A Software Tool For Building Supercomputer Applications

By

Dennis Gannon, Daya Atapattu, Mann Ho Lee & Bruce Shei
Department of Computer Science
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 224

A Software Tool
For Building Supercomputer Applications
by
Dennis Gannon, Daya Atapattu, Mann Ho Lee & Bruce Shei

August, 1987

This report is based on work supported in part by the Air Force Office of Scientific Research Grant No. AFOSR-86-0147, and University of Illinois Subcontract No. 87-102.

A Software Tool For Building Supercomputer Applications.

Dennis Gannon
Daya Atapattu
Mann Ho Lee
Bruce Shei

Department of Computer Science
Indiana University
Bloomington, Indiana

ABSTRACT

We describe a software tool that consists of an interactive environment for helping users restructure programs to optimize execution on parallel/vector multiprocessors. The system is used to help programmers fine tune codes that have already been passed through an automatic parallelizing system or codes have been designed from the start from new parallel algorithms. In particular, programs optimized for one machine can be easily reoptimized for another using this system. To accomplish this task, the tool provides mechanisms to give the programmer feedback concerning the potential performance of his code on the chosen target machine and allows the user an interactive means to guide the system through a sequence of automatic program transformations.

1 INTRODUCTION

Five years ago the subject of Parallel Computation was an exotic sub-field of computer science that consisted largely of theoretical studies of potential parallel algorithm performance, a few university hardware projects, even fewer software design efforts and (with the exception of Denelcor) no industrial products. Five years later, we now find that all U.S. supercomputers, and over one dozen mini-supercomputer vendors have entered the market with some form of general purpose parallel processing system. It is expected that every computer company will offer a scalable multiprocessor by the end of the decade.

Unfortunately, very few of these systems provide a software environment for building parallel programs that goes beyond a standard sequential language compiler and a microtasking library to support parallelism. None of our most widely used programming languages (C, Fortran, Lisp) have been given official extension to support concurrency and no two vendors agree on any of the unofficial extensions. The task of porting parallel codes from one machine to another now involves large amounts of recoding to make a program work, and large amounts of restructuring of the algorithm organization to make it work well.

The primary concern of the vast majority of users is to be able to exploit the power of a supercomputer without sacrificing portability. Consequently, these users want an automatic system such as those provided by Pacific-Sierra or KAI which will provide substantial improvement in a code without any effort by the programmer.

On the other hand, there are still a large number of hardy users of these new systems who are not content to live with the results of an automatic vectorizer or parallelizer. These are the users who are willing to invest a "reasonable" amount of extra effort if it might mean a doubling of performance beyond what the automatic system delivers. The key to this extra performance is the cost. How much is a "reasonable" amount of extra work?

In this paper we describe a programming tool designed to help users of parallel supercomputers retarget and optimize application codes. In a sense, the system can be viewed as a tool to help users "fine-tune" the output of an automatic system or, if he or she has been so inspired, optimize the design of a new parallel algorithm.

The system is an interactive program editing and transformation system that helps the user with this task. Each program that enters the system is completely parsed and all data dependences are recorded. The user then works with the system to restructure his code to a form suitable for a given target architecture. If the target is known to the system, it monitors the users transformations to the code. If the user attempts to transform the program in violation of the original semantics of the code he is warned that a change in the meaning of his program has taken place. At any time the user can ask the system to tell him what legal parallelizing transformation can be applied to a segment of selected code. More important, he can ask the system to make the program modifications the user desires. In this mode, the user is assured of the correctness of the changes in his code.

In its current form, the system, known as the Blaze Editor or "Bled", can support either FORTRAN (with Cedar and Alliant 8x extensions) or Blaze (a Pascal based functional language designed by Mehrotra and Van Rosendale [14]). In the future we plan to support C, C++ and Cedar Parallel C [9]. The target machines supported currently include the BBN Butterfly, the Alliant FX/8 and the Cedar System [7].

