Automatic Parallelization Techniques for the EM-4

Lubomir Bic and Mayez Al-Mouhamed

Department of Information and Computer Science
University of California
Irvine, CA 92717, USA.

Abstract:

This paper presents a Data-Distributed Execution approach that exploits interation-level parallelism in loops operating over arrays. It performs data-dependency analysis, based on which arrays are distributed over the different local memories. The code is then transformed to “follow” the data distribution by spawning each loop on all PEs concurrently but modifying its boundary conditions so that each operates mostly on the local sub-ranges of the data, thus reducing remote access to a minimum. The approach has been tested on the EM-4 supercomputer by implementing several benchmark programs. The experiments show that high speedup is achieved by automatic parallelization of conventional Fortran-Like programs.

Keywords: Data-parallel, loop restructuring, parallel execution, dataflow.

Automatic Parallelization Techniques for the EM-4

Lubomir Bic and Mayez Al-Mouhamed
Department of Information and Computer Science
University of California, Irvine, CA 92717

Abstract

This paper presents a *Data-Distributed Execution* (DDE) approach that exploits iteration-level parallelism in loops operating over arrays. It performs data-dependency analysis, based on which arrays are distributed over the different local memories. The code is then transformed to "follow" the data distribution by spawning each loop on all PEs concurrently but modifying its boundary conditions so that each operates mostly on the local subranges of the data, thus reducing remote accesses to a minimum. The approach has been tested on the EM-4 supercomputer by implementing several benchmark programs. The experiments show that high speedup is achievable by automatic parallelization of conventional Fortran-like programs.

1 Introduction

Distributed memory MIMD computers are among the most difficult to program, since independent processes or threads operating on their own memories and communication with other processes through message or remote memory access must be efficiently managed.

The objective of this paper is to demonstrate that the EM-4 multiprocessor [1], together with an automatic parallelization technique referred to as DDE (Data-Distributed Execution), which has originally been developed in the context of coarse-grain dataflow [2], offer an efficient computing environment in which large portions of scientific code can be parallelized using implicit parallelism.

This paper is organized as follows. The EM-4 architecture and its most important characteristics are described in Section 2. Section 3 presents the principle of DDE and the actual transformations applied to programs to extract parallelism. Section 4 presents the results of the benchmarks executed on the EM-4 and Section 5 concludes about this work.

2 The EM-4

The EM-4 distributed memory MIMD supercomputer [1, 3] has 80 PEs that are interconnected using a *direct connect topology* over an Omega network. Its important features are the *fast inter-processor communication* is and the support for *multithreading*.

To allow efficient multithreading, it is necessary to create threads and quickly switch among them by using the matching of operands required by dataflow [4]. For this, a 4-stage nested pipeline is used so that the outer 4-stage is is used for the dataflow mode and the inner 2-stage is used in sequential mode.

The first two stages perform the *direct matching* [3]. Each operand segment has an entry pointing to a dyadic instruction in the code segment. A pointer from the operand to the code segment is also created because distinct operand segments could be simultaneously pointing at the same code segment.

Stage 1 fetches the code segment pointed by each new packet. In stage 2, if the location addressed by the packet is empty, the packet is stored in that location and no further action is taken. If that location already contains an operand, it is marked empty and both operands are passed to the third stage, that performs the fetch and decoding of the instruction. Finally, the execution is performed by the fourth stage. The above cycle is repeated until an instruction indicates that sequential execution is to commence. At that time, no new packets are accepted by stage 1. Instead, stage 3 continues fetching subsequent instructions and passing them to stage 4 for execution in a normal von Neumann style. This mode continues until it is explicitly terminated by an instruction. Hence the EM-4 is capable of switching between data-driven and control-driven execution very efficiently.

The direct matching may be viewed as a mechanism for thread management [6] because it provides efficient means to: 1) execute sequences of control-driven instructions (threads) until termination or remote memory request, and 2) quickly switch to a new thread by using direct matching. Suspended threads are then resumed when the remote data becomes available. This approach is very useful to hide memory latency [5].

The EM-4 can be programmed using three distinct approaches: 1) a functional program is compiled into a dataflow graph of *strongly connected blocks*, 2) Using a library of threads, the user specifies their mapping and the distribution of data structures, and 3) using the proposed implicit parallelism with conventional languages. This last approach will be presented in the next section.

