)
```

FIG. 3. *Output Pseudo-code for the Example for SAS Machines.*

```
//  p = processor_id(), n = num_processors()
b = (M+n-1)/n
double X(b, M), Y(b, M), t(M)
for J1 = 2 to M do
  for I1 = p*b+1 to min((p+1)*b, M) do
    X(I1-p*b, J1) = X(I1-p*b, J1-1) + Y(I1-p*b, J1)
if((p+1)*b < M)
  for k = 1 to M do
    t(k) = Y(b, k)
  call send(p+1, t, M)
if(p > 0)
  call recv(p-1, t, M)
for J2 = 1 to M do
  if((p > 0) and (p*b < M))
    X(1, J2) = X(1, J2) + t(J2)
  for I2 = p*b+2, to min((p+1)*b, M) do
    X(I2-p*b, J2) = X(I2-p*b, J2) + Y(I2-p*b-1, J2)
```

FIG. 4. *Output Pseudo-code for the Example for DAS Machines.*

loop in the parallel SPMD code by projecting the polyhedron represented by this system of inequalities onto a lower-dimensional space[2]. The generated code is parameterized by the number of processors; each processor gets the number of processors and its processor id from calls to the run-time library.

Next, the code generator changes the original data declarations according to the data mapping derived by the parallelism and locality optimizer. For the example, the data layouts of arrays X and Y are changed from (M, M) to (b, M, n), thus the access $X(I_1, J_1)$ becomes $X\left(J_1 \bmod b, I_1, \frac{J_1}{b}\right)$. If the new organization has more dimensions than the original, the address calculations now include division and modulo operations. A set of special optimizations are used to eliminate most of these division and modulo operations‘. Finally, the compiler generates the barriers and lock code as specified by the communication and synchronization analysis phase. The compiler-generated SAS code for the example is shown in Figure 3.

For DAS machines, the SPMD loop nests are generated in the same manner as for the SAS machines. In addition, the code generator allocates space locally on each processor for the distributed arrays. Global array addresses in the original program are translated into local addresses. For the example, the data layouts of arrays X and Y are changed from (M, M) to (b, M), thus the access $X(I_1, J_1)$ becomes the local memory access $X(J_1 \bmod b, I_1)$. The compiler must also manage buffers for non-local data. Finally, the code generator generates necessary *send* and *receive* operations. Again, linear inequalities are used to represent the data accesses. Projections of these inequalities are used to generate the send and receive code for each processor[1]. The compiler-generated DAS code for the example is shown in Figure 4.

6 Conclusions

This paper shows how the compiler techniques used to generate correct and efficient code are common to all scalable machines, independent of whether the address space is shared or distributed. We discussed the algorithms used in the SUIF compiler to address the major issues in compiling for these machines. Our research results suggest that it is possible for compilers to automatically translate sequential dense matrix computations into efficient parallel code.

References

[1] S. Amarasinghe and M. Lam, *Communication optimization and code generation for distributed memory machines*, in Proceedings of the SIGPLAN '93 Conference on Program Language Design and Implementation, Albuquerque, NM, June 1993.

[2] C. Ancourt and F. Irigoin, *Scanning polyhera with do loops*, in Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Williamsburg, VA, Apr. 1991.

[3] J. Anderson and M. Lam, *Global optimizations for parallelism and locality on scalable parallel machines*, in Proceedings of the SIGPLAN '93 Conference on Program Language Design and Implementation, Albuquerque, NM, June 1993.

[4] M. W. Hall, S. Amarasinghe, B. Murphy, and M. S. Lam, *Interprocedural analysis for parallelization: Design and experience*, in Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, CA, Feb. 1995.

[5] High Performance Fortran Forum, *High Performance Fortran language specification, version 1.0*, Tech. Rep. CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, Jan. 1993.

[6] S. Hiranandani, K. Kennedy, and C.-W. Tseng, *Compiling Fortran D for MIMD distributed-memory machines*, Communications of the ACM, 35 (1992), pp. 66–80.

[7] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy, *The DASH prototype: Implementation and performance*, in Proceedings of the 19th International Symposium on Computer Architecture, Gold Coast, Australia, May 1992.

[8] D. Maydan, S. Amarasinghe, and M. Lam, *Array data-flow analysis and its use in array privatization*, in Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages, Charleston, SC, Jan. 1993.

[9] Stanford SUIF Compiler Group, *SUIF: A parallelizing and optimizing research compiler*, Tech. Rep. CSL-TR-94-620, Computer Systems Lab, Stanford University, May 1994.

[10] M. E. Wolf, *Improving Locality and Parallelism in Nested Loops*, PhD thesis, Dept. of Computer Science, Stanford University, Aug. 1992.