Theor Chim Acta (1993) 84: 363–375

Theoretica Chimica Acta

© Springer-Verlag 1993

# Moving beyond message passing. Experiments with a distributed-data model

**Robert J. Harrison**

Theoretical Chemistry Group, Chemistry Division, Argonne National Laboratory, Argonne, IL 60439, USA

Received October 1, 1991/Accepted April 7, 1992

**Summary.** A message-passing model provides a natural and efficient parallel implementation for many applications in chemical physics on MIMD machines. However, although the distinction between local and non-local memory is at the very heart of writing efficient parallel programs, message passing leaves all responsibility for data management to the applications. This has significant, detrimental implications for both ease of programming and efficient use of shared and distributed resources. Examined here is a simple model which increments message passing with Linda-like tools for the manipulation of distributed-data structures. This is applied to common algorithms in chemical physics.

**Key words:** Distributed-data model – Message passing

## 1 Introduction

Many algorithms in chemical physics are either parallel at a very coarse grain (e.g. Monte Carlo or SCF) or are readily addressed with either a static domain decomposition or task-driven systolic loop. In either case, message passing provides an efficient and *natural* implementation. All the distributed-memory multiple-instruction multiple-data (MIMD) applications described at this workshop adopted a message-passing model. The message-passing interface may have been vendor specific (i.e. Intel's NX [1]), or portable (e.g. Parasoft's Express [2]), TCGMSG [3], P4 [4], PICL [6], PVM [7]) or built into the language (e.g. occam [8, 9]), but they are all very similar in spirit. The essentially exclusive adoption of message passing is also reflected in the physical sciences parallel computing literature where the only common alternative adopted is loop-based parallelism, usually in a few-processor shared-memory environment [10].

However, there are many algorithms for which the derivation of a message-passing implementation is non-trivial, error-prone and far from "natural". Dynamic load balancing can be such a problem as this requires global access and updating of the pool/list/index of tasks remaining to be performed. This arises in quantum Monte Carlo simulations with a large spread in branching weights or the evaluation of *ab initio* two-electron integrals in systems with high symmetry and/or high angular momentum functions [13]. Both give rise to a large variation in the task size.

Another example is provided by an algorithm requiring asynchronous access (read or write) to the data within another process. This might occur in domain decompositions with exchange of boundary information or in accessing a shared "file" that is being buffered in distributed memory. A message-passing solution is straightforward only if the process with the data can anticipate when other processes will access data. Otherwise, each process must either explicitly poll for requests or implement fully-asynchronous event-driven mechanisms to allow the process to continue with useful work while transparently processing remote requests. The second solution is by far the most sophisticated and powerful, but is expensive to code on a case-by-case basis and requires that the programmer be knowledgeable of issues of which most physical scientists would probably rather, and quite reasonably so, remain unaware: e.g. deadlock, mutual-exclusion, re-entrancy, event-driven algorithms. Of these, only deadlock is of common concern in most message-passing programs. Also, many FORTRAN compilers are not capable of generating re-entrant code, requiring use of other languages.

In both of these examples the manipulation of shared/distributed data was the prime cause of the complexity. Consider also how these concerns are magnified on machines with many more processors (e.g. $O(10^4)$) than is common today (approx. $O(10^2)$). Few would claim to understand how to make effective use of the resources of such machines and a message-passing model, except by making the distinction between local and remote memory painfully explicit, does not contribute to an understanding of writing scalable applications.

There are possibly more parallel programming models, languages, environments, paradigms, etc. than computer scientists, and a selection of formal frameworks within which parallel programs may be specified independent of their implementation (good introductions to this literature are contained in [14–18] and in conference proceedings published in the various ACM journals). Little of this work has thus far been expressed in widely available, supported scientific programming environments (exceptions might be Linda [16, 19, 20], Strand [17], PCN [21, 22], loop level parallel FORTRAN/C compilers [14, 23–26]), but the concepts involved are of immediate utility. For the present, message passing remains the only parallel-programming environment that one can almost guarantee is both readily available and potentially efficient on any MIMD machine.

