NASA Contractor Report 181847
ICASE Report No. 89-33

ICASE

INDIRECT ADDRESSING AND LOAD BALANCING FOR FASTER SOLUTION TO MANDELBROT SET ON SIMD ARCHITECTURES

Sherryl Tomboulian

Contract No. NAS1–18605
May 1989

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665–5225

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Indirect Addressing and Load Balancing for Faster Solution to Mandelbrot Set on SIMD Architectures

Sherryl Tomboulian *
MasPar Computer Corporation
2840 San Tomas Exprswy, Suite 140, Santa Clara, CA 95051

Abstract

SIMD computers with local indirect addressing allow programs to have queues and buffers, making certain kinds of problems much more efficient. In particular we examine a class of problems characterized by computations on data points where the computation is identical, but the convergence rate is data dependent. Normally, in this situation, the algorithm time is governed by the maximum number of iterations required by each point. Using indirect addressing allows a processor to proceed to the next data point when it is done, reducing the overall number of iterations required to approach the mean convergence rate when a sufficiently large problem set is solved. Load balancing techniques can be applied for additional performance improvement. Simulations of this technique applied to solving Mandelbrot Sets indicate significant performance gains.

*This research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18605 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665.

1 Introduction

The parallelism of SIMD architectures provides significant improvement in computation speed for many applications. However, one is limited by the synchronous nature of such an engine, which requires that all processors perform the same task at the same time. In SIMD architectures that provide hardware for local indirect addressing, some of the sequential restrictions can be alleviated, making such architectures appear more closely aligned to the MIMD paradigm. This paper examines a powerful use of local indirect addressing for performing computations with data-dependent convergence rates – some points converge in a few iterations while others require numerous iterations. Load Balancing techniques, typically considered a strictly MIMD technique, are brought into play. In particular, we examine the solution to the Mandelbrot Set using this method. Simulation results show significant speed improvements with modest code investment. The basic scheme can be generalized to a class of problems.

We assume a SIMD machine model with local indirect addressing, henceforth called *SIMLAD* [8]. SIMD implies that all processors do the same thing at the same time, and that each processor has an area of local memory. It is often assumed that a pure SIMD model also restricts the processors to performing their actions on the same areas of their local memory. While this was true of some architectures such as the MPP [1], the ICL-DAP [7] , and the Connection Machine 1[6], it is not a requirement. Now the Connection Machine 2 and designs of other new SIMD architectures, such as the BLITZEN [2] are including SIMLAD, suggesting a trend in this direction.

In the SIMLAD model, as in the "standard" SIMD model, each processor has its own local memory. However, the address specified by an instruction can be part of each processors local state. For instance, in the MPP, when an instruction is issued each processor does the same thing to the same area of its local memory – e.g. all processors add 1 to location 9. It would not be possible for one processor to add to location 9 and another to location 11. In our model each processor has an address register. Using SIMLAD, the previous example could be "Add 1 to the location specified by the address register", where one processor has a 9 in its register, another an 11. This does not violate the SIMD philosophy. All processors perform the same action at the same time, but they are not restricted to performing it on the same area of their local memories.

The use of Indirect addressing is a very powerful tool, especially when dealing with architectures such as the CM-2, where the local memory is no longer confined to a few thousand bits, as in the MPP, but is 64K bits. As yet, however, this feature has not been widely exploited in SIMD architectures. Here we explore one facet of this for a particular class of problems.

2 The General Problem

There are many problems involving the same computation on different data points which have different convergence rates. This is prevalent in the solution of differential equations, particularly techniques using mesh refinement. Perhaps the simplest problem for illustrating different convergence rates is the solution of the Mandelbrot set which is described in detail later.

Suppose that each processor is responsible for one datum. The required computation will occur in each processor, and as each processor reaches it's termination condition, it will disable itself. Obviously, completion of the entire task is dependent on the number of iterations required by the last processor that finishes. For example, we might say:

```
enable all processor;
for i=1 to maxiterations
    forall enabled processors
         perform the computation;
         if convergence reached disable processor;
```

If the average number of iterations required before convergence is much smaller than the maximum number, then many processors will be idling waiting for the last few to finish. If there is only one datum per processor, not much that can be done to hasten this process, But, if each processor is responsible for the same computation on multiple data items, then the situation is quite different.

