CONF-9008114--4

PNL-SA-17937

EFFICIENT ITERATION IN DATA-PARALLEL PROGRAMS WITH IRREGULAR AND DYNAMICALLY DISTRIBUTED DATA STRUCTURES

PNL-SA--17937

DE91 000417

R. J. Littlefield

February 1990

Submitted to the
International Conference on Parallel Processing
The Pennsylvania State University
University Park, Pennsylvania
August 13-17, 1990

Work Supported by
the U.S. Department of Energy
under Contract DE-AC06-76RLO 1830

Pacific Northwest Laboratory
Richland, Washington 99352

Efficient Iteration in Data-Parallel Programs With Irregular and Dynamically Distributed Data Structures

Richard J. Littlefield *

February 6, 1990

Abstract

To implement an efficient data-parallel program on a non-shared memory MIMD multicomputer, data and computations must be properly partitioned to achieve good load balance and locality of reference. Programs with irregular data reference patterns often require irregular partitions. Although good partitions may be easy to determine, they can be difficult or impossible to implement in programming languages that provide only regular data distributions, such as blocked or cyclic arrays. We are developing Onyx, a programming system that provides a shared memory model of distributed data structures and extends the concept of data distribution to include irregular and dynamic distributions. This provides a powerful means to specify irregular partitions. Perhaps surprisingly, programs using it can also execute efficiently. In this paper, we describe and evaluate the Onyx implementation of a model problem that repeatedly executes an irregular but fixed data reference pattern. On an NCUBE hypercube, the speed of the Onyx implementation is comparable to that of carefully handwritten message-passing code.

Keywords: non-shared distributed memory, Onyx, finite element method, NCUBE, repeating data reference patterns.

*This work was supported by the U.S. Department of Energy under Contract DE-AC06-76RLO 1830 at D.O.E's Pacific Northwest Laboratory, and by the National Science Foundation under Grants CCR-8619663 and CCR-9807666 at the University of Washington. The author can be contacted at the Pacific Northwest Laboratory, P.O.Box 999, Richland, WA 99352, or via email to rik@cs.washington.edu .

1 Introduction

Non-shared distributed memory multicomputers, such as MIMD hypercubes, are attractive architectures on which to implement parallel applications. In contrast to shared memory architectures, distributed memory systems devote less hardware to interprocessor communication. Potentially, this allows better price/performance and scalability. However, it is more difficult to write an efficient program for a distributed memory system, because more attention must be paid to keeping communication costs low.

In the past, efficient distributed memory applications have been coded by hand as explicit message-passing programs. Unfortunately, such programs are longer and more complex than uniprocessor or shared-memory programs to do the same job. Several research efforts, including our own, are working on language design and compilation techniques to make it easier to write programs for distributed memory machines.

We are currently developing Onyx, a programming system targeted for scientific applications. Our goal is to allow programs to be written easily and concisely, then compiled to run at a rate comparable to that of a hand-written message passing program. In this paper, we show that this goal is feasible for programs with irregular data reference patterns that are executed repeatedly.

The Onyx programming model supports data-parallel programs with a shared memory model of distributed data structures. *Data parallel* means that algorithms are expressed in terms of operations applied simultaneously to many elements in a collection of logically similar data objects. A *distributed data structure* is any collection of data that can be referenced by a single name, even though it is physically distributed across multiple processors, and *shared memory model* means that distributed data structures can be accessed with the same syntax and semantics as conventional non-distributed structures. For example, programs using distributed arrays can be written in terms of fetches and stores, using conventional subscripts that can be computed freely.

Systems providing a shared memory model of distributed data structures have been described by several other research efforts targeting scientific applications [11, 8, 10, 3, 9]. However, prior work has considered only distributions that are static and regular, such as blocked and cyclic arrays. Onyx extends the concept to include *irregular* and *dynamic* distributions. In the extreme case, every element of an array can be individually assigned to a specific processor at run time, then moved to different processors to maintain locality as the computation progresses.

