Automatic Data/Program Partitioning Using the Single Assignment Principle

Lubomir Bic
Mark D. Nagel
John M. A. Roy

January 1989

Technical Report #89-08

Abstract

Loosely-coupled MIMD architectures do not suffer from memory contention; hence large numbers of processors may be utilized. The main problem, however, is how to partition data and programs in order to exploit the available parallelism. In this paper we show that efficient schemes for automatic data/program partitioning and synchronization may be employed if single assignment is used. Using simulations of program loops common to scientific computations (the Livermore Loops), we demonstrate that only a small fraction of data accesses are remote and thus the degradation in network performance due to multiprocessing is minimal.

Keywords: array, cache, multiprocessor, distributed, single assignment

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

This work was supported by the NSF Grant CCR-8709817. Copyright © 1989.

Automatic Data/Program Partitioning Using the Single Assignment Principle

1. Introduction

In scientific computations, most potential parallelism may be found in highly structured data such as vectors and arrays. Current parallel architectures designed to exploit this parallelism include pipelined processors (with vectorizing compiler support), SIMD array processors, and MIMD architectures with either tightly or loosely-coupled processing elements. Pipelined parallel processors form a specialized class of architectures that are capable of achieving large speedups on structured numerical computations containing large amounts of vectorizable code. However, as is noted in [R&F87], the maximum speedup is limited and depends greatly on the proportion of vectorizable code present. SIMD multiprocessor architectures also are capable of extracting large amounts of parallelism from vector problems, and the amount of parallelism generally increases with increasing numbers of processors; however, the problem domain of most SIMD architectures is limited (i.e., SIMD is not suited to general computing).

In this paper we consider loosely-coupled MIMD architectures. Because MIMD architectures do not suffer from memory contention, they have the greatest potential for large-scale parallelism. The main problems with loosely-coupled MIMD include: (1) the need to find a good partitioning of the programs/data, (2) the need to introduce synchronization primitives to avoid race conditions, and (3) the need to introduce communication primitives for exchanging data among processors.

In this paper we show that a single assignment policy can produce a large degree of parallelism while keeping the amount of communication overhead low. In particular, for programs that follow the single assignment policy, we show the following:

- A simple automatic scheme for partitioning of data and programs can be employed.
- Synchronization can be fully automatic using a memory tagging mechanism

• Communication overhead for remote data accesses can be greatly reduced by using data caches. Furthermore, due to the single assignment policy, cache coherence problems are eliminated

The paper is organized as follows. In Section 2 we discuss the benefits of single assignment with respect to automated data/program partitioning. Section 3 shows how synchronization may be simplified by using the single assignment principle. Next, in Section 4, we present a simple scheme for data/program partitioning and discuss how caching reduces communication overhead. In Section 5, we discuss some of the disadvantages of single assignment programming and some approaches for overcoming these problems. In Sections 6 and 7, we discuss our simulation and the resulting access distribution classes. Finally, in Sections 8 and 9, we end with some conclusions and a discussion of our future research.

2. Single Assignment and Data/Program Partitioning

Partitioning a program over multiple processing elements involves both *data* and *program* partitioning. The programmer can do this partitioning (e.g., fork/join and cobegin/coend). This requires experienced parallel programmers and too much debugging time to be generally applicable. It is also possible for the compiler to detect parallelism and partition the program accordingly—most currently known methods are NP-complete, although some progress has been made in this area ([PEI86] and [PGW87]). The main problem with all these approaches is that data (variables) may be read and written from many parts of the program (i.e., by different instructions). It is difficult to decide where to place a variable with respect to the possible instructions that access that variable. Furthermore, synchronization primitives must be inserted to prevent race conditions, and communication primitives must be inserted to allow sharing of data across PEs. These increase the chances for program errors.

Most of these problems can be drastically simplified if single assignment principles are used. These principles require that no variable ever be assigned more than once throughout its scope. When extended to arrays, the definition is less clear. Some have defined single assignment on arrays as the requirement that the array be treated as a single object and thus can be assigned to only once [DEN75]. This is acceptable in languages that have a complete set of array operations, yet forming such a complete set can be very difficult. A better definition of single assignment for arrays is that each *element* of an array may be assigned only once. This allows a great deal more flexibility in the use of arrays, and this relaxation of single assignment rules does not cause any problems given proper hardware support.

With single assignment, only *one* instruction will ever write to a variable; it is the *producer* of the data. We can use this fact by requiring that each variable be mapped onto the same PE as its producer (i.e., the instruction that writes it). In other words, we do not partition programs explicitly, only data. The corresponding instructions can be mapped implicitly.

This is particularly attractive with arrays, which are typically accessed via loops. Each array is subdivided into segments and these are distributed across the PEs. The loop body accessing each cell is the same for each PE, and hence a copy is distributed to each PE. Each PE uses this copy to produce its array subrange. More precisely, the partitioning of a loop takes place as follows:

- Data partitioning is accomplished by segmenting each array into pages of some fixed (perhaps parameterized) size. A page p is allocated to the local memory of PE P if $p = P \bmod N$, where N is the total number of available PEs.
- Control partitioning will be done by assigning to each PE the responsibility for updating the elements in all the array pages it contains in its local memory.