We adopt an incremental approach and add just what is needed to recover a concise and efficient expression of the class of algorithms discussed above. This acknowledges the fact that many applications contain a mix of algorithms, only some of which may require higher level tools. An incremental approach also permits applications already using message-passing environments to take immediate advantage of these tools, which will be integrated into our portable message-passing tool kit TCGMSG [3] as appropriate. In the process, we shall attempt to learn about how to take advantage of more of the parallelism in our applications.

## 2 Distributed data

### *2.1 Linda*

Linda [16, 19] is a memory model and a coordination language. Realizations of this model in C and FORTRAN are commercially available [20] and there are many directions of related current research [27]. Few details of Linda will be given here; see [19] for an introduction. The Linda memory is a tuple space, a tuple being a series of typed values. Tuples may be either passive or under evaluation (representing a thread of execution). C-Linda [19, 20] provides six operations on tuples, out(), in(), rd(), inp(), rdp() and eval() which also coordinate access to tuples. out() adds a tuple to tuple space, in() blocks until it finds a tuple that matches the specified pattern, returns the requested data and then removes the tuple from tuple space. rd() performs the same operation as in() except the tuple is left in tuple space, while inp() and rdp() are non-blocking forms which return true or false according to if the request was satisfied. eval() is similar to out() except that new processes are created to evaluate each field of the tuple. When all fields are evaluated, the tuple becomes passive.

As an example, consider creation of a tuple to simulate an element (number 199) in a distributed array which is subsequently read into the variable coeff, perhaps in another process:

```
double coeff;
out("CI vector", 199, 0.003);
rd("CI vector", 199, ? coeff);
```

Linda provides a powerful programming model of which we shall adopt only one aspect – the construction and manipulation of distributed-data structures (ref. [19] provides many examples of such data). For examples of other environments drawing on concepts from Linda see ref. [27].

## *2.2 A simple distributed-data model*

We seek to augment the successful static process message-passing model, with efficient tools to manipulate and coordinate access to shared and distributed data. The static process model implies we shall not use eval(). The applications we are considering need only simple distributed-data structures, e.g. scalar, arrays, sets of records. Current realizations of Linda [20, 27] fail to provide information on where or how tuples are stored or accessed. Indeed, with a requirement to match general tuples (partly at compile time, partly at runtime) this information is not necessarily readily available. This prevents development of accurate performance-models and can introduce inefficiencies in memory usage and communication. In particular, large applications need to be fully aware of local memory usage on typical-sized compute nodes (4–16 Mybtes). Since we have only a limited number of data types, we may explicitly declare the structure of the data, permitting the implementation to declare how and where this data is stored. We no longer have a tuple space, merely some shared data. Message passing would be extremely clumsy to coordinate access to the shared data, so we retain the coordination properties of the basic operations (out(), in(), rd(), inp(), rdp()) on the shared data.

Having thrown out so much of Linda, what is left?

• The existing message-passing interface (TCGMSG [3] or whatever is perferred).

• The basic shared/distributed-data structures that we care to support explicitly (to data: scalars; arrays; sets).

• All the distributed-data structures that can be built from the basic structures, which includes: linked lists; queues; trees.

• All the structures that arise from the coordination properties of the basic operations (e.g. semaphores, barriers, monitors).

A very rich environment! In fact, between the message passing and the distributed data, there is a great deal of redundancy in this environment. This is exactly what is needed for our present purpose, as we can freely experiment with both models in the same program.

Similar suggestions have been made in other environments related to Linda. Indexed objects (i.e. arrays) have been implemented in the object-oriented Interwork II Concurrent Programming Toolkit [28] with the goals of increasing efficiency and expressivity. The developers of the AUCC++Linda System [29] explicitly recognize tuple distribution as being critical to optimization and conclude that current hash-table implementations are inadequate. Optimizations proposed for the Mercury model [30] include user-defined data structures in tuple space to permit optimization of data distribution in a networked environment.

