MetaMP Approach to Parallel Programming

Steve W. Otto and Michael Wolfe

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 92-016

August 1992

Published in: "Frontiers '92: Frontiers of Massively Parallel Computation" - IEEE.

The MetaMP Approach to Parallel Programming

Steve W. Otto
Computer Science and Engineering
Oregon Graduate Institute
Beaverton, OR 97006

Michael Wolfe
Computer Science and Engineering
Oregon Graduate Institute
Beaverton, OR 97006

Abstract

We are researching techniques for programming large-scale parallel machines for scientific computation. We use an intermediate-level language, MetaMP, that sits between High Performance Fortran (HPF) and low-level message passing. We are developing an efficient set of primitives in the intermediate language and we are investigating compilation methods that can semi-automatically reason about parallel programs. The focus is on distributed memory hardware. Our work has many similarities with HPF efforts although our approach is aimed at shorter-term solutions. We plan to keep the programmer centrally involved in the development and optimization of the parallel program.

1 Introduction

This paper is concerned with programming large-scale parallel machines for scientific computation. Our approach is twofold: we are investigating compilation methods that can semi-automatically reason about parallel programs, and we are developing an efficient set of primitives that abstract distributed-memory hardware.

A first version of such primitives has been developed and is called MetaMP [1]. With this, the user writes a parallel SPMD program for a distributed memory computer, with explicit communication. The language explicitly supports common modes of broadcast, block, guard strip, and random access communications, which have been shown to be useful for a wide variety of algorithms [2]. Because the communication is explicit, the MetaMP compiler need not be so aggressive, and the communication cost will be clear to the user. The user has the option to exploit novel and powerful parallel algorithms that have no sequential counterpart and whose communication patterns are not obvious to a compiler.

We have exploited some techniques common in object-oriented programming, namely, self-describing data structures to implement the primitives. On the distributed-memory machine, each piece of a distributed array structure is locally self-describing. The current version of MetaMP is based on C and compiles down to a commercially available parallel message-passing system, Express.

Several realistic applications have been written in MetaMP. These include a parallel Multigrid solver for partial differential equations, an N-body gravity program, Gaussian elimination, and a cyclic Jacobi eigensolver. MetaMP has similarities to Hypertasking [3] and to systems such as Dino [4] and BLAZE [5].

Though MetaMP has several important advantages over writing programs directly in the message passing system, it is not a complete answer. Currently, programming in MetaMP is manual and the user must carefully reason about the parallel program in order to avoid mistakes. Therefore, we are studying the feasibility of a "proof" tool that can prove the equivalence of a MetaMP program and a certain sequential program. Once correctness is achieved, the user would also like some guidance as to performance. We plan a "perf" tool that will give a performance prediction of the parallel program using a linear performance model. We have already developed such a tool for our loop restructuring system, Tiny [6].

Our approach has many similarities with efforts to define and implement High Performance Fortran (HPF) [7, 8]. An essential difference is that we plan to keep the programmer centrally involved in the development and optimization of the parallel program. The optimizations used for distributed machines can be substantially different than those used for other computers, and knowledge of the communication patterns is necessary for efficient execution. Automatic compiler analysis can never replace or even approach manual program design, because compilers are necessarily restricted to *correctness-preserving* optimizations; users, on the other hand, can determine that a different formulation is equally correct. Successful parallel program development must consist of a non-trivial *dialogue* between the user and the compilation system. The system can help pinpoint performance problems, aid in mechanical aspects of program design, and help the user focus on algorithmic choices. Nonetheless, it is up to the user to guide the system towards the parallel program.

2 Interactive Programming Tools

Given the parallel language MetaMP and its compiler, a user will want to write correct but efficient parallel algorithms. We envision three tools to aid users develop and tune parallel algorithms for MetaMP.

```
for (i = 0; i < M; ++i){
  for (j = 0; j < N; ++j){
    for (k = 0; k < L; ++k){
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Figure 1: Matrix multiplication loops.