As a simple example of this partitioning method, suppose we have a multiprocessor with four PEs and a page size of 32 elements. Given three arrays A, B, and C, (each of size 100) PE 0, PE 1, and PE 2 will each contain a single page of each array. PE 3 will contain a partial page (4 elements) of each array. For the following simple loop:

```
     DO 10 i = 1,100
10 A(i) = B(101-i) + C(i)
```

all four processors begin executing simultaneously—PE 0 fills A(1..32), PE 1 fills A(33..64), PE 2 fills A(65..96), and PE 3 fills A(97..100). Note that for most of the loop, each processor must access elements of array B that lie on a different processor than the executing processor. A method for eliminating nearly all of this communication overhead will be presented in Section 4.

Thus, data and program partitioning are achieved using simple rules which take advantage of single assignment. These rules are sufficient for most common forms of loops (see Section 6).

3. Synchronization Through Single Assignment Programming

Single assignment principles allow the implementation of a simple automatic synchronization mechanism. Each memory cell has two states—undefined or defined. If a cell is undefined, it may also have a queue of read requests associated with it. Hardware enforces the write-before-read requirement. Some examples of architectures that have this type of write-once/read-many memory access mechanism include HEP [S&F83] [SMI81] and I-structure memory in dataflow [ANP87] [A&C86].

Prior to execution, an array is either undefined or filled with initialization data (if specified in the program). Each PE may write only into undefined array cells and only into those mapped to that PE (i.e., each PE is the producer of *only* the array subranges mapped to it). This is achieved by screening the array indices so that the right hand side of the assignment is evaluated only for a given PE's subranges. Whether only the correct indices are generated, or if they all are generated and then screened is an implementation detail.

The point is *not* to perform the calculations in a PE not responsible for writing the associated element.

Race conditions are avoided by this single assignment policy. There will never be a race condition for writes to memory cell, since only one PE may write to any particular cell and writing more than once results in a runtime error.

Thus the single assignment rule automatically enforces synchronization in a distributed manner; no explicit synchronization mechanisms are necessary—a major issue in other programming paradigms.

4. Inter-processor Communication

We have seen that single assignment yields simple partitioning and synchronization schemes. Remote read accesses, however, are not eliminated, since any instruction may read any data item. If data is mapped onto the reading PE, the access is local, otherwise it is remote; the PE must request the value from the responsible PE by sending a message. Remote reads are synchronized just like local reads—if the data item is not available, the request is queued, and if the data item is available, the *page* containing that item is sent back. During this remote read the requesting PE can perform other useful work. The requesting PE may resume filling its subrange when the page arrives. This is where the benefits of array caching come in, and array caching is greatly simplified because of the single assignment principle.

Since the central idea in single assignment programming is to permit only one write to any element, by requiring single assignment we can guarantee that a page fetched from a remote PE and cached locally will not need any further updates during the lifetime of the array, ignoring for now the possibility of partially filled pages. Given this, each PE may safely cache a remotely fetched page in a local data cache, preventing future accesses of the same remote page. The cache used will be of fixed size and thus must use some sort of page replacement strategy. For our simulation, we chose a least–recently–used page replacement strategy. This choice leads to some interesting results discussed in a later section.

Without single assignment, partitioning data among PEs is possible, but it would require excessive communication overhead to allow any instruction to write to any location of an array. In addition, caching would be nearly useless as each write performed would require the update of all remote caches containing the modified page. The machine could broadcast or multicast these updates to avoid the inefficiencies of individual messages, but the broadcasts would still strain the network facilities. Not only that, but without single assignment the caches would be inconsistent for the duration of the page modification broadcast (cache coherency problem). If no cache approach is taken, no page modification broadcasts will be necessary, and there will be no inconsistency problems. But, the use of caching leads to considerable decreases in total remote accesses performed as is shown later in our simulation results.

We consider a set of loops (extracted from the Livermore Loops benchmark program) with data access patterns that are typically found in scientific programs. Using these we show that a simple data partitioning approach works well even with many PEs.

The main questions we are interested in answering are:

Given a simple static data partitioning scheme,

- how important is each program's access pattern?
- what is the overall percentage of remote accesses?
- how much can this be reduced by adding a data cache?
- how well balanced are the remote accesses?

5. Problems with Single Assignment

From the above discussion we see that enforcing single assignment policy can offer several advantages for MIMD architectures. Experience with single assignment languages has shown, however, that it is difficult to implement programs under such a restriction.

Much of this attitude arises from the ingrained nature of the von Neumann model. The requirement of single assignment is not as restrictive as it might appear—from a programming standpoint, there are several alternatives, including:

- Use a single assignment language. Here the rule is enforced automatically through the language semantics. In some cases, adherence can be determined completely at compile time (e.g., functional languages [ACK82]).
- Use a conventional language. In this case, the burden is placed on the programmer to ensure that the rule is not violated. In most cases, the same programming techniques and algorithms can be used, but arrays cannot be reused—once written, they cannot be changed. A way to relax the single assignment policy in a controlled manner so that memory costs do not become too high is presented below. Conventional compilers can be modified to perform data path analysis to help programmers adhere to single assignment rules.
- Use an automatic conversion tool. For many conventional loops, this conversion will be straight-forward and can be done by a translator program. These translators will tend to increase the amount of memory used for array storage, especially in those programs that reuse arrays many times in the same loop.