This tool is one part of a much larger programming environment known as the Faust Project. This effort, based at the Center for Supercomputer Research and Development in Urbana, Illinois has designed a common software platform for a number of programming tools including a performance analysis package, a program debugger, Bled, and a graphics based program maintenance system. All the software has been written to use the X system from the MIT project, Athena and, therefore, it will run on any Unix-based workstation.

In this paper we describe the current Bled system as well as two important extensions that are being added at the time of this writing.

One extension is a performance prediction package that can be invoked from within Bled to help the user choose which formulation of his algorithm will run best. There are two components to this performance prediction package. First is a code generation estimator that can give the user feedback in the form of estimates of such quantities as the ratio of vector instructions to data movement, or the amount of code devoted to synchronization overhead. Second, this package provides estimates of cache behavior and local memory utilization.

The second extension is a portable runtime environment that supports a dynamic "microtasking" facility that incorporates ideas from the Argonne Schedule package [5] and the MIT multilisp system [10].

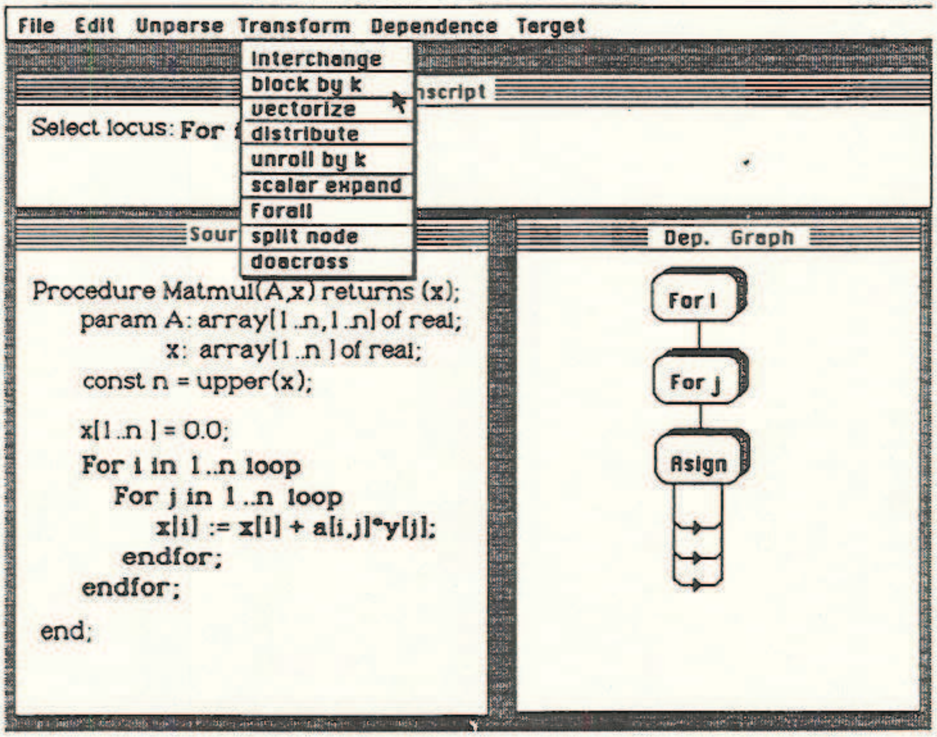
2 A SAMPLE USER SCENARIO

To illustrate how a user would interact with this system we shall step through a very simple example. The Blaze subroutine below is a simple matrix times vector routine.

```
Procedure MatVec(n,A,x) returns: y;
param A: array[1..n, 1..n] of real;
    x,y: array[1..n] of real;
    n: integer;
begin
    for j in 1..n loop
        for i in 1..n loop
            y[i] := y[i]+A[i,j]*x[j];
        end;
    end;
end;
```

The user loads this program into the system as if he were entering a text editor. The result is a window displaying the program and a list of menu headers (as illustrated in Figure 2.1). The first thing he may wish to do is to tell the system which machine is the intended target of this optimization. Currently this menu only lists three active choices: the BBN butterfly, the Alliant FX/8 and the Illinois Cedar. (We plan to extend this list to include the IBM RP3, the CRAY 2, the Connection Machine, and the ETA-10 during the next two years.)