Before looking at a few examples we need to define the model more precisely. Presently, there are three basic shared-data structures:

**Scalar** An item which is kept by a single process specified when the data is declared, along with the size of the item in bytes (future implementations may support explicit typing of the components of an item to support heterogenous environments).

**Array** An indexed list of items. Presently, only a single index is supported and items are assumed to be of fixed size which is specified when the data is declared. Most important is that the placement of the data is also declared. This may be a linear distribution in chunks of size $K$, so that item $i$ is held by process $p = (i/K + origin) \bmod P$, $P$ being the number of processes, *origin* being an offset corresponding to the process holding element zero. It is intended also to support pseudo-random and user defined hash functions.

**Set** An unordered set of items (again, currently of fixed size) which may be distinguished only by contents. An out() stores the data locally if possible. Otherwise, it searches in order of increasing distance for a node that can hold more data. Similarly, an in() returns local data in preference to searching, again in order of increasing distance, for remote data.

The following attributes of the basic operations are also relevant:

• Memory is allocated when an item is created with out() and is freed when the item is destroyed with in(). The internal memory management routines (DDmalloc(), DDfree()) are callable by the application so available memory may be shared between application and tool kit.

• References to data that are stored locally incur only the cost of copying the data into the user space, plus overhead from memory management, simple indexing operations and procedure calls.

• References to remote scalar/array data cause a single message to be sent directly to the node holding the data which responds with a single message when the data becomes available.

• Multiple writes (out()'s) to a scalar or array item are resolved in an unspecified order.

• Fairness is guaranteed in multiple reads (in()'s) by queuing requests in FIFO order.

If not specified otherwise, basic operations are intended to behave exactly as the corresponding tuple operations of C-Linda.

## 3 Distributed data in action

First, let's convert the simple example of C-Linda syntax given above to the distributed-data model.

```
#include "ddata.h"
  double coeff;
  int      CiVector, DDeclare();
  CiVector = DDeclare("CI vector", ARRAY, LENGTH, sizeof(double),
                       CHUNKSIZE, 1, ORIGIN, 0);
  OUT(CiVector, 199, &0.003);
  RD(CiVector, 199, &coeff);
```

The array is declared with the call to DDeclare() which returns the integer handle used to access this data structure. This use of handles retains the ability to assign or swap entire distributed structures, without requiring either a preprocessor or any pattern matching in the tool kit. The OUT() and RD() operations are, in this instance, syntactically very similar to the C-Linda equivalents.

### *3.1 A barrier*

We are now back in a shared-memory environment with all the associated problems – mutual exclusion, race conditions, etc. Using message passing, process synchronization was trivial – simply exchange messages. Synchronizing processes via shared data is substantially more subtle, but fortunately is a problem of classical concurrent computing and has long been solved (for an introduction see ref. [31]). There are examples of simple barriers and semaphores using C-Linda in ref. [19]. The distributed-data tool kit provides a routine Barrier() which block until all processes rendezvous at that point. Following are the code fragments that implement the barrier. First, the internal shared variables must be declared and initialized (by process zero):

```
  long Barrier_count = DDeclare("Barrier Count", SCALAR,
                               LENGTH, sizeof(long), NODE, 0);
  long Barrier_nthru = DDeclare("Barrier Nthru", SCALAR,
                               LENGTH, sizeof(long), NODE, 0);
  long zero = 0;
  if(NODEID_() = = 0)
  OUT(Barrier_Count, &zero);
```