In statically allocated systems, the resulting inefficiency with memory usage can be solved by providing a special array re-initialization construct. Each PE's re-initialization must synchronize in some way with the re-initialization requests of all other PEs. We have formulated a method for performing this synchronization that is based on the concept of a *host processor*. In this method, each array in a computation has a specific PE assigned to it as an administrative center called the host processor. The host processor serves as a gathering point for re-initialization messages. In order to evenly divide this work among all PEs, the compiler ensures that the host processors are evenly distributed among the arrays. For the re-initialization of some array *A*, each PE sends a re-initialization message to *A*'s host processor. These messages are collected until the last PE has requested re-initialization. Once this happens, the host processor for *A* broadcasts a message to the other PEs informing them that *A* can now be reused. Thus, the host processor acts as a synchronization point for *A* so that no PE uses attempts to write to an out-of-date version of *A*. This prevents the creation of too many copies of an array in tight loops at the expense of an artificial synchronization point. Deallocation of arrays must be based on the same kind of host processor synchronization. (A more complete discussion of this mechanism is beyond the scope of this paper.)

6. Description of the Simulation

In order to study the effect of using a single assignment MIMD machine with a per-PE array cache, we implemented a simulation to measure the distribution of local, cached, and remote reads for an abstract multiprocessor architecture. The parameters that we varied were:

• number of processors

• page size (in units of atomic data elements)

Since the main goal of the simulation was to show that an array cache would decrease the percentage of remote accesses required, we chose a small fixed cache size (256 elements). Since the number of cache pages is dependent on the page size, the number of cache pages varied as well, but was not a simulation variable. Even a cache size this small proved sufficient to reduce the remote access percentage in many cases.

7. Simulation Results

Using the Livermore Loops, we mapped arrays onto a set of PE using the partitioning scheme described above with multidimensional arrays mapped to a linear address space through row–major ordering. Accesses to array elements were categorized as follows: write (always local), local read, cached read, remote read. The totals of each access type were accumulated for the execution of each program. For each loop, the percentage of all reads which were remote (% of Read Remote) indicates how well our approach handles the loop access pattern. Another important measure of performance is the distribution of work among the processors. The following sections present the results of the simulation.

7.1. Remote Access Overhead

By examining graphs produced by the simulation data, we were able to classify the various loops based on their access patterns. The four classes we observed are described below.

7.1.1. Class 1: Matched Distribution

The first class we observed consisted of those access patterns that have all array indices equal to one another throughout the execution of the loop, i.e. there is no skewing of array accesses. A typical loop fragment from a member of this class, 1-D Particle in a Cell, is:

```
      DO 1 k = 1,n
1     RX(k) = XX(k) - IR(k)
```

Note how the same array index is used for all array accesses in the calculation. Access patterns that fall into this class will always achieve a 0% remote access ratio. Caching has no effect on the access ratio since each PE can write to its segments by reading segments of the other arrays locally.

7.1.2. Class 2: Skewed Distribution

The second class we observed, skewed distribution (SD), displays a sequential access patterns in matched distribution, but the indices used in each array are offset from one another by a constant. As the index steps through the arrays, remote accesses will need to be performed for the elements lying past page boundaries. Since a page boundary implies a remote access (except for the single PE case), the loops in this class perform remote accesses.

We found that loops in this class occur often in the Livermore Loops. For example, Hydro Fragment, Tri-Diagonal Elimination, Equation of State Fragment, Explicit

Hydrodynamics Fragment, First Sum, and First Differential were all in this class. The inner loop fragment from Hydro Fragment is:

```
      DO 1 k = 1,n
   1  X(k) = Q + Y(k)  *  (R*ZX(k+10)  + T*ZX(k+11))
```

SD access patterns tend to achieve a very low (< 10%) remote access ratio (see Figure 1). This is because the access pattern displays a large amount of locality of reference—the number of remote accesses is usually small as the skew is generally a few elements. When the skew is large, the remote access percentage increases, but caching eliminates the cost of a larger skew. The effect of caching in this case depends on the value of the skew constant. For a skew of one, the cache has no effect, for a skew of two, the cache saves one remote access, and so on. For larger page sizes, the cache helps proportionally to the page size. Of course, if the page size is too large, the work will not spread over a sufficient number of PEs.

