The EM-4 Under Implicit Parallelism

Lubomir Bic and Mayez Al-Mouhamed

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

Abstract

The EM-4 is a supercomputer that offers very fast interprocessor communication and support for multithreading. In this paper we demonstrate that the EM-4, together with an automatic parallelization technique referred to as *Data-Distributed Execution* (DDE), offer a computing environment in which large portions of scientific code can be executed without the need for any explicit parallelism.

DDE 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 by implementing several benchmark programs representative of common scientific applications. 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 communicating with other processes through message or remote memory access must be managed efficiently. While a number of such machines have been built to date, their software development lags far behind due to the lack of understanding of and the consequent lack of programming support for such machines. The most common approach is to extend existing languages with various primitives for process control, synchronization, and communication, and to leave it up to the programmer to develop parallel algorithms using explicit parallelism.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
ICS-7/93 Tokyo, Japan
© 1993 ACM 0-89791-600-X/93/0007/0021...$1.50

While these approaches may be justified when developing algorithms or programs that are difficult to parallelize, they are unnecessary in cases where the problems are highly regular and parallelism is abundant. For such programs, automatic parallelization is just as effective, provided the underlying architecture offers adequate facilities to support the derived parallel code.

The objective of this paper is to demonstrate that the EM-4 multiprocessor [1, 2], 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 [3, 4], offer a computing environment in which large portions of scientific code can be executed without the need for any explicit parallelism to be specified by the programmer.

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 is a distributed memory MIMD supercomputer [1, 2, 5] with 80 PEs but may be expanded to over 1000 processors. Its most important features towards promoting implicit parallelism are the *fast interprocessor communication* and the support for *multithreading.* Let us address these in turn.

The 80 PEs of the EM-4 are interconnected through an Omega network by using a *direct connect topology.* This topology configures the 80 PEs into 16 groups of 5 PEs each. Within each group, a message must traverse at most 4 links to go from any PE to any other. Across clusters, the average number of hops [2] is proportional to $\log(npe)$, where npe is the number of processors. This approach provides for fast communication in addition to

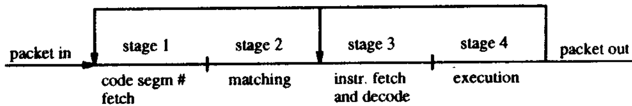

Figure 1: EM-4 Pipeline

dynamic load distribution at each node.

To allow efficient multithreading, it is necessary to create threads and quickly switch among them. Originally, the EM-4 was conceived as a dataflow/von Neumann hybrid; it is capable of executing sequential code as well as performing the matching of operands required by dataflow [6]. This is achieved by using a 4-stage pipeline so that the first two stages can be bypassed in case of sequential code. A simplified block-diagram of the pipeline is shown on Figure 1. This, in fact, results in a nested pipeline, where the outer four stages are used for the dataflow mode and the inner two stages are used in sequential mode.

The first two stages perform what has been termed *direct matching* [5]. Whenever a function is invoked, an operand segment is created such that there is one entry in the operand segment for each dyadic instruction in the code segment. A pointer from the operand to the code segment is also created, since potentially distinct operand segments (one for each invocation) could be simultaneously pointing at the same code segment.

Whenever a data packet arrives at stage 1 of the pipeline, the code segment number is fetched from the corresponding operand segment. The location addressed by the packet is examined in stage 2. If it is empty, the packet is stored in that location and no further action is taken by the subsequent stages. If, on the other hand, it 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 corresponding instruction. Finally, the execution is performed by the fourth stage.

The above cycle is repeated until an instruction, through a special bit, 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 using registers. 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 above direct matching mechanism may be viewed as a mechanism for thread management [9] because it provides efficient means to: 1) execute sequences of control-driven instructions (threads) until termination or a remote memory request is encountered , 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 and has been identified as one of the major requirement in multiprocessing [7].

Barrier synchronization has been provided through a library and, due to the fast interconnection network of the EM-4, performs very efficiently. For example, a reduce/add over 80 PEs takes $15\mu sec$.

I-structures [8] and other memory synchronizing data structures are supported only through software and, consequently, are not very efficient.