The significance of having the user tell the system about the choice of target is threefold. First, and most obvious, we would like to have the system generate code for the given target. We will say more about this later. Second, it is important that program transformations that are inappropriate for the target machine be disabled. Third, and of most immediate concern to the user, selecting a target will enable the appropriate performance estimators which are described below.


File
Edit
Unparse
Transform
Dependence
Terget
Interchange
block by k
vectorize
distribute
unroll by k
scalar expand
Forall
split node
doacross
nscript
Sour
Dep. Graph
For I
For j
Asign

Select locus: For

```
Procedure Matmul(A,x) returns (x);
    param A: array[1..n,1..n] of real;
            x: array[1..n ] of real;
    const n = upper(x);

    x[1..n ] = 0.0;
    For i in 1..n loop
        For j in 1..n loop
            x[i] := x[i] + a[i,j]*y[j];
        endfor;
    endfor;

 end;
```

Figure 2.1. BLED Screen After Program Load.
The Window on the Right is the Data Dependence Graph.

Optimizing Code For The BBN Butterfly

To begin working with the system the user selects a segment of code, which we call a 'focus', on which to apply the tools BLED provides. For example, suppose the target is the BBN Butterfly and the user picks the innermost loop (by a mouse selection) as his focus. Among the menu headings he has at the top of the screen, is one called "Transforms". This menu contains a list of program restructuring transformations that can be used to expose concurrency or make parallelism explicit. For example, the user may have decided that he wishes the innermost "for" loop to be run in parallel. By selecting the transformation "forall" the system will first verify that the transformation can be legally applied. This requires a search of the data dependence graph to make sure that the appropriate conditions are satisfied so that the transformation can be legally applied. (A more detailed discussion of this process is given in Section 3.) In this case the transformation is legal and the code now takes the form

```
for j in 1..n loop
     forall i in 1..n do
          y[i] := y[i]+A[i,j]*x[j];
     end;
end;
```

Following this operation, the user could invoke the code generator, but a better use of the system is to first invoke the analysis tool. This involves the selection of a menu item "analysis: Parallel Loop". Because the target machine, the Butterfly, executes such loops at the cost of a function call and an atomic increment to the index, the reply will be in the form

```
Loop Overhead to Body ratio > 50%.
Suggestion: increase granularity by blocking,
     merging or loop interchange.
```

Of the three suggestions, loop interchange is the easiest. After this operation the loop takes the form

```
forall i in 1..n do
     for j in 1..n loop
          y[i] := y[i]+A[i,j]*x[j];
     end;
end;
```

The "forall" loop body will now be large with respect to the loop overhead, but another problem that may inhibit performance is memory contention for shared data. In this case a second analysis tool called "analysis: Cache Management" can be invoked. As will be described in more detail in section 4, this tool will report the following information when the program focus is the "forall" loop.

```
Cache/Local memory analysis for iterate i:
     Suggest local copies of: A[i,1..n], x[1..n]
     for n=100 hit ratios will be 0.99, 0.9999
```

This (too cryptic) message suggests that local copies of these variables be made in each processor. A major shortcoming of this form of the analysis is that the user is not informed of the penalty for failing to take this advise. In section 4 we discuss the future extensions of this analysis that will provide expected improvements in multiprocessing efficiency as a result of this memory management. The resulting program will take the form

```
forall i in 1..n do
var: x_local, A_local: array[1..n] of real;
     A_local[1..n] := a[i,1..n];
     x_local[1..n] := x[1..n];
     for j in 1..n loop
          y[i] := y[i]+A_local[j]*x_local[j];
     end;
end;
```