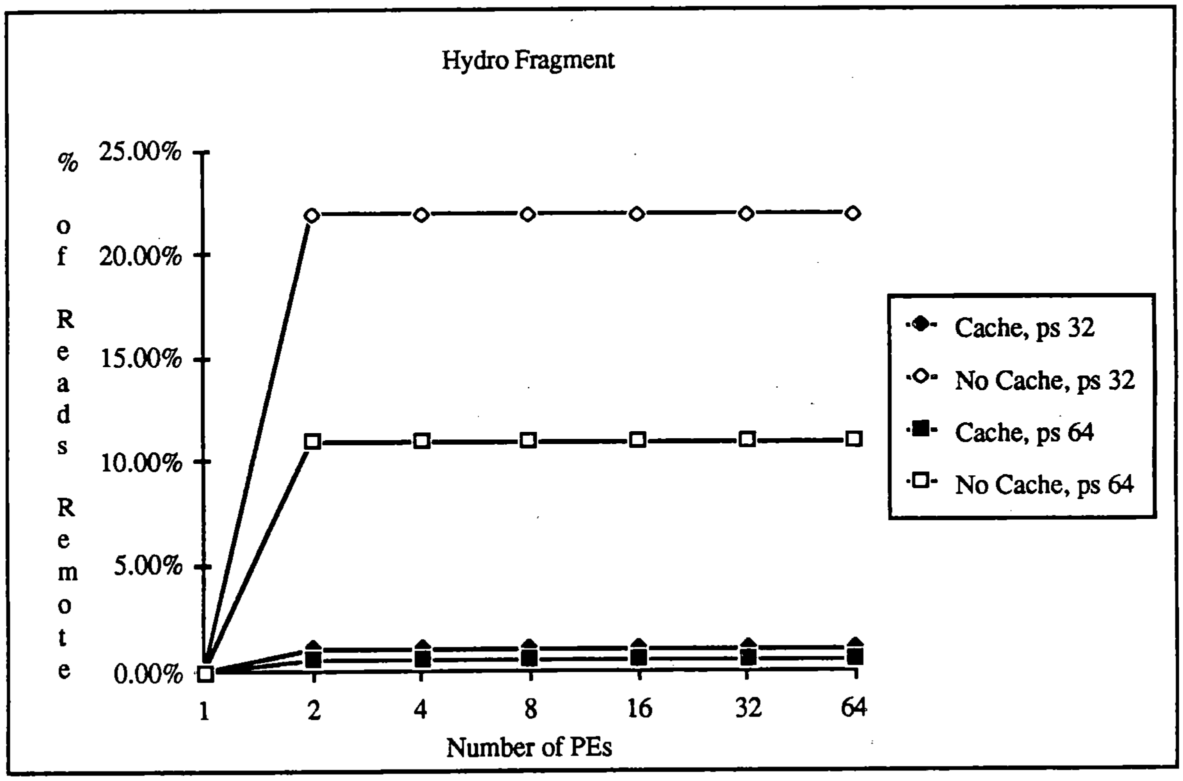

FIGURE 1. SKEWED ACCESS PATTERN (SKEW OF 11). CACHING IS IMPORTANT IN THIS COMMON CLASS.

7.1.3. Class 3: Cyclic Distribution

This third class, cyclic distribution (CD), occurs when a fixed set of pages is accessed in a cyclic order. The Incomplete Cholesky-Conjugate Gradient is an excellent example of this. The bulk of the loop is:

```
   II = n
   IPNTP = 0
22 IPNT = IPNTP
   IPNTP = IPNTP + II
   II = II/2
   i = IPNTP
   DO 2 k = IPNT+2, IPNTP, 2
   i = i + 1
2  X(i) = X(k) - V(k)*X(k-1) - V(k+1)*X(k+1)
   IF (II.GT.1) GOTO 22
```

Note that this is single assignment; the characteristics of this loop restrict the value of *i* such that $i > k+1$. The access distribution is cyclic because the write index (*i*) is changing twice as slowly as the read index (*k*). This allows caching to become nearly perfect as the number of PEs increase. At 32 PEs with cache size of 64, each PE is responsible for the writing of only one page. Once a remote read is done, the remote page remains cached.

Without a cache, CD displays poor performance, since the accesses jump from page to page and most are remote. However, with a cache the percentage of remote accesses decreases as the cache size increases *and* as the number of PEs increases. The explanation for this is that as the computation gets spread over more and more PEs, the total size of the cache increases. Thus, as the number of PEs increases and each PE is responsible for writing a smaller portion of the array, the cycle length tends to decrease for each PE. Given this, each PE is more likely to contain all of an access cycle in its cache (see Figure 2).