```
for (is = 0; is < $procs C[*][]$; ++is){
  for (i=$lower C[:is][]$;i<$upper C[:is][]$;++i){
    for (j = 0; j < N; ++j){
      for (ks = 0; ks < $procs B[*][]$; ++ks){
       for (k=$lower B[:ks][]$;k<$upper B[:ks][]$;++k){
         C[i][j] += A[i][k] * B[k][j];
       }
      }
    }
  }
}
```

Figure 2: Strip-mined version of Figure 1.

2.1 Program Development

While our experience with MetaMP has been with algorithms written explicitly for this model, many parallel programs are written by migrating or evolving sequential programs or shared-memory parallel programs into a distributed memory program. Much of the migration process is *creative*, such as deciding how to distribute the data structures, while the rest of the process is *mechanical*, such as strip-mining, reordering and restructuring loops. We have extensive experience with high-level loop restructuring, initially aimed at uncovering parallelism in sequential programs. In this context, we want to use the restructuring process to find an efficient form of a parallel program.

For example, a distributed matrix multiplication algorithm could have been derived through the following restructuring process. Start with the sequential algorithm in Figure 1. Now suppose we decide to block-distribute each of the three arrays in the first dimension. The `i` and `k` loops need to be *strip-mined*, producing the program in Figure 2. One of the two strip loops (`is` or `ks`) must correspond to the "parallel" loop; the user here chooses `is`, based on estimation of the communication required and because it carries no *dependence relations*. In the MetaMP SPMD environment, the "parallel" loop disappears, since the same program is executed by each processor and so the loop is implicit, but for now we leave it in place. This version of the parallel program uses only local references for the `C` and `A` arrays, but must communicate the `B` array; in fact processor `ks` must broadcast its portion of `B` for each iteration of the `ks` loop.

The frequency of the broadcasts can be reduced by moving the `ks` loop outwards, as in Figure 3; this is allowed by classical *loop interchanging* since neither the `j` nor the `i` loop carry any dependence relations [9].

```
for (is = 0; is < $procs C[*][]$; ++is)   $ parallel $
  for (ks = 0; ks < $procs B[*][]$; ++ks){
    $ broadcast B[:ks][] $
    for (i=$lower C[:is][]$;i<$upper C[:is][]$;++i){
      for (j = 0; j < N; ++j){
       for (k=$lower B[:ks][]$;k<$upper B[:ks][]$;++k){
         C[i][j] += A[i][k] * B[k][j];
       }
      }
    }
  }
}
```

Figure 3: Loop interchanging to reduce communications.

A much more efficient distributed algorithm is derived, however, by *rotating* the **ks** loop with respect to the (implicit) **is** loop [10]. The order in which the **ks** blocks are executed is different for each processor:

processor **is**	order **ks**
0	0, 1, 2, 3
1	1, 2, 3, 0
2	2, 3, 0, 1
3	3, 0, 1, 2

As can be seen from the table, each processor starts out using its local block of the `B` array, as indexed by `ks`. Loop rotation is legal in this case because the loop-carried dependence relations are for associative reductions. Note that the parallel program does accumulate the result in a different order than the original sequential program, so may accumulate a different roundoff error. The communication is now nearest neighbor around a ring, signified in MetaMP by the `roll` primitive. After eliminating the implicit "parallel" loop and changing to the notation of MetaMP, we have exactly the program given in [1] and shown in Figure 4. A set of pictures describing the parallel algorithm is given in Figure 5.

The goals of the restructuring process were to maximize parallelism, minimize communication, and confine communication as much as possible to efficient patterns. Had the dependence relations prevented the rotation transformation, we could still use the `roll` pattern provided in MetaMP to execute the loop in wavefront fashion [10].

This example demonstrates the power and importance of the restructuring process: the parallel program is not a simple distributed version of the original sequential algorithm, and the restructuring required and communication patterns used will not be discovered by automatic compilers. Moreover, the restructuring process is not a simple application of current parallel compiler optimization algorithms. New transformations are required, and the user must be "in the loop."