This flexibility provides a powerful way to specify the *partitioning* (distribution of data and computations) for programs with irregular data reference patterns. However, accessing data that is distributed arbitrarily at runtime seems certain to be more expensive than accessing data that has a regular and static distribution. In this paper, we show that the difference can be made surprisingly small in some important cases.

This paper focuses on compilation and runtime techniques for programs that require irregular partitions and that execute each unique data reference pattern many times. Our model program is extracted from a real-world finite element application using irregular meshes, such

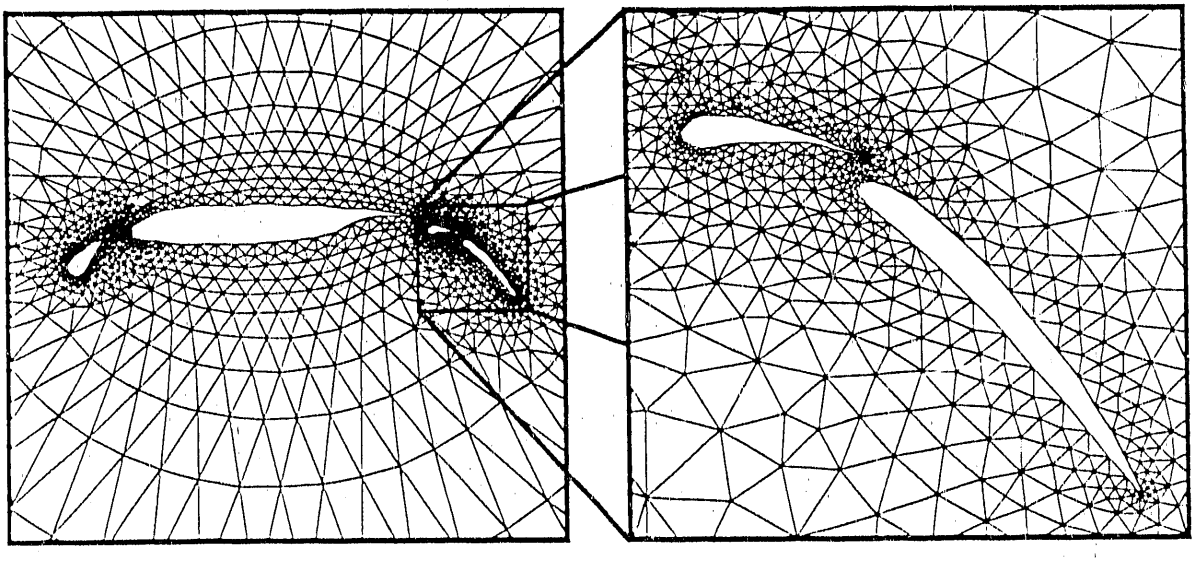

Figure 1: A portion of the irregular triangular mesh used by our model program. (Mesh courtesy Dimitri Mavriplis of NASA/ICASE).

as shown in Figure 1. This application is typical of many scientific codes, in that its patterns of data references are repeated many times in the course of a single execution. Each pattern is irregular and dependent on input data, but once established, it is executed many times. Onyx exploits this behavior to achieve low average access cost. The first time each pattern occurs, it is interpreted using a relatively expensive procedure that translates references to distributed data structures into combinations of local memory references and communications. This information is cached for use with subsequent occurrences of the same pattern. It turns out that this approach eliminates almost all of the overhead for subsequent iterations. The overall result is that execution speed of the Onyx implementation is comparable to that of carefully hand-written code.

In following sections, we discuss these issues in more detail. Section 2 gives a brief overview of the Onyx programming model. Section 3 describes the model problem and proposes language constructs to program it. Section 4 describes our basic implementation techniques, focusing on the treatment of distributed shared data. Section 5 discusses caching for repeated patterns. Section 6 presents and discusses performance data. Section 7 covers related work, and Section 8 summarizes and discusses future directions.

2 The Onyx Programming Model

Onyx is targeted for data-parallel scientific applications intended to run on distributed memory multicomputers. Its programming model includes the following features:

- *Explicit data parallelism.* Onyx provides an execution model with a single thread of control, whose operations are performed in parallel in each of many logically similar data objects. The programmer is expected to choose algorithms for which this model is appropriate, and to write code in which the data parallelism is clearly indicated.