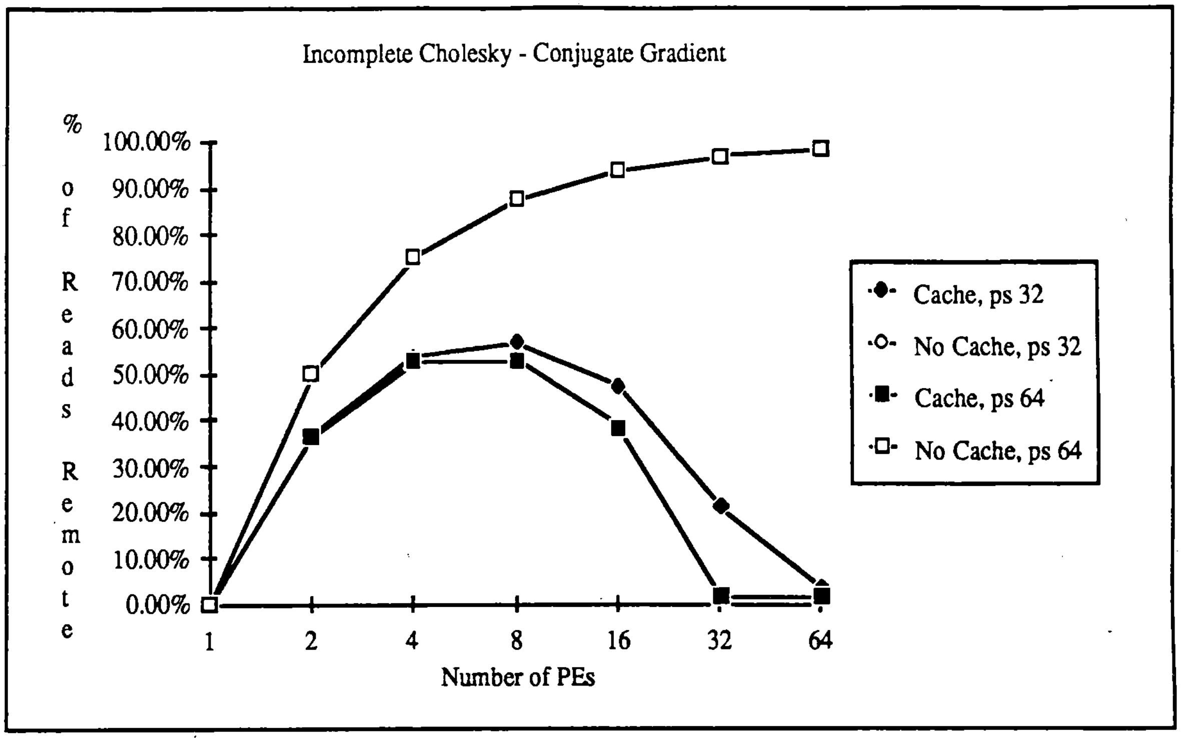

FIGURE 2: CYCLIC ACCESS PATTERN. CACHING AND PAGE SIZE CAN REDUCE THE PERCENTAGE OF REMOTE READS SIGNIFICANTLY.

The 2-D Explicit Hydrodynamics Fragment is an example of CD in which the cycling arises from the multidimensionality of the arrays. In one dimension, skewed distribution occurs, but in the other dimension, the pages are accessed in a cycle, so we observe a decrease in the percentage of remote accesses as the number of PEs increases. This behavior can be seen in Figure 3. An inner loop fragment from the 2-D Explicit Hydrodynamics Fragment is:

```
    DO 70 k = 2,6
    DO 70 j = 2,n
    ZA(j,k) = (ZP(j-1,k+1) + ZQ(j-1,k) - ZP(j-1,k) - ZQ(j-1,k))
          * (ZR(j,k) + ZR(j-1,k)) / (ZM(j-1,k) + ZM(j-1,k+1))
    ZB(j,k) = (ZP(j-1,k) + ZQ(j-1,k) -  ZP(j,k) - ZQ(j,k))
          * (ZR(j,k) + ZR(j,k-1)) / (ZM(j,k) + ZM(j-1,k))
70 CONTINUE
```

Notice how both indices are skewed such that a cycle occurs in the access pattern.