The actual barrier, which may be used repeatedly, is implemented by

```
 void Barrier()
 {
  long counter, nthru = 0, zero = 0;
  IN(Barrier_count, &counter);     /* Count processes at barrier */
  counter++;
  if (counter==NNODES_()) {       /* If everyone is here */
    nthru = 1;                     /* I'm the first thru */
  } else {
    OUT(Barrier_count, &counter);/* Output counter */
    IN(Barrier_nthru, &nthru);     /*Wait for nthru to be posted */
    nthru++;                       /*I'm thru now */
  }
  if (nthru < NNODES_())
    OUT(Barrier_nthru, &nthru);  /*Post no. thru so far */
  else
    OUT(Barrier_count, &zero)      /*Last thru resets the barrier */
}
```

The counting of those going through the barrier is required so that the last process through out()'s zero for the shared counter "Barrier Count", guaranteeing that other processes cannot race through subsequent invocations of Barrier(). This algorithm is simple but inefficient, scaling as the number of processes.

## *3.2 Load balancing in QMC*

The preceding example might give the impression that things have become worse, not better! Let's now examine a simple and effective approach to load balancing an application that is iteratively processing a distributed set of records which take a widely varying amount of time to process. A good example of this, quantum Monte Carlo with heavy branching, was introduced above. The simple message-passing solution of using a single master and multiple slave model does not scale to massively-parallel machines as the master becomes a bottleneck. One can adopt a hierarchy of masters but the program becomes complex. It is far better to distribute completely the administrative responsibilities along with the data. The following code shows the structure of a load-balanced QMC kernel in pseudo-C using the distributed-data tools.

```
struct Psip {                                          L1
  int generation;
  int weight;
  ⟨...⟩
} psip;
int Old = DDeclare("Old Psips", SET,                   L2
                   LENGTH, sizeof(psip));
int New = DDeclare("New Psips", SET,
                    LENGTH, sizeof(psip));
Initialize(Old);
while (nblocks--) {
  while (INP(Old, &psip)) {                            L3
    while (psip.generation < limit && psip.weight)     {
      AdvancePsip(&psip);                              L4
```

```
        while (psip.weight > 1) {
          OUT(Old, &psip);                        L5
          psip.weight--;
          }
        }
        if (psip.weight) OUT(New, &psip);         L6
  }
  temp = New; New = Old; Old = temp;              L7
  DGOP_(&type, Averages, &N_Averages, "+", scratch); L8
}
```

The structure psip (L1) contains all the information about a single psip (age, multiplicity, coordinates, energy, etc.). Two distributed sets of psips are declared (L2), corresponding to old and new lists, and the contents of the old list are initialized. The code then loops through several sampling blocks. For each block it withdraws an element from the old list (L3), if one is available, and moves it (L4) for the requisite number of generations, accumulating averages, etc. If random branching requires creation of additional copies, this is performed (L5) by putting copies back onto the old list. If a psip makes it all the way to the end of a block without being absorbed, it is put on the new list (L6) and the next old psip is taken (L3). When all the old psips have been processed, the old and the new lists are exchanged (L7) and each process's contributions to the global averages are accumulated (L8). This last step uses a global reduction operation from the message-passing tool kit TCGMSG [3], though it could have been done less efficiently using shared data.

The above implementation transparently performs load balancing because the psips are stored in the globally accessible, distributed data structures. The properties of the operations INP() and OUT() specified above also imply that close to the minimum of message passing is used in doing this. No message passing is performed unless either no psips are available locally, or there is no room to store a brachned/new psip. In both instances, requests are sent to nodes in order of increasing distance, minimizing contention. Excess population from one node "diffuses" to nodes with smaller populations.