  The examples in this paper are written in a dialect of Pascal augmented by a `forall` statement. This choice was made solely for clarity of presentation, based on the fact that Pascal has a `record` declaration whose syntax and semantics are familiar to most readers. The same functionality clearly could be achieved with other base languages, by augmenting the syntax or recognizing data parallel code fragments through control and dataflow analysis.[1]

- *Distributed data structures with a uniform name space.* Collections of data elements, such as arrays, can be physically distributed across processors, but programs reference data elements by name and subscript without regard for where they physically reside.

- *Shared memory data model.* Access to distributed data structures is done with the same language constructs and semantics as for non-distributed data. That is, the statement `a[i] = b[j[i]]` is valid and does the same thing regardless of whether `a`, `b`, and `j` are distributed, and if so, how.

- *Partition-independent coding and results.* Onyx assumes responsibility for transforming loop headers, creating processor-local data structures, synchronizing, and communicating as required to implement whatever partitioning is specified. Computational code, as written by the programmer, is independent of the partitioning, and programs produce the same results regardless of how the partitioning is changed.[2] This allows a program to be tuned for maximum performance without altering its correctness.

- *Explicit partitioning.* The underlying machine topology and attendant communication costs are exposed by allowing (but not requiring) the programmer to explicitly assign data and computations to particular processors. Again, this assignment is a matter of tuning for performance and does not affect computational code or results. Onyx will provide defaults for any partitionings that are not specified, based on heuristic analysis of control and dataflow.

3 Programming the Model Problem

Our model problem is extracted from an application that solves for air flow over a wing, using an irregular triangular mesh like the one shown in Figure 1. One portion of that

[1] In fact, we anticipate that most of our future work will be based on Fortran because of the availability of application codes already written in that language.

[2] This restriction may be relaxed in deference to efficiency, when dealing with certain operations that are technically non-associative, such as floating-point addition.

```
var
  v: array [1..MAXNODES] of float;              /* node values */
  vnew: array [1..MAXNODES] of float;           /* temporary new values */
  nnbr: array [1..MAXNODES] of integer;         /* number of neighbors */
  nbr: array [1..MAXNODES,1..MAXNEIGHBORS];     /* indices of neighbors */
  c: array [1..MAXNODES,1..MAXNEIGHBORS];       /* multiplying constant */
  int nnodes;                                   /* actual number of nodes */
  t: float;

SmoothNodes ()
{ for i = 1 to nnodes {
    t = 0.0;
    for j = 1 to nnbr[i] {
      t += c[i,j] * v[nbr[i,j]];
    }
    vnew[i] = t;
  }
  for i = 0 to nnodes
    v[i] = vnew[i];
}
```

Figure 2: Conventional serial code for the smoothing calculation on an irregular mesh.

application executes a smoothing calculation that simply replaces each node with a weighted average of its neighbors. We choose that function for our model program.

For a uniprocessor, the calculation might be expressed as shown in Figure 2. Note that the data reference pattern of this program is both irregular and repeating. It is irregular because the mesh topology is determined by arbitrary indices contained in the nbr array (as opposed to having a parametric form), and it is repeating because those indices don't change between executions of SmoothNodes.

To parallelize this program requires two tasks. First, a good partition must be *determined*, i.e., data nodes (array elements) and computations must be assigned to processors so that, as far as possible, neighboring nodes reside in the same processor and all processors have an equal amount of work. In this application, there is a natural spatial relationship between interacting nodes, so it is relatively easy to determine a good partition.[3] Second, the partitioning must be *implemented*, i.e., each processor must be loaded with data structures that represent the partitioning, and with a program that can use those data structures to efficiently perform the required computations and communications. In many cases, efficiently implementing the partitioning is more difficult than determining it.