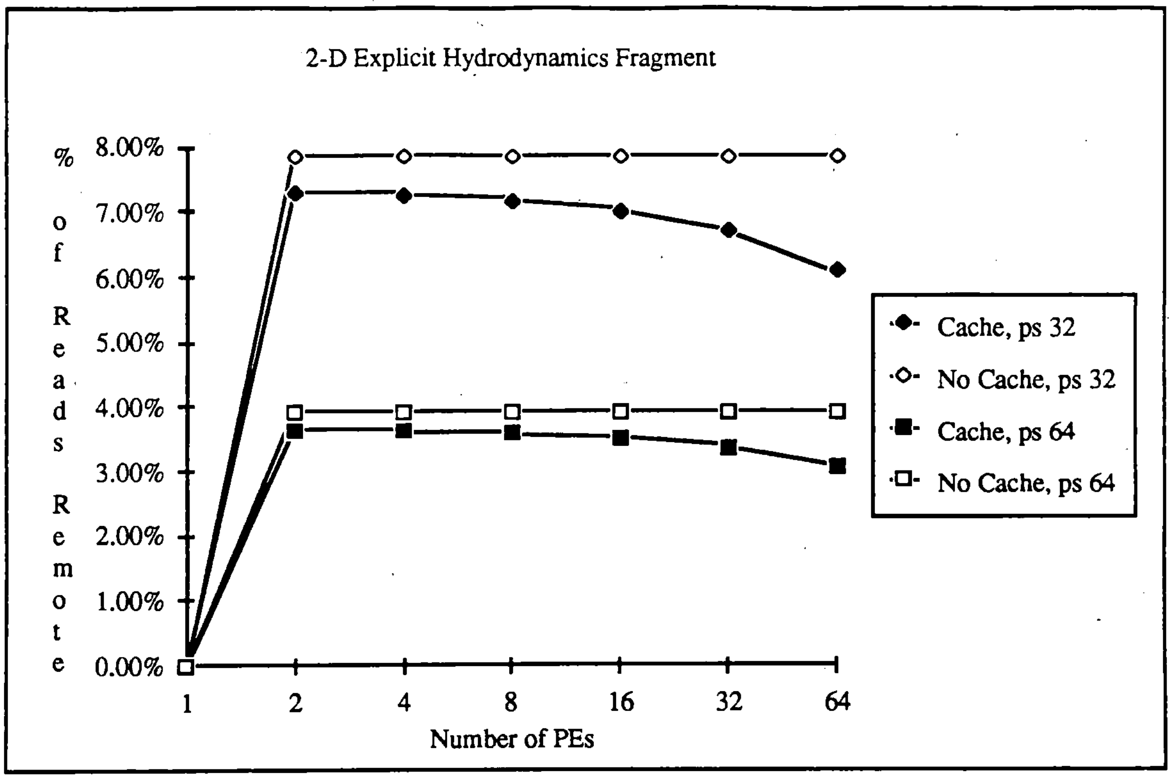

FIGURE 3: CYCLIC AND SKEWED ACCESS PATTERN COMBINATION. EXHIBITS EXCELLENT RESULTS AIDED FURTHER BY CACHING.

The examples above are rather counter-intuitive, yet very important results. Currently we are conduction further research to determine under what configuration or parameters a given program would approach 0% remote access ratio.

7.1.4. Class 4: Random Distribution

The final class is the random distribution (RD). RD covers loops that access various parts of the linear address space in a seemingly random fashion. This behavior can occur when multi-dimensional arrays are combined with skewed accesses. The General Linear Recurrence Equations and A.D.I. Integration are both in this class. Inner loop statements from the A.D.I. Integration are:

```
DO 8
U1(kx,ky,2) = U1(kx,ky,1) + A11*DU1(ky) +A12*DU2(ky) + A13*DU3(ky)
            + SIG*(U1(kx+1,ky,1) - 2.*U1(kx,ky,1) + U1(kx-1,ky,1))
U2(kx,ky,2) = U2(kx,ky,1) + A21*DU1(ky) +A22*DU2(ky) + A23*DU3(ky)
            + SIG*(U2(kx+1,ky,1) - 2.*U2(kx,ky,1) + U2(kx-1,ky,1))
```

```
      U3(kx,ky,2) = U3(kx,ky,1) + A31*DU1(ky) +A32*DU2(ky) + A33*DU3(ky)
                  + SIG*(U3(kx+1,ky,1) - 2.*U3(kx,ky,1) + U3(kx-1,ky,1))
    8 CONTINUE
```

RD exhibits large remote access ratios regardless of the presence or absence of caching (see Figure 4). This invariance can be due either to a cycle in the access pattern that is too large to fit in the cache, or to effectively random page accesses (e.g., permutation lookups). The effect of the cache is minimal, because no page is being kept until it is needed again. This is similar in many ways to thrashing in virtual memory systems. It is possible that increasing the number of PEs will help only if the access patterns form a cycle that is too large to fit in the cache. Increasing the cache size will help here by allowing a complete cycle to reside in the cache or increasing the probability of a cache hit simply by having more of the remote pages stored locally.