It is reasonable to assume that using some of the new SIMD computers each processor could be responsible for upwards of 1000 data elements. In the classic SIMD mode of computation the code would be identical to the fragment given above and would iterate through the data elements. In the pseudo-code below, the notation Vector x[1000] means that each processor holds a local variable X, which contains 1000 elements. If there are 20,000 processors, there would be $20,000 \times 1,000$, or $20,000,000$ data elements in total.

```
Vector X[1000];
  for h= 1 to 1000 /* for all data elements */
    enable all processor;
    for i=1 to maxiterations
        forall enabled processors
             perform the computation on datum x[h];
             if convergence reached disable processor;
```

In this case, computation would take time $1000 \times$ ***maxiterations*** because we assume that at least one of the processors is computing a point that will require maxiterations for convergence. Hence, this solutions is governed by the maximum number of iterations. In cases where the mean number of iterations required for convergence is significantly less than the maximum, many processors will be sitting idle, wasting cycles. In the next section we will see that by using indirect addressing and invoking statistics and the law of large numbers, a solution dependent on the *average* number of iterations rather than the ***maximum***, can be obtained.

3 The Solution

The solution is a valuable technique not currently used in SIMD programming. We are assuming a method where each processor is responsible for the computation for a large number of points.

In the above code fragment, computation is performed on each local $x[i]$ for the same i, and all processors must converge on a solution before they proceed to iterate on the next point. This is necessary in the "classic" SIMD model, but in SIMLAD a more natural approach can be taken. When a processor has finished computing on element $x[i]$, either because it has converged on a solution or because it has reached the limit on the number of iterations, it moves on to element $x[i+1]$. It doesn't care if its neighboring processor is still iterating on its local element $x[i]$. If there are a sufficiently large number of points in each processor, and the probability of convergence is random, then by using this technique all processors should finish at about the same time with high probability. This time should be proportional to the average number of iterations required for each point times the number of points in each processor.

In the pseudo code below for this method, the Vector data structure implies that there is one copy of this data structure on each processor. That is, each processor has it's own data array X with 1000 elements. Each processor has a local index into this data array and a local counter to keep track of the number of iterations it has performed on the current point. The FORALL statement subselects the processors whose local data elements meet the specified conditions.

```
Vector X[1000], /* local data */
       INDEX,   /* local pointer into X array */
       ITERS;   /* number of iterations for that data point*/

     FORALL processors
          INDEX <- 1; /* all processors start at the first */
          ITERS <- 0;

     while (some-processors-busy) /* termination for entire program*/
        forall processors with INDEX<= 1000
           perform computation on datum X[INDEX];
           ITERS <- ITERS + 1;
            /* for those processors whose elements converged, or reached
       the maximum allowed, proceed to the next datum */
           FORALL  processors that have converged
                OR (ITERS > Maxiterations) THEN
                 INDEX <- INDEX + 1; /* proceed to next datum*/
                 ITERS <- 0;         /* reset iteration count */
```

The actual implementation of this method is slightly more complicated. The different tests can be parameterized for better efficiency. For example, the above code checks for convergence and proceeds to fetch the next point on every iteration. Since a local-indirect fetch generally takes more cycles than a normal memory reference, it may be desirable to fetch a new point every tenth cycle. Likewise, it is not necessary to check for global convergence of all processors on every cycle. Let I_g represent the global convergence interval, and I_l represent the interval

before loading the next data point once convergence is reached. We introduce a scalar integer counter to check the number of iterations.

```
done <- FALSE;
count <- 0;
while NOT(done)
   forall processors with INDEX<= 1000
      perform computation on datum X[INDEX];
      ITERS <- ITERS + 1;
      FORALL converged OR (ITERS > Maxiterations) THEN
         set processor converged;
      if (count MOD I1 = 0) THEN /* if new load interval */
         forall converged processors   /* for those processors     */
            INDEX <- INDEX + 1;        /* that are done computing  */
            ITERS <- 0;                /* advance to the next point*/
   count <- count + 1;
  if (count MOD Ig = 0)
      if (global_convergence) done = TRUE;
```

This parameterized version allows one to tune the solution to the characteristics of the architecture and problem.