With programming languages that support only regular distributions, the two partitioning tasks (determining and implementing) are intertwined. For example, arbitrary partitions cannot be specified in a system that supports only blocked and cyclic arrays. Given that limitation, one is forced to renumber data elements in subtle ways so that a regular distribution can be tricked into producing something close to the desired result. In general, it is not possible to achieve optimum partitioning using regular distributions. Blocked and cyclic distributions, for example, require that the number of array elements in each processor be the

[3]We used recursive bisection [5] in the experiments reported here.

```
type nodeT = record {
    v: float;                    /* node value */
    vnew: float;                 /* temporary new value */
    nnbr: integer;               /* number of neighbors */
    nbr: array [1..MAXNEIGHBORS] of integer;  /* indices of neighbors */
    c: array [1..MAXNEIGHBORS] of float;      /* multiplying constant */
    procID: prodID_T;            /* processor ID (for partitioning)
  }

var
  node: array [1..MAXNODES] of nodeT distributed arbitrarily;
  nnodes: integer;                          /* actual number of nodes */

InitiallyMoveNodes ()       /* partitioning -- one time only */
{ forall i = 1..nnodes
    node[i].MoveToProcessor (node[i].procID);
}

SmoothNodes ()              /* executed many times */
{ forall i = 1..nnodes on owner(node[i]) {
    with node[i] do {
      var t: float;
      t = 0.0;
      forall j = 1..nnbr {
        t += c[j] * node[nbr[j]].v;
      }
      vnew = v;
   } }
   forall i = 1..nnodes on owner(node[i])
     with node[i] do v = vnew;
 }
```

Figure 3: The smoothing calculation in Onyx, using an arbitrary assignment of array elements to processors.

same, give or take one element. However, a good partition may place many more elements in some processors than in others, if those elements are easier to process. (Boundary nodes, for example, usually are handled differently from internal nodes.)

In contrast, Onyx provides a clean separation between the two partitioning tasks. Determining a good partition is declared to be outside the scope of the language, and is expected to be done with some application-specific tool. Onyx then concentrates on making it easy to specify any chosen partition and to implement it efficiently. In the case of our model problem, we simply retain whatever node numbering the programmer or user finds convenient, and specify the partitioning by individually assigning array elements to processors.

An Onyx program using this approach is shown in Figure 3. This Onyx program closely resembles a Pascal program to do the same job, and for the most part it can be understood to have Pascal's semantics. However, there are some features that require explanation.

- `forall` denotes a data-parallel piece of code. Conceptually, the body of the `forall` is executed simultaneously for each different value of the control variable.

- `distributed arbitrarily` declares that any element of the array can be associated with any processor. There is no parametric relationship between subscript and processor number.

- `node[i].MoveToProcessor(ID)` is an intrinsic function that causes the array element `node[i]` to be relocated onto processor ID. That processor becomes its `owner`. Note that since all the MoveToProcessor calls are embedded in a `forall`, the relocations are done in parallel. (Again, the processor ID's are expected to be determined by some application-specific partitioning tool. We use a stand-alone partitioner, and read processor ID's from an input file. They might equally well be computed on the fly.)

- `on owner(node[i])` causes the body of the `forall`, for each different `i`, to be executed on whichever processor is currently the owner of `node[i]`. This is an example of explicitly partitioning the computation. In this case, the explicit specification just echoes the default, which is to assign each instance to whichever processor appears to have fastest access to the required data, based on compile-time analysis. In other cases, it may be better to perform a computation on some other processor, perhaps to improve load balance at the expense of extra communication.

To implement this program, the Onyx compiler must transform it into a runtime form that efficiently iterates over local data structures and provides efficient communication for references to remote data. These topics are the subjects of the next two sections.

4 Distributed Shared Data

Onyx provides a *shared data model*, i.e., distributed data can be accessed from any processor simply by making a reference to it. All communications are implicit; the programmer's view is that array elements are simply fetched and stored. This model could be implemented in a variety of ways, differing especially in how much hardware support is used. For flexibility and portability, we use a purely software-based system that requires no memory management hardware and that maintains a dense address space on each processor.