### *3.3 A distributed file buffer*

In addition to a poor software environment, most highly-parallel computers have remarkably inadequate I/O capabilities (cf. the discussion on the Touchstone Delta in the paper by Kendall et al. in these proceedings). As such it is almost essential to buffer a file in memory as far as possible, a task that is relatively complex with simple message-passing tools. The following (untested) FORTRAN routine implements a simple buffering algorithm for a shared, record addressable file (fixed length records) where each node has an independent file pointer.

```
      subroutine ReadRecord(unit, index, record)
      include 'ddata.h'
      parameter (nrec = 131072, lenrec = 8192)
      integer unit, index, FileBuffer, DDeclare
      logical INP, RDP
      byte record(lenrec)
      save FileBuffer
```

```
c
      if (index.le.nrec) then
         if (.not. RDP(FileBuffer, index-1, record)) then
            read(unit,rec = index) record
            call OUT(FileBuffer, index-1, record)
         endif
      else
         read(unit,rec = index) record
      endif
      return
c
      entry WriteRecord(unit, index, record)
      if (index.le.nrec) then
         call OUT(FileBuffer, index-1, record)
      else
         write(unit,rec = index, record)
      endif
      return
c
      entry FlushBuffer(unit, record)
      do index = 1, nrec
         if (INP(FileBuffer, index-1, record))
     $      write(unit, rec = index) record
      enddo
      return
c
      entry DeclareBuffer()
      FileBuffer = DDeclare('File buffer', ARRAY,
     $                      CHUNKSIZE, nrec/numnodes(),
     $                      LENGTH, lenrec, ORIGIN, 0);
      end
```

The first 1-nrec records of the file are buffered in memory. The remainder are read from disc as required. The code is simple enough to warrant no explanation. The FORTRAN unit is opened and closed as usual. The application is unaware of the buffering, except through increased performance, and the requirement that FlushBuffer() must be called before closing the file. The values for nrec and lenrec correspond to buffering the first giga-byte of the file – a reasonable thing to do on the full Touchstone Delta, corresponding to dedicating 1/8th of physical memory to the file. The manner in which the file buffer is best distributed is application specific. One useful model is for each node to hold a block of consecutive records, so that it is straightforward for a process to access only the records local to it, while still retaining ready acess to the other sections. The entry point DeclareBuffer() realizes this[1].

[1] In practice calls to DDeclare() are isolated in an automatically called initialization routine, ensuring global definition before use

## 4 Performance

The distributed-data tools described above are only experimental but their performance is an important concern. Following are the first two such tests to be run, both highlighting successes and failures of the curent version.

### *4.1 Latency – passing a message round a ring*

The time to pass a message of varying length round a ring of varying size was measured on the iPSC/i860 using message passing (NX [1]) and the distributed-data interface. A simple performance model for the elapsed time ($t$) is:

$$t = P(\tau_0 + n\tau_1) \tag{1}$$

where $P$ is the number of processes in the ring, $\tau_0$ is the overhead, $\tau_1$ is the transmission time per byte and $n$ is the length of the message in bytes. The raw message-passing data fits to the values $\tau_0 = 78-85\ \mu$s and $\tau_1 = 0.362\ \mu$s/byte. The corresponding values through the distributed-data interface are $\tau_0 = 150-160\ \mu$s and $\tau_1 = 0.395\ \mu$s/byte. The latency is consistent with two messages being passed for every remote data access (one for the request, one for the response). The asymptotic distributed-data transfer rate is very slightly lower than that of raw message passing due to the cost of copying the buffer in the OUT() operation (copying the buffer runs at approx. 30 Mb/sec, thus the total time should be $1/30 + 0.362 = 0.395\ \mu$s/byte).

On two processors, the distributed-data times do not fit the simple performance model. The cause for this is not fully resolved, but is thought to be associated with time spent in memory management, etc., *after* having responded to a remote request. With more than two nodes the above benchmark does not measure this work, which is overlapped with the message being sent between other nodes.