3 Data-Distributed Execution

The basic philosophy of DDE is to distribute the arrays over the PEs to minimize the amount of remote data transfer required during the execution of concurrent threads. We consider programs written in a conventional language, such as Fortran or Fortran-like c constructs.

At run time, each parallel loop is associated with two families of threads: 1) global threads (GT) are created by sub-dividing the range of the parallel iterator, and 2) partitioning each GT into local threads (LT). The GTs promote inherent parallelism and the LTs provide the PEs the opportunity to hide remote memory access (RMA) by performing context switching to ready LTs.

3.1 Analysis and Restructuring

Dependence analysis [9] is used to identify *loop-carried-dependencies* (LCD) that inhibit parallelization of the loop and lead to generation of a single scalar thread. Global threads will be created for loops having only *loop-independent-dependencies* (LID).

Reduction of the granule size of LCD loops is done by removing parallelizable code fragments using known techniques such that *loop distribution* and *partial parallelization*. These fragments are inserted closer to their data producer or consumer that belong to LID loops with the same loop headers. This causes immediate references to become subject to identical loop constraints. Next, the domain of each array that is indexed by the parallel iterator index is implicitly distributed across the PEs to yield the least number of remote memory accesses. Finally, a voting technique allows finding the most frequently used array distribution that becomes the global distribution.

Renaming [7] multiple write to the same variable is performed in order to make the code obey the *single assignment* principle, i.e. allows a value to be written only once. This eliminates possible race conditions and produces the correct result regardless of loop scheduling.

3.2 Transformations for Parallelism

We will use the generic program example in Figure 1 to illustrate the creation of global and local threads. The first step is to replace all array definitions (line 1) by a call to an *allocate() function*, which, at run time, performs a distributed allocation of the array

```
Sequential Code:

1   int A[][], B[][];
2   for (i = 0; i < n1; i++)
3     for (j = 0; j < n2; j++)
4       a[i][j]=some_comp(B[i][j],..);

Transformed Code:

5   A = allocate(ROW,...);
6   B = allocate(ROW,...);
7   for (p = 0; p < NO_PEs; p++)
8     fork(pe[p], i_loop, ...);
9     void i_loop(...)  {
10    lb = max(0, get_my_start_i(A));
11    ub = min(n1, get_my_end_i(A));
12    for (i = lb; i < ub; i++) {
13      for (j = 0; j < n2; j++) {
14        fork(self_pe, j_loop,...);
15  }
16  void j_loop(...)  {
17    value=some_comp(read_array(B,i,j),..);
18    write_array(A,i,j,value);
19  }
```

Figure 1: Program transformation

by sending requests to all PEs to allocate their own local subranges (lines 5-6). The type of distribution is determined, for each array, based on the preceding program analysis. Given an array, A, consider each access $A[i, j]$ within a loop. If this is a singly nested i loop, or a nested loop with i as inner index, then mark the access as a column access. Next, it counts the number of loops with row versus column accesses and choose the distribution having the highest frequency.

In Figure 1, the parameter *ROW* indicates that the arrays are to be distributed row-major, that is, each PE will be responsible for a certain subrange of the index i.

To implement DDE, each loop is started on all PEs concurrently so that each PE operates on a different subrange of the original loop. For nested loops it is first necessary to determine the loop nest that controls the array distribution. In most cases, all arrays accessed within a given loop will have been distributed along the same dimension. In this case, the index along which the arrays were distributed determines the loop level to be distributed. In the rare cases where not all arrays accessed within a given loop have been distributed along the same dimension, we count the number of array accesses along each dimension and select the most frequently used one to determine the loop level to be distributed.

In Figure 1, both arrays were distributed row-major and hence the *i_loop* (line 2) was chosen for distribution. It has been transformed into the function called *i_loop*() (line 9) and is spawned on all PEs using the loop shown on lines 7–8. This loop executes on PE_0 that is the master PE.

To make each PE operate on a different subrange, the code to compute the local lower and upper bounds (*lb*, *ub*) for the distributed loop is inserted (lines 10-11). This code, referred to as the *Range Filter*, accesses the header of the array the loop operates on and, from the recorded distribution information, computes the local subrange. The functions *get_my_start_i*() and *get_my_end_i*() represent the retrieval of the starting and ending *i_indices*, which are different for each PE. These are then combined using the *max* and *min* functions with the boundaries of the original loop, in this case, the values 0 and *n*1, respectively.