4 Analysis

The pure SIMD version of the algorithm takes time proportional to
$P * MaxI * T_C$.
where $MaxI$ is the maximum number of iterations and T_C is the time for one iteration of the computation.

On the other hand, for the SIMLAD model, assuming that points are randomly distributed among the processors, and that there are a sufficiently large number of points per processor, then we can approximate the time for execution by
$P * MeanI * T_C$
where $MeanI$ is the mean of the convergence rate for all points. We justify this in the following way. We know that the overall time is governed by the maximum time required for a single processor to converge. How do we know what the convergence rate of a single processor is? By the law of large numbers we know that for a sufficiently large random sample the mean of the sample will approach the mean of the distribution. Hence, for all processors the overall number of iterations will approach $P * MeanI$.

This simplified characterization implies that the second method would always be superior, but omits the overhead intrinsic in the new algorithm. A more complete characterization can be given by: $[P * (MeanI + I_L/2) * T_O] + T_G/2 * T_O$

and

$T_O = T_C + T_L/I_L + T_G/I_G$
where

- T_L is the time required to perform an indirect load
- T_G is the time required to perform a global convergence check
- I_L is the frequency interval necessary to perform an indirect load
- I_G is the frequency interval necessary to perform a global convergence check.
- T_O is the overall time required for one iteration, which includes performing the computation and the overhead for the test.

In this characterization the time required to finish an iteration is not simply $MeanI$, but has the additional time of $T_L/2$. This comes from the fact that a value will converge, but the processor cannot proceed to the next value until the indirect load has occurred which only happens every T_L times. Once converged, a processor has to wait at most T_L cycles. However, in the best case, the T_L check occurs the very cycle that the point converges, so the processor does not wait at all. Hence, on average, the processor has to wait an extra T_L cycles once it has stopped. (For $I_L = 1$, this factor of $T_L/2$ disappears, but writing this unique case in the equation is rather messy, and is not represented here.

T_O is a fairly straightforward derivation. It is T_C plus the overhead for global checking and load. Since the the global check and Indirect load are only executed every I_G and I_L time intervals respectively, then only that fraction of the computation is included in the time for one iteration.

The second term in the equation, $(I_G/2) * T_O$, is an insignificant portion of the overall execution time, but is included for completeness. Similar to the the $T_L/2$ term, once all processors have finished execution they may have to wait for the global convergence check to come around. Because I_G tends to be small compared to the overall number of iterations, and the equation is an approximation, the equation is more simply characterized by:

$P * (MeanI + I_L/2 * (T_C + T_L/I_L + T_G/I_G)$

When comparing the SIMD and SIMLAD approaches, in cases where the T_C is large, the overhead is insignificant so the mean only has to be slightly less than the maximum to benefit. For these situations the new algorithm is clearly superior.

If the computation is relatively small, and the mean convergence time is large but still less than the maximum, it is still possible to realize significant improvement by having a large I_F and I_G, which decreases the overhead. For example, if $MeanV = 500$ and $MaxV = 1000$, and computation is equal to overhead, then if $I_L = 1$, so that we are testing every time, then the new algorithm would just break even or be even a little worse. But, if $I_L = 100$, then the number of iterations increases to 550, but the overhead becomes insignificant providing better than a 40% improvement.

As we have previously stated, these evaluations are based on the fact that the processors have a random distribution of the points. The section on Load Balancing will address some of these issues for the solution to the Mandelbrot set.