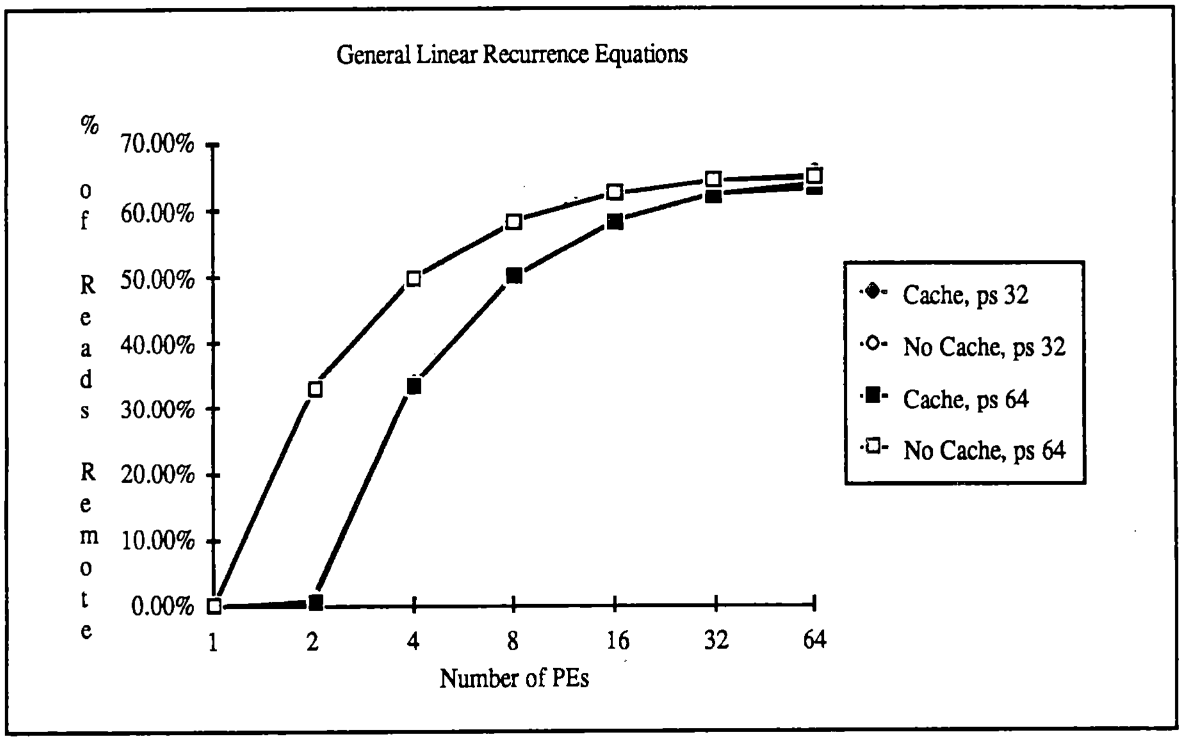

FIGURE 4. RANDOM ACCESS PATTERN. POOR PERFORMANCE OF RD CAN BE OVERCOME BY LARGER CACHE SIZES.

7.2. Load Balancing

The previous section showed that automatic partitioning can result in very small ratios of remote accesses when measured over the entire processor network. Another important aspect of automatic partitioning is load balancing (i.e., how evenly distributed are the computations?).

To consider load balancing behavior we use the number of remote and local reads per PE as a measure of how well the program is distributed. Figure 5 shows that each of the sixty-four PEs performs a comparable number of remote reads and local reads, hence the *area-of-responsibility* concept balances most loops well. In each loop, each PE performs similar amounts of remote access because each PE was responsible for similar amounts of the array. In cases where the amount of remote reads depends upon which element is being written, the load balance can be skewed. In these cases the lightly loaded PE can continue on with the program or context switch to another program. We found that nearly all of the Livermore Loops exhibited a load distribution pattern like that in Figure 5.

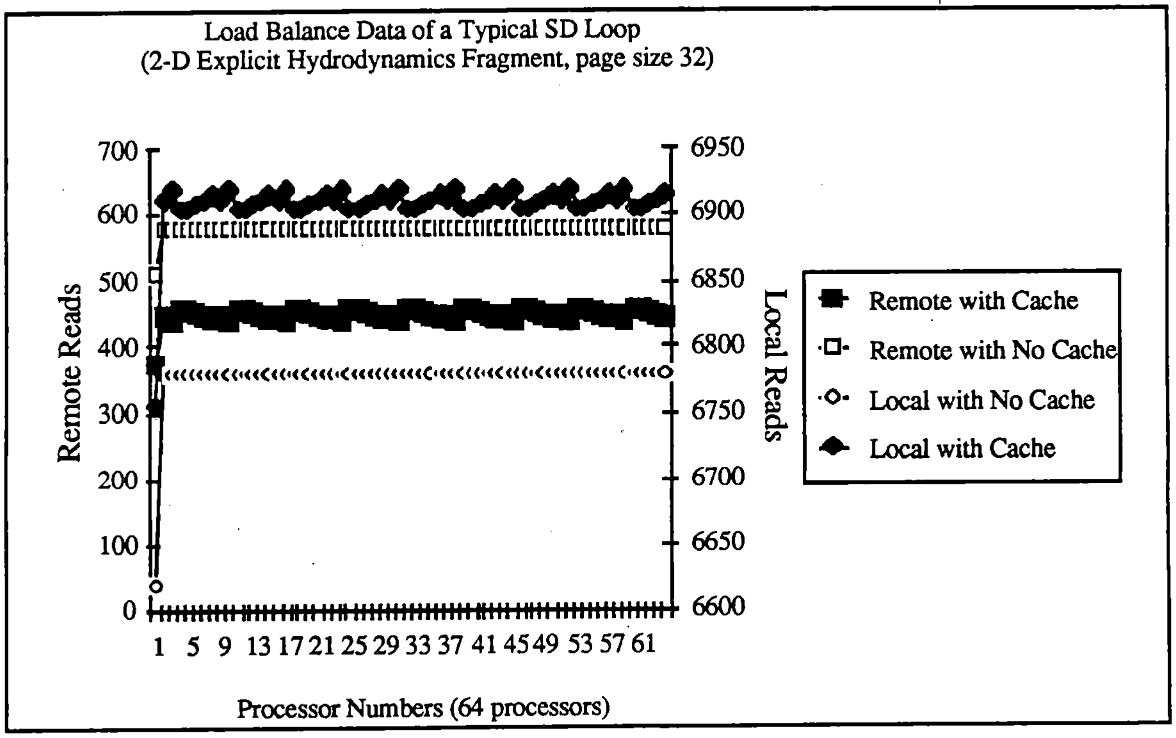

FIGURE 5. TYPICAL REMOTE ACCESS LOAD BALANCE. EVENLY BALANCED LOADS RESULT FROM THE AREA-OF-RESPONSIBILITY CONCEPT.

8. Conclusions

The combination of single assignment, areas-of-responsibility, and caching leads to low communication overhead and well-balanced loads when applied to the majority of the Livermore Loops. Single assignment permits the exploitation of large numbers of PEs automatically. Synchronization problems are solved through the adoption of the single assignment policy. By segmenting array writes using the area-of-responsibility concept, all PEs perform roughly the same number of remote accesses. These two concepts allow caching to be implemented without extensive communication, and caching is central to reducing remote accesses in the most common classes.