```
$ processors p1 $  // 1D mesh of processors
int C[M][N]; $ array C[M:p1][N] $
int A[M][L]; $ array A[M:p1][L] $
int B[L][N]; $ array B[L:p1][N] $
main()
{
  $ Alloc $    // Allocate distributed arrays
  ...          // read in arrays, init C to 0

  for (l=0; l<$procs B[*][]$; ++l) {
    for (i=0; i<M; ++i) ${$     $ doMine C[*][] $
      for (j=0; j<N; ++j) {
        for (k=0; k<L; ++k) ${$ $ doMine B[*][] $
          C[i][j] += A[i][k] * B[k][j];
        $}$
      }
    $}$
    $ roll B[*][] $
  }
  ...          // write out C
}
```

Figure 4: **mat.mmp**: A MetaMP program for the matrix multiplication algorithm of Figure 3.

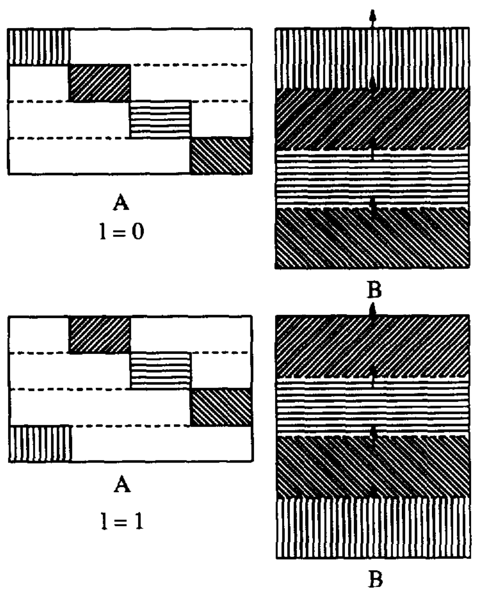

Figure 5: The first two (of the total of four) roll cycles for matrix multiplication on four processors. Arrows denote the direction of dataflow of **B**, the shading represents which partial rows of **A** are combining with which partial columns of **B**. Elements of the same shading pattern are combined. "1" refers to the loop index **l** in Figure 4.

2.2 Correctness

In MetaMP, proving correctness of the parallel program is complicated by the addition of explicit communication statements. Such a parallel program may have no simple sequential counterpart, and so it is impossible to use the HPF approach, where erasing the data distribution and parallel syntax gives an equivalent scalar program, which can then be debugged. Let us define the ***natural sequentialization*** of a MetaMP program as the same algorithm with one processor. We are designing analysis algorithms that will be encoded in a tool to find the essential differences between the parallel MetaMP program and its natural sequential counterpart. The key will be to point out to the user where the parallel program will differ from the scalar program; this will not necessarily be an error, but will inform the user where the programs exhibit essential differences that may lead to numerically different results. The basic idea is similar to that proposed by the PTOOL project [11], but our effort is focused on parallel programs with explicit communication. Dependence analysis for the parallel program is complicated by the fact that the parallel program only preserves dependence between processors when there is explicit communication. In MetaMP, the analysis is feasible since the communication patterns are explicit in the program and known to the compiler.

A good example of this is Gauss-Seidel relaxation [2]. In the parallel case this is typically written so as to minimize the number of communication calls. This implies that some of the values coming from remote processors may be old – that is, to minimize communications we have violated a dependence relation. A Jacobi solver doesn't have this problem since it explicitly uses values only from the previous iteration, but on the other hand it does not converge as rapidly as Gauss-Seidel. The typical parallel implementation of Gauss-Seidel is a mixed relaxation, and does not correspond to the sequential program; yet since it can converge faster and uses less memory than strict Jacobi, and uses less communication than strict parallel Gauss-Seidel, it has measurable advantages.

For this type of program, our analysis tool will determine that the natural sequential program has loop-carried dependence relations. The parallel program can only satisfy interprocessor dependence relations at communication points; thus the parallel program satisfies the dependence relations only within each processor domain. Between processors, the dependence relations are carried by the communication statements. This difference will be presented to the user, who can then determine that, yes, in fact the parallel program is different but nonetheless correct (converges to the same values).

Note that to be useful, this analysis will be significantly different than simple dependence analysis. One of the lessons learned from the PTOOL project was that if the user is swamped with voluminous dependence information, potential problems will be hidden and often missed. In the matrix multiplication analysis, for example, the tool must understand associative reductions and the reordering implied by the loop rotation.