The programmer specifies array references in terms of an array identifier and global indices. Translation from global indices to processor number and local address is specified by a mapping function or table that describes how the array is distributed. Onyx supports several styles of distribution, chosen somewhat arbitrarily by matching application requirements against the possibility of efficient implementation. A distribution may be fully parametric, such as a regular blocked or cyclic distribution. It may be piecewise parametric, such as an irregular blocked or rectangular decomposition. Or it may be fully enumerated, in which case any array element can be assigned to any processor arbitrarily. In this paper, we evaluate only the fully enumerated option.

At runtime, each array element may occur on only a single processor or may be *replicated* (copied) on several processors. In either case, each element is uniquely associated with one processor called its *owner*. The owner is responsible for keeping a master copy of the element

and making sure that replicates are updated as required. If an element's owner may change at runtime, as in our example, the distribution is said to be *dynamic*. In that case, the element is also statically assigned a *forwarder* processor, whose job is simply to keep track of who the current owner is and relay requests to it. The combination of owners and forwarders essentially comprise a distributed directory for dynamic distribution mappings.

In general, the system works as follows. To fetch an array element, a processor first checks to see whether a local copy exists, either because that processor owns the element or because it has been locally replicated. If not, it allocates space for the element and sends a request to that element's owner or forwarder. The owner responds by sending back the required data and making a note to itself that the replicate exists. The local copy, pre-existing or newly created, can then be accessed with just a local memory fetch. To update an array element, a message containing the new value is sent to that element's owner, which changes its copy and propagates the update as required. Race conditions are precluded by the data-parallel semantics of Onyx, so no mechanisms are required for mutual exclusion.

There are several obvious improvements to the basic scheme. First, remote references occurring in parallel code (`forall`'s) can be processed in batches. This allows multiple requests to be bundled into a single message to amortize startup costs. In a hypercube or other multipath network, bundling can also reduce routing overhead. Second, as in our model program, it is often practical to partition the computation so that all work needed to update an element is done by that element's owner. When this can be done, it avoids the necessity of collecting contributions from several processors, which allows one set of messages to be eliminated. Third, repetition can be exploited to reduce communications requirements, by sending data directly to where it is known to be needed. This avoids explicit request messages and the forwarding associated with them for dynamic distributions.

To exploit the repetition, we use a variation of the *inspector/executor* approach introduced by Crowley et.al.[4]. The first time a data reference pattern is seen, it is processed by the *inspector*, a relatively long and slow code that resolves all the references, figures out how to handle the replication, and builds tables to describe the required actions. Subsequent occurences of the pattern are handled by the *executor*, a short fast code driven by the tables. Both the inspector and the executor are generated by the compiler from the same source program.

This approach results in compiling the procedure `SmoothNodes` into a piece of code that works as shown in Figure 4. The exact code at points A and B depends on the replication policy. In most circumstances, the best approach probably is to create replicates the first time the pattern is encountered, and use them repeatedly through subsequent iterations. We call these *persistent replicates*. With persistent replicates, the code at point A will be executed only once, and the code at point B will be executed each iteration. If some of the replicated cells can be reused without updating, this approach will reduce data traffic. In the applications we have studied, opportunities for reuse appear frequently. In other circumstances, it may be better to use *transient replicates* that are discarded immediately after the computation. This might be appropriate, for example, if memory were very restricted and

```
        — inspector
        if (this data reference pattern is not known) then
           for i such that this processor owns node[i] and i is in 1..nnode
              determine which nodes [node[i].nbr[j]], etc. have a remote owner
                 and therefore need to be replicated
           endfor
           cooperate with other processors so that owners know about
              all the required replicates
           determine and remember which processors have replicates of nodes
              that this processor owns and needs to update
           remember that this pattern has been seen
        endif
        — executor
(A) ⇒   if required, create replicates
        for i such that this processor owns node[i] and i is in 1..nnode
           translate all node[node[i].nbr[j]] into local memory addresses
              and perform the computation
(B) ⇒   if required, propagate updates
```

Figure 4: Basic implementation of distributed data references in the Onyx program. The code at points A and B depends on the replication policy, as described in the text.

the replicates might have to be discarded after updating but before they could be reused.[4]

In terms of the volume of data transfer and number of synchronization points, this type of executor is as efficient as if the programmer coded the communications explicitly. As Mehrotra points out [8], unnecessary messages and synchronization are eliminated, since after the first execution of a pattern, each processor knows exactly what data must be exchanged with which other processors.