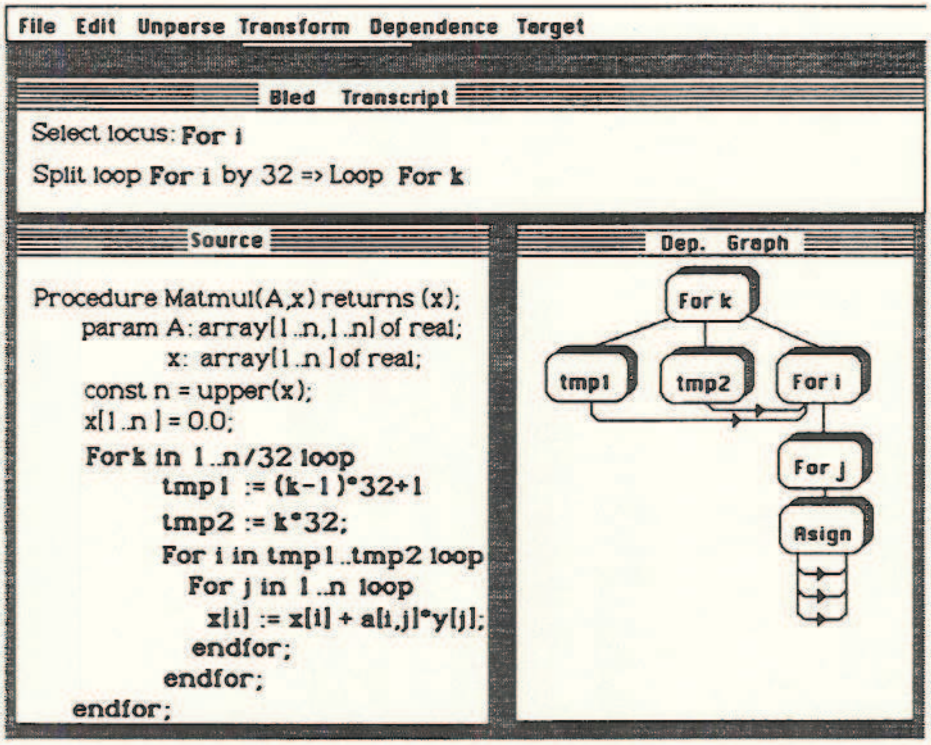
Clearly *x_local* need only be initialized once per processor but in Blaze we have no way to express this, so this task is left to the code generation step.

Optimizing Code For The Alliant FX/8.

Suppose the target machine were chosen to be the Alliant FX/8. Each processor on this machine has a vector instruction set and a vector register set that generalizes and extends the M68020 which forms the basis of the rest of the processor design. The execution of parallel loops requires almost no overhead, but best performance is achieved only if the shared cache memory and the vector instruction set is optimally utilized.

To optimize the matrix-vector routine for the FX/8 we would begin by recognizing that the vector instructions operate from registers of length 32. For this reason, the user may wish to "block" the inner loop in vector segments of length 32. This requires two steps. First the user selects "block by 32" from the transformation menu as illustrated by the screen dumps in Figures 2.1 and 2.2. In the second step the inner loops are vectorized.

The result of these two operations is shown below.



```
Select locus: For i
Split loop For i by 32 => Loop For k
```

```
Procedure Matmul(A,x) returns (x);
    param A: array[1..n,1..n] of real;
           x: array[1..n ] of real;
    const n = upper(x);
    x[1..n ] = 0.0;
    Fork in 1..n/32 loop
           tmp1  := (k-1)*32+1
           tmp2 := k*32;
           For i in tmp1..tmp2 loop
             For j in 1..n  loop
               x[i] := x[i] + a[i,j]*y[j];
             endfor;
           endfor;
   endfor;
```

Figure 2.2. System State After Loop Blocking Transformation.

```
for j in 1..n loop
     for k in 0..n/32-1 loop
          k1 := 32*k+1;
          k2 := 32*(k+1);
          y[k1:k2] := y[k1:k2] + A[k1:k2,j]*x[j];
     end;
end;
```

At this point, a third component of the performance analyzer can be invoked. The **Vector Code Analyzer** will make an estimate of the quality of the vector instruction utilization for this loop. In this case it will report

```
Vector instructions  Frequency
Load/Store           2*n*n/32
Multply-Add          1*n*n/32
Arith. Efficiency = 33% (for n=100)
```