### *4.2 Load balancing*

The QMC example above was simulated with the AdvancePsip() routine performing 20 double precision square roots, incrementing the generation counter and recomputing the weight. The weight was randomly set to 0.5 with a probability of 18/19 and to 10 otherwise. The weight was converted to an integer by the usual trick of adding a uniform random number in (0, 1) and truncating. The size of a psip was padded to 128 double-precision words and an initial population of 10,000 psips was used with 100 generations per block. To make the total work independent of the number of processors, a distinct random number generator seed was carried along with each psip. The seed was incremented with a fixed large number upon branching, providing approximate independence of the copy from the original. However, the randomness inherent in ordering of accesses to remote data implies that successive runs with the same number of processors will still exhibit different distributions of work.

Figures 1, 2, and 3 display results from the iPSC/i860 for the first five blocks of the simulation on varying numbers of processors. The entire population is initially placed on a single processor, the worst case which also corresponds to a single-master multiple-worker model. The most direct measure of load-balance is

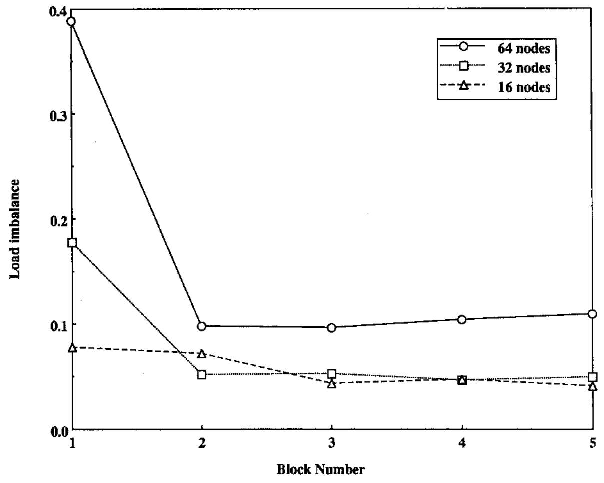

**Fig. 1.** Model QMC work load imbalance versus block number for 16, 32 and 64 processes on an iPSC-i860. The imbalance is measured by the ratio of the difference and sum of the maximum and minimum work done by any node. An imbalance of 0.0 indicates perfect balance. A value of 1.0 indicates at least one process did no work

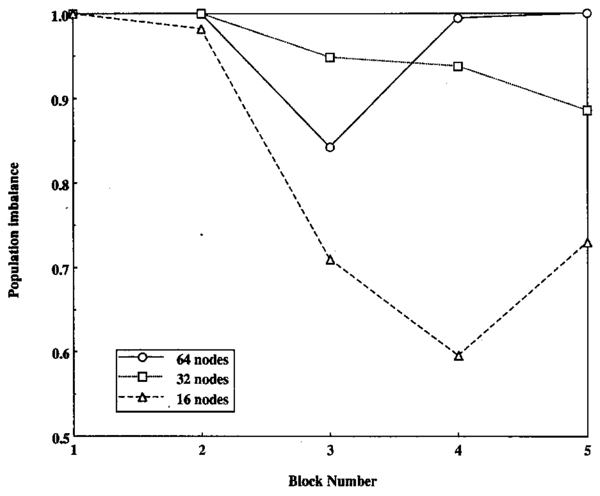

**Fig. 2.** Model QMC population imbalance versus block number for 16, 32 and 64 processes on an iPSC-i860. The imbalance is measured by the ratio of the difference and sum of the maximum and minimum population on any node. An imbalance of 0.0 indicates all populations are equal. A value of 1.0 indicates at least one process has no psips

given by the range of the number of moves performed by each node (Fig. 1). After two blocks on 64 nodes the work imbalance is approximately 10%, even though the population imbalance (Fig. 2) is always greater than 60% due to expected statistical fluctuations. Increasing the total population would improve the efficiency further, by smoothing out fluctuations.