In terms of instructions executed, however, this type of executor still imposes overhead. Each array reference requires a conversion from global indices to local memory address, and remote references also require searching a directory of replicates. The impact of this overhead varies with the style of distribution and the amount of computation per reference. For simple computations and complex distributions like our model problem, the overhead is likely to be several times the computational cost. This would be unacceptable for practical use. In the next section, we will discuss modifications to the inspector and executor that can virtually eliminate this overhead.

It is important to note that overhead of this type cannot be seen in experiments that

[4]The Kali implementation of Koelbel and Mehrotra [7] essentially implements transient replicates, describing them in terms of "communication buffers".

```
— inspector
for i such that node[i] is local and i is in 1..nnode
   translate node[i] into a local address and save it in table T
   save node[i].nnbr in table T
   for j = 1..nnbr
      translate node[node[i].nbr[j]] into a local memory address
         and save it in table T
      endfor
   endfor

— executor
while (table T is not empty)
   retrieve np = address of node[i] from Table T
   retrieve n = nnbr[i] from table T
   t = 0.0
   for j = 1..n
      retrieve onp = address of node[node[i].nbr[j]] from table T
      t += np→c[j] * onp→v      /* pointer references */
   endfor
   np→vnew = t
endwhile
```

Figure 5: Using address caching to improve execution speed for repeated patterns of references to distributed data.

merely vary the number of processors. Since all processors do similar amounts of overhead work, the overhead exhibits perfect speedup. The program runs more slowly with any number of processors, but the speedup curves look better. To see the overhead, one must compare against an alternative implementation that doesn't have it.

5 Caching the Data Reference Pattern

There is a general rule for making programs run fast: don't recompute what you can tabulate. In our case, the inspector follows this rule by carrying its symbolic execution all the way down to the level of computing local memory addresses, and saving these for use by the executor and communication routines.

The exact code depends on the policy for handling replicates. The simplest situation is when replicates retain their absolute addresses between iterations. In that case, to continue our example, the inspector and executor contain code that works as shown in Figure 5.

Note that essentially all the overhead costs have been moved into the inspector, which is executed only once per pattern. This includes not only the reference resolution, but also whatever time may be required for each processor to determine which computations it has been assigned (i.e., to execute the code described by "for i such that..." in Figure 4).

The remaining processor overhead is primarily due to the need to determine whether the data reference pattern is already known. In general, this requires a combination of compile time dataflow analysis and runtime checking. In our example, dataflow analysis indicates that the pattern depends on the distribution of node and the values of nnbr and nbr at each node. In the context of our model problem, it is easy to determine that these do not change after initialization. Realistic situations will be more complicated, especially if procedure parameters are involved. We anticipate that an adequate solution can be obtained by having the runtime system combine scalar values and structure update times into a unique key that is associated with the pattern, but this issue requires further study. In any case, the cost of determining whether the pattern has been encountered before is amortized over all the calculations done by the executor.

This executor's computational loop is quite efficient. In fact, it is essentially identical to what one would get by replacing the nbr indices with direct pointers to the referenced nodes. We generate similar executors for packing, unpacking, and routing communication buffers needed to create or update replicates.

6 Performance

We have hand-compiled the Onyx program discussed above, implemented it on an NCUBE hypercube multicomputer, and measured its performance. These measurements include separate timings for computation, packing and unpacking message buffers, and communication. Communication is further broken down into system primitives and user-mode message routing. (Our NCUBE is a first-generation machine with software routing, and we use the Caltech "crystal router" strategy [5] to avoid forwarding at the operating system level.)