The vector efficiency term is computed based on the fact that for diadic vector instructions the processor is capable of two floating point operations every clock cycle. Vector loads and stores do not contribute to the total number of arithmetic operations but they do consume large amounts of time. If we look at the inner loop we notice that the indecies on the vector segments for y do not depend upon j. In this case one simple "loop interchange" transformation will bring the j loop inside and the processor need only load and store each y segment once. The rest of the time it can remain in a vector register. Parallelizing the outer loop gives the code below and the corresponding statistics.

```
forall k in 0..n/32-1 do
     k1 := 32*k+1;
     k2 := 32*(k+1);
     for j in 1..n loop
          y[k1:k2] := y[k1:k2] + A[k1:k2,j]*x[j];
     end;
end;
```

The output from the code analyzer is now

```
Vector instructions  Frequency
Load/Store           2*n/32
Multiply-Add         1*n*n/32
Arith. Efficiency = 99% (for n=100)
```

Notice that this simple transformation has had the effect of a three fold improvement in vector unit efficiency. The resulting code is, in fact, the fastest matrix-vector form for this machine.

3 SYSTEM ORGANIZATION AND THE TRANSFORMATION MODULES.

The system is organized as a set of four interacting modules: the parser/analyzer, the transformation module, the performance estimator, the code generator, and the user interface manager. In this section we describe the Parser/analyzer, the user interface and the program transformation modules.

The Program Parser and Analyzer.

At the time a program module is loaded into the system it is completely parsed and a full control dependence graph and symbol table is constructed. The control dependence graph is based on a statement level version of the PDG of Feranti and Ottenstein [6]. Each program statement is represented by a graph node in the control dependence graph. An arc goes from one node to another if the latter is "control dependent" (in the sense of [6]) on the former. In the case of structured programs this graph is always a tree as illustrated in Fig 3.1.

The nodes of the control dependence graph, called BIF nodes, represent program statements, declarations and control forms. Of course, this is not enough detail to describe a program. Within each program statement there are one or more expressions describing the parameters of the statement. For example, a "for loop" in Blaze or FORTRAN takes the form

For <var> from <lower_bound> to <upper_bound> by <increment>.

```
For i in 1..n loop
   x[i] :=1.5;
   if( i > 3) then
       x[i] := 2;
   else
       y[i] := 3;
   endif;
endloop;
```

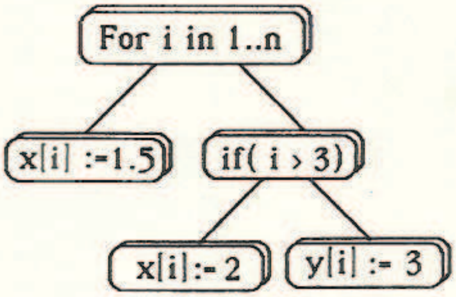


Figure 3.1. Program Control Dependence Graph

Each such expression is represented by a low-level dataflow graph which is attached to the BIF node which forms a template for the statement. While there is nothing new about this type of representation, we have found that its simplicity provides for great generality in the language it supports. Hence, the same graph can be used to represent several different programming languages.

The basic program graph has been defined with a general imperative programming language in mind. Using the same internal structure we currently represent either Blaze or FORTRAN programs. The extensions to support C are not complete, but only require the addition of a pointer type and the corresponding dereferencing operators. The way this works is that common control constructs, i.e. "for" loops, if-then-else blocks, case statements, variable assignments and function calls, form the core of the common semantics of each of these languages. There is one node type for each of these fundamental forms. For those features that exist in one language but not the others, we add extra node types. For example, FORTRAN has a "goto" statement and Blaze does not; also Blaze and C have record type variables and FORTRAN does not. The most important point is that, at the level of program semantics, and, especially in the area of control structures, there is very little difference between these languages. This means that we can write general program transformation modules that work at the level of common semantics.

We provide a special parser and "unparser" for each language. The unparser is the devise for recovering the original source for display in the text window of the system. This works by simply traversing the control dependence graph and, using the symbol table generated by the parse, reproducing the original source up to, but not including the programmer's use of white space. (We do save comments and try our best to put them back in the original positions. This can be difficult if the "original position" no longer exists after