It is necessary to increase the level of parallelism within each PE by locally spawning the iterations of the next outer nest as separate LTs. This is analogous to the previous transformation except that the PE specified by the *fork*() primitive is the local PE. The *j_loop* becomes a separate function (lines 16-19) and is spawned for each *j* on the PE as a local thread (line 14).

The final transformation is to replace each reference to an array element by a call to the *read_array*() or *write_array*() function, which determines the location of the given element (local or remote) and performs the access. Given an array $A[n_1, n_2, \ldots]$, assume that the array is to be distributed along a given dimension n_d. We interpret n_d as a binary number and use the leading k bits as the PE number and the remaining bits as the local index. The number k is determined by right-shifting n_d until the result is smaller than the total number of PEs. The number k is then stored in the array header and used by the access functions.

Programs are optimized by inlining the inserted functions, notably the *read_array* and *write_array*, and moving of invariant code outside of the loops. The schedule, resulting from the insertion of the various fork and barrier primitives, is also improved by moving loops that do not need to wait for a particular barrier in front of that barrier. Hence a form of a greedy schedule is implemented.

4 Results

This section presents the results of applying the proposed DDE approach to: 1) the conduction loop of the SIMPLE benchmark, and 2) the matrix multiply, and running the code on the EM-4.

4.1 SIMPLE

SIMPLE is a well-known benchmark program [8] that simulates the behavior of a fluid in a sphere, using the Lagrangian Formulation.

In this experiment, we have considered the *conduction function* which is the main and most difficult portion to parallelize. The code consists of a number of singly and multiply nested loops iterating over several 2-D arrays.

The resulting parallelism profile is shown in Figure 2. The measured speedup was 65 and the average idle time was 9.09%. The extracted parallelism was nearly 77 during most of the computation time. This parallelism profile is excellent because PE_0 and two other PEs – those holding the boundary rows – were idle during most of the computation time.

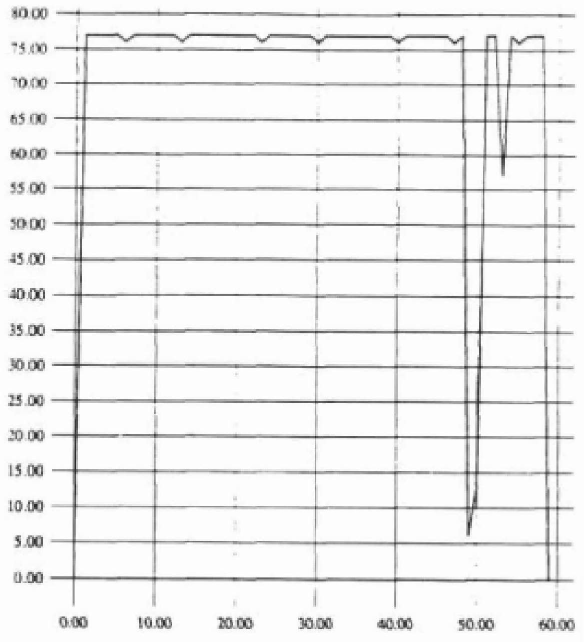

Figure 2

The drops in parallelism, resulting from barriers that could not be masked by other work, were steep, narrow, and few in number. The shape of the drop is a clear indication of the EM-4's superior communication network.

The small number of the drops and the fact that they do not extend all the way down to a single PE is an indication of the available parallelism in a typical scientific application. There were sufficient numbers of independent loops that could be run concurrently and thus mask the effect of much of the idle time resulting form barriers.

4.2 Matrix Multiply

Using the DDE apprach, we have parallelized the simple triply-nested loop of the matrix multiply. Three experiments were carried out for matrix sizes (multiples of 79) of 79^2, 158^2, and 316^2, respectively. The parallelism profile for the second experiment (Figure 3) is representative of those of the other experiments. The obtained speedups were 7.5, 8.6, and 9.1 for each of the above matrix sizes, respectively.

The resulting speedup is quite modest, even when the matrix size is large. However, the speedup obtained by a manual parallelization was also low and hence the result indicates that the automatic parallelization approach performs very well.