Further analysis indicates that the time taken to determine that there are no old psips anywhere in the machine is inefficient. This is currently done with a naive point-to-point algorithm with expense linear in the number of processors. A tree-based algorithm results in logarithmic expense and is being implemented. Figure 3 displays estimated speedup with and without this overhead (the performance on one processor is estimated from that on four processors as the calculation will not fit in available memory). With the overhead removed, the 64-processor calculation exhibits a speedup of 7.17 relative to 8 processors. However, the 8-processor time shows a super-linear speed-up of 2.07 relative to 4 processors. This is possibly due to a statistical fluctuation in load-balancing,

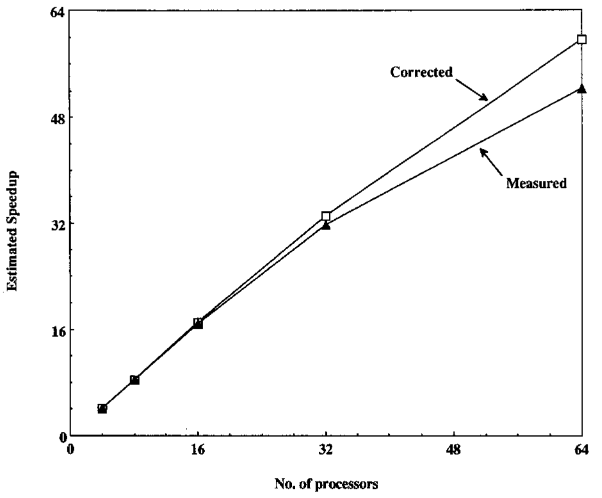

Fig. 3. Model QMC estimated speedup on an iPSC-i860. The curve labelled measured represents raw data. The corrected curve differs by subtraction of time taken to determine that no old psips remain, as discussed in the text

but could also be due to increasing overhead due to management of the memory free list with fewer processors and thus more items per node. More analysis is needed before a detailed evaluation of this benchmark is possible.

## 5 Conclusions

The distributed-data programming environment presented above preserves much of the raw message passing:

• Message passing can be simulated efficiently through the distributed-data interface. The current environment also allows the message-passing tools to be called directly if the extra latency is intolerable.

• The programmer retains the vital ability to determine placement of the application's data.

• The underlying machine model is still that of a distributed-memory machine – remote data is more expensive to access than local data.

The last two points are particularly important because without these it is not possible to model program performance accurately [14, 32].

The distributed-data environment provides several significant improvements over raw message passing that result both from its relationship to Linda and the restriction of the Linda memory model to specific data structures with known distribution.

• An "uncoupled programming style" [16] – a process need not even be aware of another process to provide data or resources for that process.

• The synchronization of processes resulting from access to distributed data is minimal in the sense that write operations do not block and read operations block automatically until data is made available.

• A uniform interface to distributed data is provided while still retaining full information on locality which may optionally be used to improve efficiency. The entire memory of the machine is now a resource that can be effectively exploited.

• In shared-memory environments, use of distributed data is substantially more efficient than message passing.

The first two points support the observation that it so far seems much easier to write correct efficient programs using the distributed-data interface than with message passing. The distributed-data model provides a very "natural" framework for writing a much broader class of applications than message passing. It exposes more of the machine resources to the application, and more of the application to the environment for increased optimization.

There are also some negative aspects:

• Coordinating processes through shared data is much harder than with message passing (cf. the implementation of Barrier()).

• It is not clear how portable this model is. Can the assertions about data placement, which equate to access cost, and the event-driven approach be supported on all interesting platforms?

• Aspects of the current implementation are deficient. For instance, an INP() operation on an empty set takes time $O(P)$ to complete, rather than $O(\log P)$.

• Debugging tools are non-existent.

Finally, independent forrays, such as this, into programming models are both educational and enjoyable. However, a portable, quality environment to support massively-parallel scientific applications can only result from a significant team effort. More collaboration is needed between physical scientist and computer scientist in the specification of such an environment.

*Acknowledgments.* This work was supported by the U.S. Department of Energy, Office of Basic Energy Sciences, Division of Chemical Sciences, under contract W-31-109-ENG-38. I thank R. Littlefield, E. Lusk and R. Stevens for many interesting discussions on parallel computing and R. Shepard and J.-K. Fisher for reading of this manuscript.