5 Mandelbrot Simulations

The computation of the Mandelbrot set was chosen because it illustrates the concept clearly. A good description of implementing the set can be found in Scientific American [4], and a more mathematical discussion is given by Devaney [3]. While being a good problem for pedagogical purposes, the computation is so small that the overhead involved in indirect addressing is larger than in problems of greater complexity, but benefits are still significant. The technique extends naturally to more complicated algorithms.

The recursive statement used to define the set is $z = z^2 + c$, where c is a point in the complex plane. We stop the recursive computation when $z > 2$, indicating that it is diverging, or when $iterations > 1000$, whichever comes first.

We simulated this problem for a 100 processor system with 100 memory locations. Results were positive. Using the "standard" approach, the computation would take $maxiterations * memorysize$, which in our case would be 100,000 iterations. We found that this could be decreased significantly, although performance varies over different regions and granularity of the Mandelbrot set.

For example, we ran the problem on the region (± 1,± 1), and achieved completion in 82,175 iterations, a 20%improvement. However, over this region we noted that the total average number of iterations needed was actually 54,529, so there was actually room for almost 50% improvement. Unfortunately, in the Mandelbrot set, as in most real problems, the convergence rate is not uniformly distributed, but rather has areas in which all points tend to converge very quickly or very slowly. For further improvements we turn to the technique of load-balancing.

6 Load Balancing

The methodology of load balancing [1] is traditionally not applied to SIMD architectures because it appears to be contrary to the synchronous single control nature of the system. However, we have shown here that a single control does not preclude working on different areas of the data. The key to the success of the method presented here is to have enough points per processor and the points should be representative of the whole. Under these conditions the overall average convergence rate at each processor is almost the same and is close to the mean of the convergence.

As previously mentioned, in the Mandelbrot set the convergence rate is not uniformly distributed. Hence the straightforward approach we used of allocating one column of the complex plane to a processor is not necessarily the most effective for this method of computation.

We tried two additional approaches for allocating points to processors. A fairly simple technique takes the column allocation method method and skewed each row of data randomly in the x direction so that the regions were not quite aligned with the processors. For example, for x[1] in all processor we might skew the value over 3 processors, for x[2] perhaps skew a negative 22 processors. The skewing method has an advantage that it provides some alleviation from

[1] See [5] for a general reference on Load Balancing Techniques.

the processors being assigned a contiguous region in the plane, but does not require expensive overhead computation to achieve this. Skewing in this manner is just a local computation based on the stride and skew amount. A more complicated approach involves performing a random permutation of the points as assigned to processors. Since random permutations are computationally expensive, this is the most extravagant of the methods.

As one might suspect, in the simulations performed, not counting set-up times, the completion rates varied relative to the complexity of the data distribution. The most permutation method finished first, the skewing method next, and the simple column allocation last. A surprising feature is that the simplest method, just assigning each processor a column in the plane, frequently performed well simply because the number of points per processor is fairly high. Even though there is some locality, the area picked still had significant variation. While the random permutation method usually performed the best it was not significantly better than the row skewing method and the overhead of the set-up time simply does not justify the performance gains. We advise row-skewing as a general optimization for this problem.

With the Mandelbrot set, the variation in mean convergence rate varies significantly depending on the region and granularity picked. The table below gives some empirical results obtained by our simulator for different regions. The simulator had 100 processors, each responsible for 100 points. (Actual hardware implementation of the SIMLAD variety described here would be expected to have at least 10,000 processors each with over 1,000 points per processor.) For these simulation parameters, the maximum number of iterations – the number required by a pure SIMD model – would be 10,000 iterations. The table gives three different iteration values. The first iteration value is the mean number of iterations for processor completion; this is be the ideal minimum. The second iteration value is the number of iterations using SIMLAD solution method presented here, but without load balancing, just assigning each column in the plane to a processor. The third iteration value gives the number of iterations when using the simple skewed load balancing technique. We follow the iteration number by two percentages. The first is the percent improvement of the SIMLAD method as compared to the non-indirect method. The second is the efficiency of the method; that is, how close it comes to the average number of iterations per processors, which is the theoretical minimum.

The table shows a variety of results from different regions in the complex plane.