The EM-4 can be programmed using three distinct approaches. The original design was intended for the execution of functional languages. A functional program is compiled into dataflow graphs, where components, referred to as *strongly connected blocks* do not require any external inputs other than the operands of the first instruction, i.e., operands of other instructions are generated within the block. This enables the block to execute in an uninterrupted control-driven manner. The remaining instructions are executed in a data-driven style using operand matching.

The second type of programming [9] consists of using a thread library. In this case, the programmer: 1) designs threads and specifies their mapping, and 2) explicitly handles the distribution of data structures.

The third style of programming, which is discussed in this paper, attempts to exploit implicit parallelism using conventional languages. This approach will be presented in the next section.

3 Data-Distributed Execution

The approach described in this paper focuses on *data parallelism*, specifically, parallelism at the iteration level of loops operating on arrays. We consider programs written in a conventional language, such as Fortran or Fortran-like c. The main reason is that DDE, in its present form, provides transformations for only loops iterating over arrays. Fortunately, there is a vast body of scientific programs that fit that description.

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. 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) uniformly partitioning each GT into local threads (LT). While GTs promote inherent parallelism, the LTs provide each PE the opportunity to hide remote memory access (RMA) by performing context switching to ready LTs. The efficiency of this approach depends on the distribution of arrays so that a given GT is mapped to the PE whose local memory

contains most of the array references that are used by that GT. While a perfect data and code alignment is not always possible due to imperfections in the analysis and other factors, the number of RMAs is kept to a minimum.

3.1 Analysis and Restructuring

Dependence analysis [14, 15] is used to determine the chronology of operations such that data dependencies are preserved. *Loop-carried-dependencies* (LCD) 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) that are free of LCDs. To reduce the granule size of LCD loops, we use traditional methods to remove parallelizable code fragments from LCD loops using *loop distribution* and *partial parallelization*. Next, we attempt inserting these fragments closer to their data producer or consumer expressions that belong to LID loops with the same loop headers. This identifies all the LID loops that can be used for generating the global threads.

In some cases, load inbalance may occur among the PEs because the parallel iterator loop count is small compared to *npe*. To load balance the PEs, a number of parallel outer loops are combined such that the number of their instances, that is the number of GTs, becomes the closest to *npe*.

Renaming [10] multiple write operations to the same variable is performed in order to make the code obey the *single assignment* principle, which allows a value to be written only once. This eliminates possible race conditions [11] and produces the correct result regardless of loop scheduling.

The above loop restructuring places data producers closer to their data consumers, thus, causing more immediate references to become subject to identical loop constraints. Because each instance of a parallel loop is used to generate a GT that is mapped to one PE, 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. This enables finding different array distributions over the PEs depending on the way each array is referenced in different loops. A voting technique allows finding the most frequently used array distribution that becomes the global distribution.

3.2 Transformations for Parallelism

We will use the generic program example in Figure 2 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 by sending requests to all PEs to allocate their own local subranges

```
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 2: Program transformation

(lines 5–6). The type of distribution is determined, for each array, based on the preceding program analysis. Without loss of generality, we consider 2-D arrays and their column-major and row-major distributions but the approach can easily be extended to arbitrarily dimensioned arrays. The heuristic operates as follows:

- 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.
- If it is a singly nested j loop, or a nested loop with j as inner index, then mark the access as a row access.
- Count the number of loops with row access versus column access. Depending on which is *more frequent*, choose row or column distribution for the matrix.

In Figure 2, 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, i.e., to make the code *follow the data distribution*, the following basic approach is used. Each loop is started on all PEs concurrently. The loop code, however, is augmented 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. This is based on the distribution of the arrays operated on by the loop. 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.

Once a level is chosen, the corresponding for-loop construct is transformed into a function and the necessary code is inserted to fork the GTs over all PEs. In Figure 2, 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. It uses the fork primitive of the EM-4 thread library, which specifies the target PE, the function to be called, and any arguments to be passed to that function.

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 $n1$, respectively.

The transformations performed so far resulted in the creation of a GT on each PE. To mask memory latency resulting from remote memory accesses, it is necessary to increase the level of parallelism within each PE. This is achieved by locally spawning the iterations of the next lower nest of the distributed loop as separate LTs. This is analogous to the previous transformation except that the PE specified by the *fork*() primitive is the local PE. In Figure 2, 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). For more deeply nested loops this transformation can be repeated at yet lower levels until a sufficient level of parallelism is achieved.

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. The implementation of these functions depends on the specific array mapping algorithm. For

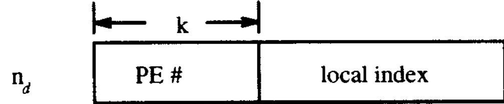

Figure 3: Array Mapping Using Dimension n_d

the programs presented in this paper, we have implemented the following mapping strategy. 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, as illustrated in Figure 3. 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 as follows. Assume we are given an element $A[i_1, i_2, \ldots]$ to be accessed. The read function performs the following computations:

```
pe = nd >> k;
loc_indx = mask[k] & k;
global_read(pe, address(A, loc_indx, j));
```

The first instruction right-shifts n_d by k bits to obtain the PE number. The second instruction masks out the leading k bits (using an array of predefined masks) to get the local index. The third operation then performs the actual memory read by computing the local address of the element and retrieving the value from the given PE.

Since the address calculation is essentially the same as for accessing an array element in a sequential system, the additional overhead are the two binary operations and the initiation of the global access. This modest overhead is well worth the performance gain achieved through parallelism.

Before the program is submitted for execution, a number of optimizations are performed. The most significant ones are the inlining of the inserted functions, notably the *read_array* and *write_array* functions, and moving of all invariant code outside of the loops. The program 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 the *Livermore loop 3* and the *conduction loop* of the SIMPLE benchmark, These programs have been run on the EM-4.

4.1 Livermore Loop 3

The Livermore Loop 3 [13] consists of a reduction/multiply over two vectors z and x with size n. The loop is repeated a constant number of times in order to accurately measure the execution time on the EM-4.

Analysis reveals no recurrence and hence both arrays x and z are distributed by allocating a subrange of $\lfloor n/npe \rfloor$ elements to each PE, where $npe = 64$ in this experiment. The accumulation of partial results was performed using a barrier add operation.

To measure the speedup, the execution time of the sequential version was obtained by compiling the program using a commercial c compiler and running it on one node of the EM-4. By varying the size of the vectors from 1000 to 10,000 and 20,000, the resulting speedups were 10.5, 42.5, and 51, respectively. The average idle time for each PE was 56%, 21%, and 13%, for each of the above vector sizes.

This demostrates that the EM-4 is capable of performing reduction quite efficiently. Due to its fast communication network, the grain size may be fairly small. Even for a vector size of 1000 elements where each GT consists of only 15 multiplications, a 10-fold speedup is achieved by using 64 PEs. Increasing the granule size of the GT significantly improves performance, such as the 51-fold speedup for 20K vectors.

4.2 SIMPLE

SIMPLE is a well-known benchmark program [12] that performs a hydrodynamics and heat conduction simulation and is indicative of large-scale scientific code that is executed on today's supercomputers. It 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 portion of SIMPLE and the most difficult to parallelize compared to its other routines. The code consists of a number of singly and multiply nested loops iterating over several 2-D arrays.

After a simple transliteration from Fortran into c, the code was analyzed for recurrences. Since there were no simple intra-loop dependencies to be removed, only barriers were inserted as necessary. The code was then translated automatically according to the steps of Section 3.2. After optimization, the code was executed such that PE_0 performs the allocate() and work distribution functions and the remaining 79 PEs executed the actual computation.

The resulting parallelism profile is shown in Figure 4. 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, given that PE_0 and two other PEs – those holding the boundary rows – were idle during

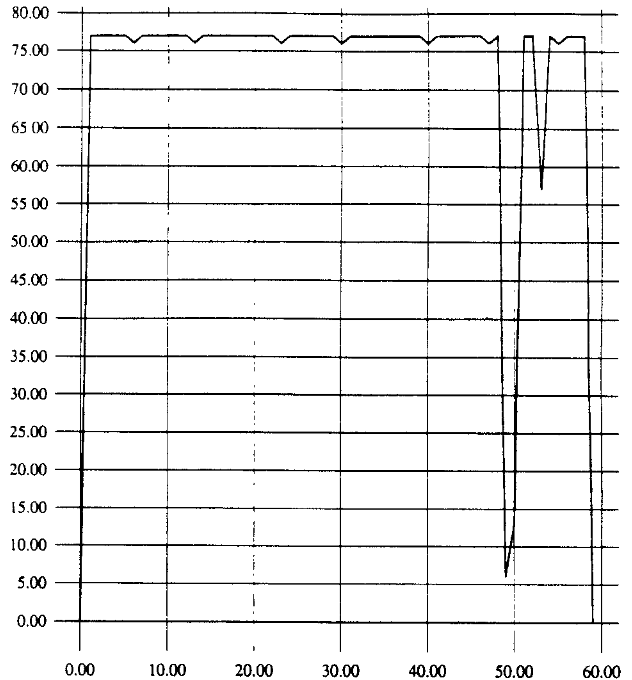

Figure 4: Parallelism Profile of Condunction Loop

most of the computation time.

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 drops is a clear indication of the EM-4's superior communication network. It allows workload to be distributed throughout the machine and results to be rapidly collected in a central place, thus keeping the idle time due to barriers to a minimum.