As a standard of comparison, we also implemented a highly optimized uniprocessor version in the C language, and ran it on a single node of the NCUBE. This standard is as stringent as we could make it – "highly optimized" means coding loops with pointers and register variables, applying induction optimizations by hand, examining the object code to see what the compiler was doing, and refining the uniprocessor code until we could get no further improvement.[5]

Results are shown in Figure 6. There are several interesting features of these numbers.

- The Onyx executor performs the computation *faster* than the hand-optimized C code. This may be surprising, but it is easy to explain – pointers are faster than indexing. The relatively large difference (14%) is apparently a quirk of the NCUBE compiler/hardware combination. When we repeated this comparison with other compil-

[5] We did not, however, explicitly convert the nbr arrays from numeric indices to pointers. That is essentially what Onyx does, and there seemed little point in comparing an apple to itself.

	Optim-ized C	Onyx					
Processors, P_n	1	1	2	4	8	16	32
Times (1)							
Computation	0.435	0.373	0.1907	0.0955	0.0483	0.0246	0.0127
Message Packing	–	–	0.0029	0.0025	0.0030	0.0021	0.0024
Communications							
routing	–	–	0.0017	0.0027	0.0049	0.0038	0.0062
sys I/O (2)	–	–	0.0042	0.0053	0.0075	0.0096	0.0118
phy I/O (3)	–	–	0.0012	0.0009	0.0013	0.0007	0.0008
Total per iteration,							
T_n (4)	0.435	0.373	0.2011	0.1075	0.0653	0.0423	0.0363
Inspector Setup	–	0.577 (5)	2.212	1.233	0.932	0.595	0.541
Relative speed, S_n (6)	0.86	1	1.85	3.47	5.71	8.82	10.26
Efficiency, E_n (7)	–	100%	92%	86%	71%	55%	32%

Notes:

(1) Times for individual steps are reported as the maximum seconds used by any processor for that step.

(2) Time spent in system I/O primitives, including buffering and message startup, but excluding physical I/O time.

(3) Physical I/O time, estimated from transfer byte counts and published link speed.

(4) Total wall clock time per iteration. May not equal the sum of times for steps because of measurement uncertainty.

(5) This inspector was customized for standard array storage and no remote references.

(6) Relative speed is defined as $S_n = T_1/T_n$, where T_1 is the 1-processor Onyx time. See text for the reasons behind this definition.

(7) Efficiency is defined as $E_n = S_n/P_n$, for P_n processors and a relative speed of S_n.

Figure 6: Measured performance for the smoothing calculation on a small irregular mesh of 1854 nodes, average 5.7 neighbors/node.

ers and processors (Encore, Sequent, VAX, DECStation), the hand-optimized C was almost equal to Onyx. Because of these results, we have chosen to use the single-processor Onyx time as a reference in table 5. It is appealing to use the C time, since this would further improve the Onyx performance numbers, but we believe this would be misleading.

- The time spent in software routing is roughly equal to that spent in system-level communication primitives. Both of these are much higher than the data transfer time based on the link speed. This means that the Onyx implementation could greatly benefit from hardware routing and reduced message startup cost. It also means that there is very little to be gained from attempting to overlap communication and computation for this application.

- Setup cost for the inspector is in the range of 11-15 iterations of the computation, for all grain sizes tested. This time could undoubtedly be reduced, considering that little effort has been spent on optimizing the inspector. However, it is important to note that even the current setup time would be a small fraction of total run time in typical applications with hundreds or thousands of iterations. In addition, the calculation in our model program is lightweight – only two floating point operations per (potentially) remote data reference. The heavier calculations found in realistic applications would further reduce the relative setup cost.

We believe that these results also indicate that the Onyx implementation is competitive with a handwritten message passing code, for this simple model program. Our performance results indicate that the Onyx executor can compute over local data structures faster than the conventional indexing code that a human would write. Similarly, the executors produced for message packing, unpacking, and routing are also as good or better than a human would write, and there is no unnecessary data transfer or synchronization. For more complex applications, the Onyx approach remains to be tested. We have no data, for example, on how well the Onyx approach will detect opportunities to reuse data instead of transferring it, which is a common optimization in hand-written programs.