The parallelism profile of the matrix multiply algorithm (Figure 3) shows a pronounced trailing edge that causes some load imbalance. This is surprising because of the regularity of the problem, the distribution, and the architecture. The trailing edge takes up on the order of 20% of the total computation and accounts of most of the idle time (11.7%) measured for this problem. This idle time could be eliminated if memory synchronization were available on the EM-4. Other computation could then partially overlap with the matrix multiply loop.

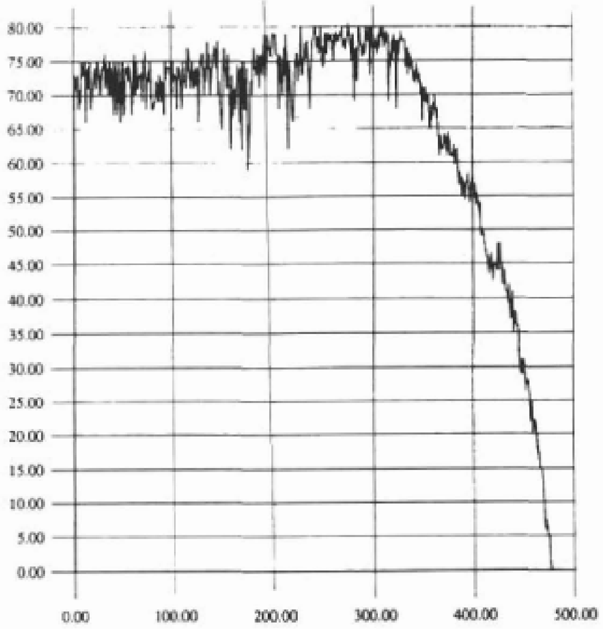

Figure 3

5 Conclusions

There are many real world applications that a hybrid machine like the EM-4 could exploit without requiring the labor-intensive and error-prone task of manual parallelization. Significant speedup can be achieved using Fortran-like programs that iterate over large data structures.

Automatic parallelization will, of course, not eliminate the need for the human involvement as was the case with the matrix multiply, where automatic parallelization of a given algorithm yielded only marginal speedup. Hence, the proposed approach is only one component of a parallel programming environment, which must take into consideration the user, the language, the compiler, the architecture, and the various development tools.

References

[1] Sakai, S., Yamaguchi, Y., Hiraki, K., Kodama, Y., Yuba, T. *'An Architecture of a Dataflow Chip Processor', Proc. 16th Annual Int'l Symp. on Computer Arch., Jerusalem, Jun. 1989*

[2] Bic, L., Roy, J.M.A., Nagel, M. *'Exploiting Iteration-Level Parallelism in Dataflow Programs', 12th Int'l Conf. on Distributed Computing Systems, Yokohama, Japan, Jun. 1992*

[3] Sakai, S., Hiraki, K., Yamaguchi, Y., Kodama, Y., Yuba, T. *'Pipeline Optimization of a Dataflow Machine', Advanced Topics in Data-Flow Computing, Prentice-Hall, Ed. J-L. Gaudiot and L. Bic, 1991*

[4] Arvind, Bic, L., Ungere, T. *'Evolution of Data-Flow Computers', Advanced Topics in Data-Flow Computing, Prentice-Hall, Ed. J-L. Gaudiot and L. Bic, 1991*

[5] Arvind, Iannucci, R.A. *'Two Fundamental Issues in Multiprocessing', Proc. DFLVR Conf. on Parallel Processing in Science and Engineering, Bonn Bad Godesberg, Germany, Jun. 1987*

[6] Sato, M., Kodama, Y., Sakai, S., Yamaguchi, Y., and Koumura, Y. *'Thread-Based Programming for the EM-4 Hybrid Dataflow Machine', Proc. 19th Annual Int'l Symp. on Computer Arch., Gold Coast, Australia, May 1992*

[7] Cytron, R., Ferrante, J. *'What's in a Name? -or- The Value of Renaming for Parallelism Detection and Storage Allocation', Proc. Int'l Conf. on Parallel Processing, Aug 1987, pp. 19-27*

[8] McMahon, F.H. *'The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range', UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, Dec. 1986*

[9] Padua, Wolfe, M. *'Advanced Compiler Organization', Comm. ACM, Dec. 1989, pp. 1184-1201*