## References

1. iPSC/2 Users Guide (1988) Intel Corporation
2. Express is a product of ParaSoft, Mission Viejo, CA
3. Harrison RJ (1991) Int J Quant Chem in press
4. p4 are a set of portable message passing routines being distributed by Ewing Lusk of the Math and Comp Sci division at Argonne. They are the current version of the tools described in Ref. [5]
5. Boyle J, Butler R, Disz T, Glickfeld B, Lusk E, Overbeek R, Patterson J, Stevens R (1987) Portable programs for parallel processors, Holt, Rinehart, Winston, NY
6. Geist GA, Heath MT, Peyton BW, Worley PH (1990) Oak Ridge Natl Laboratory Tech Report TM-11616:1
7. Beguelin A, Dongarra J, Geist A, Manchek R, Sunderam V (1991) Oak Ridge Natl Laboratory Tech Report TM-11826:1
8. May D (1983) ACM SIGPLAN Notices 18:69
9. Bowler KC, Kenway RD, Pawley GS, Roweth D (1984) Occam 2 Programming Language, Prentice-Hall, Englewood Cliffs, NJ
10. The IBM LCAP project provides a counter example with use of loop and process level parallel extensions to FORTRAN on their mixed shared and local memory architecture. See Refs. [11, 12] and references therein
11. Watts JE, Dupuis M, Villar HO (August 29, 1986) IBM Tech Rep KGN-78:1

12. Dupuis M, Watts JD (1987) Theor Chim Acta 71:91
13. Harrison RJ, Kendall RA (1991) Theor Chim Acta 79:337
14. Almasi GS, Gottlieb A (1989) Highly parallel computing, Benjamin/Cummings, Redwood City, CA
15. Andrews GR (1991) Concurrent Programming: principles and practice, Benjamin/Cummings, Redwood City, CA
16. Carriero N, Gelernter D (1989) Communications of the ACM 32:444
17. Foster I, Taylor S (1990) Strand, new concepts in parallel programming. Prentice-Hall, NJ
18. Chandy KM, Misra J (1988) Parallel program design, A foundation. Addison-Wesley, Reading, MA
19. Carriero N, Gelernter D (1990) How to write parallel programs. A first course. MIT Press, Cambridge, MA
20. C-Linda is distributed commercially by Scientific Computing Associates, New Haven, Connecticut
21. Chandy KM, Taylor S, Kesselman C, Foster I (February 1990) Caltech Comp. Sci. Tech. Report CS-TR-90-03:1
22. Foster I, Taylor S (January 1990) Argonne National Laboratory Report ANL/MCS-TM-137:1
23. Several tools (e.g. VAST, MIMDIZER, FORGE) are developed and distributed by Pacific Sierra, Palcerville, CA. See the article by J. Levesque in these proceedings.
24. Kuck DJ, Davidson ES, Lawrie DH, Sameh AH (1987) in: Dongarra J (ed) Experimental parallel computing architectures. Elsevier, North-Holland, Amsterdam, p 1
25. Guzzi M, Padua D, Hoeflinger J, Lawrie D (1990) J Supercomputing 3:37
26. Cooper KD, Kennedy K, Torczon L (1986) ACM SICPLAN Notices 21:58
27. Wilson G (ed) (1991) Linda-like systems and their implementation. Edinburgh Parallel Computing Centre Techn Rep 91-13
28. Bain WL (1989) ACM SIGPLAN Notices 24:95
29. Callsen CJ, Cheng I, Hagen PL, p 39 in [27].
30. Schoinas G, pp. 105 in [27]
31. Ben-Ari M (1982) Principles of concurrent programming. Prentice-Hall, Englewood Cliffs, NJ
32. Anderson RJ, Synder L (1991) Proc IEEE 79:480