To answer our primary questions:

• How important is each program's access distribution?

Four different access classes cover the range of scientific computing. The most common class (SD) exhibits extremely low percentages of remote accesses (1% to 10%). Other, poorer performing classes (also less common) can be aided by larger cache sizes.

• What is the overall percentage of remote accesses?

For most access distributions, the percentages of remote accesses are less than 10% when using a cache of 256 elements (fairly small). For certain access distributions (RD) the remote access percentage can be rather high. We are continuing research into how to handle this special access distribution class.

• How much can the remote accesses be reduced by adding a data cache?

Depending upon the access distribution class, caching can have anywhere from a minimal effect to an extremely large effect (e.g., for an SD loop with large skew, we observed a reduction from 22% remote reads to 1% remote reads). Since SD is by far the most common class, this reduction is significant in many areas of scientific parallel computing.

• How well balanced are the remote accesses?

Because single assignment and equal partitioning force a nearly equal number of writes on each processor, the number of remote reads are also fairly equal. Thus the remote accesses are well balanced for the majority of cases. Our simulation results show this to be true for almost all of the Livermore Loops. The exceptions are those computations that are inherently difficult to parallelize under any paradigm and exhibit access patterns that correspond in many ways to thrashing in virtual memory systems.

Process alignment, as currently being considered by some researchers, ([PGW87] and [A&N87]) is no longer necessary. The analysis used in process alignment was used to transform SD loops to decrease communication overhead. By caching the elements in pages, the locality of reference in SD loops is exploited, and only one remote read is necessary for all elements in a page (in real systems, a single page might have to be fetched more than once if that page is only partially filled at the time of the first request, but the overall communication overhead will still be much smaller).

9. Future Research

This is the first step in the development of a new approach to distributing arrays. The concepts presented here play a key role in the design of a parallel model of execution on which we are currently working. To further understand the advantages and disadvantages of this approach, we need to examine a variety of issues:

- How will vector to scalar operations be implemented? Current ideas include the extension of the host processor mechanism to allow collection of subrange results.
- A more sophisticated simulation will better explore the problems of execution time and network contention.
- A better approach to RD access patterns is needed. Different partitioning schemes need to be explored as well as larger cache sizes. We will look into how the techniques developed for handling thrashing in virtual memory systems apply to this model.

- If it turns out that the different classes of access patterns form a nonintersecting set with respect to performance under different partitioning methods, then we must explore ways for providing different programmer– or compiler–selectable partitioning schemes. These would allow the programmer or compiler to select the partitioning method based on some analysis of the access behavior. For example, we have seen that our simple modulo partitioning scheme performs worse for certain loops than a division scheme. If no third scheme can be found that allows all types to perform well, it may become necessary to allow the selection of one or the other scheme based on the access distribution class.
- Other parameters might be programmer– or compiler–selectable. For example, allowing the programmer or compiler to select the page size might prove useful for reducing communication overhead in some classes of loops. We need to determine if such variability can be provided efficiently.

We are currently extending our simulation so it provides more information, and we are adding the mechanism described in this paper to a low level "emulation" of the execution model we are developing. Based on these preliminary results, we believe that our approach will eventually answer a difficult question in distributed processing: how can data be efficiently distributed?

10. References

[A&C86] Arvind and D.E. Culler. Dataflow Architectures, *Annual Reviews in Computer Science*, Vol. 1 1986, pp. 225-253.

[A&N87] A. Aiken and A. Nicolau. Loop Quantization: an Analysis and Algorithm, Technical Report 87-821, Dept. of Computer Science, Cornell Univ., March 1987.

[ACK82] W.B. Ackerman. Data Flow Languages, *Computer*, Feb. 1982, pp. 15-24.

[ANP87] Arvind, R. Nikhil, and K. Pingali. I-structures: Data Structures for Parallel Computing, Computation Structures Group Memo 269, Laboratory for Computer Science, MIT, February 1987.

[DEN75] Dennis, J. B. First Version of a Dataflow Procedure Language. MAC Technical Memo 61, MIT, Cambridge, Mass.

[PEI86] J.-K. Peri. Program Partitioning and Synchronization on Multiprocessor Systems, Ph.D. Thesis, Univ. of Illinois at Urb.-Champ., Rept. No. UIUCDCS-R-86-1259, Mar. 1986..

[PGW87] J. Peir, D. Gajski, and M. Wu. Programming Environments for Multiprocessors, *Supercomputing*, North-Holland, 1987, pp. 73-93

[R&F87] D. A. Reed and R. M. Fujimoto. *Multicomputer Networks: Message-Based Parallel Processing*, MIT Press, 1987.

[S&F83] *Architecture and Applications of the HEP Multiprocessor Computer System,* Denelcor, Denver, Colorado, 1983.

[SMI81] B.J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System, *Society of Photo-Optical Instrumentation Engineers*, Vol. 298, Real-time Signal Processing IV, Aug. 1981, pp. 241-248.