```
Floating Point Ops
n/6 + n^2/2 + n^3/3
Memory Ops
 - n/6 + 3*n^2/2 + 2*n^3/3
Stride-1 in inner loop
 - n/3 + n^3/3
Non-stride-1 in inner loop
5*n/6 + n^2 + n^3/6
Invariant in inner loop
 - 2*n/3 + n^2/2 + n^3/6

Parsed ch
*Count   Rowwise  Colwise  Write    Msgs     Quit     Xcape
```

Figure 6: Operation counts assuming C-like array layout.

2.3 Performance

One of the main reasons that vectorizing compilers have proven successful is that the user can easily predict the performance of the generated code by looking at the report of the vectorized loops. We take the same approach, keeping the user "in the loop" for critical performance tuning. Our approach is based on a *linear performance model*, where the performance of the system is modeled on the counts and coefficients of certain critical parameters of the algorithm.

We have already demonstrated the feasibility of symbolically counting critical parameters in an algorithm in the Tiny program restructuring tool. As an example, a Cholesky decomposition program was automatically, symbolically analyzed to count the frequency of the following parameters:

- Floating Point Ops - floating point additions and multiplications
- Memory Ops - total array element loads and stores
- Stride-1 in inner loop - loads/stores that are to consecutive array elements in memory.
- Non-stride-1 in inner loop - loads/stores with stride > 1.
- Invariant in inner loop - loads/stores invariant in inner loop.

Other counts are also made, but do not apply in this example. Assuming a C-like rowwise storage order, the counts are as shown in Figure 6 (this figure is taken from a screen dump of Tiny).

With this technology we can count frequency of different communication operations as well as processor activity. This performance prediction module can be built into the restructuring tool to provide instant feedback on the benefits of the optimization process.

References

[1] S.W. Otto. MetaMP: A higher level abstraction for message-passing programming. Technical Report CS/E 91-003, Oregon Graduate Institute of Science and Technology, 1991. This preprint is electronically available (in PostScript) via anonymous ftp from `cse.ogi.edu`, in directory `pub/tech-reports`.

[2] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, Englewood Cliffs, NJ, 1988.

[3] M. Baber. Hypertasking support for dynamically redistributable and resizable arrays on the iPSC. In *The Sixth Conference on Hypercube Concurrent Computers and Applications*, pages 59–66. IEEE Computer Society Press, 1991.

[4] M. Rosing, R.B. Schnabel, and R. Weaver. Dino: Summary and examples. In *The Third Conference on Hypercube Concurrent Computers and Applications*. ACM Press, 1988.

[5] P. Mehrotra and J. Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5(3):339–61, 1987.

[6] Michael Wolfe. The Tiny loop restructuring research tool. In *Proc. 1991 International Conf. on Parallel Processing*, volume II, pages 46–53, St. Charles, IL, August 1991. Penn State Press.

[7] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report Rice COMP TR90-141, Dept of Computer Science, Rice University, 1990.

[8] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran – a Fortran language extension for distributed memory systems. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-time Environments for Distributed Memory Machines*. Elsevier Press, 1992.

[9] John R. Allen and Ken Kennedy. Automatic loop interchange. In *Proc. SIGPLAN '84 Symp. on Compiler Construction*, pages 233–246, Montreal, Canada, June 1984.

[10] Michael Wolfe. Loop rotation. In David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, pages 531–553. MIT Press, Boston, 1990.

[11] J. R. Allen, Donn Baumgartner, Ken Kennedy, and Allan Porterfield. PTOOL: A semi-automatic parallel programming assistant. In Kai Hwang, Steven M. Jacobs, and Earl E. Swartzlander, editors, *Proc. 1986 International Conf. on Parallel Processing*, pages 164–170, St. Charles, IL, August 1986.