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. Even though we have used only a rudimentary scheduling technique (move loops in front of barriers if possible), 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.

5 Conclusions

In this paper we have investigated an approach to parallel programming using transformations applied at the source level of sequential programs. The approach was investigated for the EM-4 multiprocessor in order to evaluate its performance under implicit parallelism. The primary conclusion we draw from our experiments is that 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. It has been shown that a significant speedup is achievable on regular Fortran-like programs that iterate over large data structures, such as the SIMPLE benchmark. Primarily, this is attributed to the EM-4's fast communication network and its support for multithreading. The main shortcoming in the EM-4 design is the lack of support for memory synchronization. The consequent use of barriers results in parallelism drops which cannot always be masked by other computation due to the difficulty in finding efficient schedules and/or the lack of inherent parallelism.

We recognize that automatic parallelization will not eliminate the need for the human programmer's involvement. Other well-known algorithm parallelized using DDE yielded only marginal speedup. To solve these problems more efficiently, different algorithms must be developed, which cannot be done without human intelligence. Hence we view automatic parallelization as only one component of a parallel programming environment, which must take into consideration many components, including the programmer, the language, the compiler, the machine architecture, and the various development tools. The programmer's involvement is essential for algorithm development, as already suggested, and for parallelizing complex structures and intricate programs. The structure of most real world programs, however, is quite simple and very regular, and offers an abundance of parallelism. Thus it is a waste of the most precious resource when programmers are required to analyze such programs by hand and to explicitly insert parallelizing and synchronizing primitives when automatic parallelization techniques would perform equally well, as long as the architecture provides the necessary communication, multithreading, and synchronization support.

References

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

[2] Yamaguchi, Y., Sakai, S., Hiraki, K., Kodama, Y., Yuba, T. *'An Architectural Design of a Highly Parallel Dataflow Machine', Information Processing 89, Ed G. Ritter, Esevier Scientific Publishers, North-Holland, 1989*

[3] Bic, L. *'A Process-Oriented Model for Efficient Execution of Dataflow Programs', J. Parallel and Distributed Computing, Vol. 8, No. 1, pp. 42-51, Jan 1990*

[4] 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*

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

[6] 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*

[7] 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*

[8] Arvind, Thomas, R.E. *'I-Structures: An Efficient Data Type for Functional Languages', Computer Science Tech. Rep. TM-178, MIT, Cambridge, MA, Sep. 1980*

[9] 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*

[10] 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*

[11] Ackerman, W.B., *'Data Flow Languages', IEEE Computer, Feb. 1982, pp. 15-25*

[12] W. P. Crowley, C. P. Henderson, T. E. Rudy *'The SIMPLE Code', UCID 17715 Lawrence Livermore Laboratory February 1978*

[13] 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*

[14] Padua, D.A., Kuck, D.J., Lawrie, D.H. *'High-speed Multiprocessors and Compilation Techniques', IEEE Trans. Computers, Sep. 1980, pp. 763-776*

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