7 Related Work

The techniques discussed in this paper are most closely related to those of Mehrotra [8, 7]. Koelbel and Mehrotra [7] discuss using a regular blocked distribution to address problems like our model. However, that work does not identify the need for irregular (arbitrary) distributions, and the performance statistics it reports were obtained from a model problem with a rectangular grid pattern, where regular distributions (blocked or cyclic) are well known to be optimal. This paper extends their work by addressing distributions that are irregular and specified at runtime, and by introducing the concept of persistent replicates. It also differs in using a truly irregular pattern for evaluation and in comparing performance against the standard of a "best uniprocessor program".

Our address caching strategy is similar to that described by Crowley et.al. [4], who use it to implement a matrix solution iteration. We generalize that work by using the strategy to implement application-independent parallel access to distributed arrays. Our executors are also somewhat faster because all distribution checking code is moved to the inspector. (Crowley et.al. [4] report a 10% penalty, compared to conventional indexing; ours runs faster than conventional indexing.) However, we do not attempt to parallelize a computation expressed in serial form, which is Crowley's major point [4].

Our concepts of data distribution and replication, particularly the use of persistent replicates, also borrow much from Emerald [6], Bal's Orca [1], Munin [2], and other efforts designed to provide the functionality of shared data while running on distributed systems. However, our language model and implementation techniques are substantially different, and none of those systems have attempted to provide efficient execution of the sort of problem we have addressed here.

8 Summary and Conclusions

We have discussed the efficient implementation of a model program with irregular but repeating data reference patterns. This example requires an irregular partitioning, which is easily specified in the Onyx system in terms of moving data elements to specific processors. Onyx exploits the repetition by caching information from the first iteration of each pattern, in order to make subsequent iterations as fast as possible. The model program has been hand-compiled and implemented on an NCUBE multicomputer, and its performance has been measured.

The computational loops of the Onyx program were found to be as fast as hand-optimized uniprocessor code, and its communication requirements were as low as could be achieved by hand-coding. This suggests that the approach is well suited to programs in which the repetition count is relatively high and for which there is enough memory to allow caching its patterns.

Further work is required in several areas. For patterns which are regular or nearly so, such as grid algorithms on domains with irregular boundaries, it should be possible to greatly reduce cache sizes by compressing them, perhaps using linear sequence techniques. This approach could be especially effective in combination with compile-time replication, which would allow owned and replicated data to be placed in the natural addressing relationships of non-distributed arrays. More attention must be paid to the issue of how one detects repeating reference patterns in programs of realistic complexity. Finally, the entire approach must be evaluated against a wider range of applications, again using programs of realistic complexity.

References

[1] H. E. Bal. *The Shared Data-Object Model as a Paradigm for Programming Distributed*

Systems. PhD thesis, Vrije Univ. Amsterdam, October 1989.

[2] John Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. Technical Report COMP TR89-98, Rice Univ., November 1989.

[3] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2(2):151–170, Oct 1988.

[4] Kay Crowley, Joel Saltz, Ravi Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. Technical Report 89-7, ICASE, 1987.

[5] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors*. Prentice Hall, 1988.

[6] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Feb 1988.

[7] Charles Koelbel and Piyush Mehrotra. Supporting shared data structures on distributed memory architectures. Technical Report CSD-TR-915, Purdue Univ., October 1989.

[8] Piyush Mehrotra and John Van Rosendale. Compiling high level constructs to distributed memory architectures. Technical Report 89-20, ICASE, 1989.

[9] Michael J. Quinn and Philip J. Hatcher. Data parallel programming on multicomputers. Technical Report PCL-88-17, Parallel Computing Lab, Dept. of Comp.Sci., Univ. of New Hampshire, Durham, NH 03824, Oct 1988.

[10] Anne Rogers and Keshav Pingali. Process decomposition through locality of reference. In *Conference on Programming Language Design and Implementation*, pages 69–80, 1989.

[11] P. S. Tseng. *A Parallelizing Compiler for Distributed Memory Parallel Computers*. PhD thesis, Carnegie Mellon University, May 1989.

END

DATE FILMED

10/26/90