Iterations needed to solve Mandelbrot with 100 processors, 100 points each								
real	imaginary	avg iters	new iters	%impr	%eff	newLB iters	%impr	%eff
(-2.0,+0.5)	(-1.25,+1.25)	24875	91079	9	27	29762	70	84
(-1.0,+1.0)	(-1.0,+1.0)	35462	68185	32	52	41294	59	86
(-0.6,-0.5)	(-0.6,-0.5)	34512	54456	46	63	38566	61	89
(+.26,+.27)	(+0.0,+.01)	65101	100000	0	65	70295	30	93
(-1.26,-1.24)	(+.01,+.03)	83403	97643	2	85	88320	12	94

The method described above uses a static load balancing scheme. We allocate the points

before computation begins and hope that we did a good job. Another alternative is dynamic load balancing. There are many forms that dynamic load-balancing can take, but dynamic load balancing involves communication, which is expensive in SIMD architectures. One approach is to have processors that have finished computing all their points reach over and grab some from their direct local neighbors. It is not clear that the overhead involved justifies the approach – it is an interesting area for future study.

7 Conclusions

As SIMD architectures develop and become more widespread new programming paradigms will arise to make efficient use of them. In this paper, we presented a method for using local indirect addressing to achieve faster solutions for some problems with data-dependent convergence rates. We investigated at a simple case, the Mandelbrot set, and achieved significant success. The traditional MIMD technique of load balancing proved highly effective, exemplifying the greater freedom and power of the SIMLAD approach. The general mechanism described here is a powerful technique that will no doubt become widely used in data parallel programming as SIMLAD architectures become more available.

References

[1] K.E. Batcher, "Design of a Massively Parallel Processor", IEEE Trans. on Computer, September 1980, pp. 836-840.

[2] E. Davis, J. Reif, "The Architecture and Operation of the BLITZEN Processing Element", 3rd Intl. Conf. on Supercomputing, May 1988.

[3] Devaney, R.L., *An Introduction to Chaotic Dynamical Systems*, Benjamin Cummings Publishing Co., Menlo Park, 1986.

[4] Dewdney, A.K, "Computer Recreations: A Computer Microscope Zooms in for a Look at the Most Complex Object in Mathematics", *Scientific American*, August 1985, pp. 16-21.

[5] G.C. Fox, et al, *Solving Problems on Concurrent Processors*, Vol 1., Prentice Hall, 1988.

[6] W.D. Hillis, *The Connection Machine*, MIT Press, 1985.

[7] S.F Reddaway, "DAP – a distributed array processor", First Annual Symposium on Computer Architecture, (IEEE/ACM), Florida, 1973.

[8] S. Tomboulian, D. Middleton, "Evaluating Local Indirect Addressing in SIMD Processors", ICASE Report(89-30) in preparation, Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton VA.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-181847 ICASE Report No. 89-33		

4. Title and Subtitle	5. Report Date
INDIRECT ADDRESSING AND LOAD BALANCING FOR FASTER SOLUTION TO MANDELBROT SET ON SIMD ARCHITECTURES	May 1989
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Sherryl Tomboulian	89-33
	10. Work Unit No.
	505-90-21-01

9. Performing Organization Name and Address	11. Contract or Grant No.
Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	NAS1-18605
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Contractor Report
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Final Report

16. Abstract

SIMD computers with local indirect addressing allow programs to have queues and buffers, making certain kinds of problems much more efficient. In particular we examine a class of problems characterized by computations on data points where the computation is identical, but the convergence rate is data dependent. Normally, in this situation, the algorithm time is governed by the maximum number of iterations required by each point. Using indirect addressing allows a processor to proceed to the next data point when it is done, reducing the overall number of iterations required to approach the mean convergence rate when a sufficiently large problem set is solved. Load balancing techniques can be applied for additional performance improvement. Simulations of this technique applied to solving Mandelbrot Sets indicate significant performance gains.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
SIMD processing	61 - Comp. Prog. & Software 64 - Numerical Analysis Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	10	A02