The P4 Parallel Programming System, the Linda Environment, and Some Experiences with Parallel Computation

ALLAN R. LARRABEE

Advanced Systems Laboratory, Research and Technology, Boeing Computer Services, Seattle, WA 98124-0346

ABSTRACT

The first digital computers consisted of a single processor acting on a single stream of data. In this so-called "von Neumann" architecture, computation speed is limited mainly by the time required to transfer data between the processor and memory. This limiting factor has been referred to as the "von Neumann bottleneck." The concern that the miniaturization of silicon-based integrated circuits will soon reach theoretical limits of size and gate times has led to increased interest in parallel architectures and also spurred research into alternatives to silicon-based implementations of processors. Meanwhile, sequential processors continue to be produced that have increased clock rates and an increase in memory locally available to a processor, and an increase in the rate at which data can be transferred to and from memories, networks, and remote storage. The efficiency of compilers and operating systems is also improving over time. Although such characteristics limit maximum performance, a large improvement in the speed of scientific computations can often be achieved by utilizing more efficient algorithms, particularly those that support parallel computation. This work discusses experiences with two tools for large grain (or "macro task") parallelism.

1 INTRODUCTION

Any computer that is not constrained to a single stream of instructions on a single stream of data can be characterized as a parallel computer. However, there are several levels or "granularities" of parallelism that are usable in today's parallel computers:

1. Pipelining—a fine grained form of parallelism in which a stream of data is processed by an assembly line of instructions. After the assembly line or pipe is full, one result emerges from the pipe for every new piece of data that starts down the pipe.
2. Array processing—a collection of processing units under one control, all execute the same instruction in parallel on different elements of data stored in separate memory.
3. MIMD (multiple instruction multiple data)—separate units operate independently on different data sets and communicate via common memory, message passing, or over a communication network.

Present day compilers can generate efficient code for vector (Type 1) and array processor (Type 2) machines. We are concerned here pri-

Received March 1992
Revised June 1993

Scientific Programming, Vol. 2, pp. 23–35 (1993)
CCC 1058-9244/94/030023-13

marily with the problems facing a scientific programmer who wishes to decrease a lengthy runtime by utilizing large grain MIMD-style concurrency. Compilers for the automatic generation of a code that takes advantage of multiple processors of this type are currently in an immature stage of development.

Whatever the resources available, a user can always wait for faster hardware or, alternatively, utilize existing machines (e.g., a cluster of individual workstations or a machine with multiple processors) to decrease runtime. However, unless a code is to be run repeatedly, it is probably not worth the effort to produce a parallel version given the current primitive state of tools to assist in this process and burden of proving correctness.

One purpose of this paper is to give scientific programmers some guidance to help them decide whether the speedup that can be obtained is worth the effort required to achieve that speedup. The speedup of a parallel program that is obtainable is related to the fraction of the program that is inherently scalar. This is formalized in Amdahl's Law [1] that states. . . .

$$\text{Overall speedup} = 1/[(1 - P) + (P/S)]$$

where P is the fraction of the original computation time that can be enhanced by parallel execution and S is the ratio of the time required for execution of the P section of the code to the time required for execution of the P section of the code after parallelization. If 40% of a particular application requires scalar execution and it is possible to achieve a speedup of 25 for execution of the remaining 60%, then the overall speedup would be $1/[(1 - .6) + (.6/25)]$ or an overall speedup of "only" about 2.36. Amdahl's Law emphasizes the fact that programmers desiring speedups via parallelization can be severely limited for certain algorithms.

Two approaches for parallelization will be discussed. One is a supported, commercial product [2] that involves a new way of thinking about programming (via so-called coordination languages). The other is in the public domain [3] and supports a message-passing style of parallel programming. Any programmer just entering the field of parallel computation is faced with a myriad of (mostly nonstandard) computation environments, libraries, precompilers, languages, etc. The two approaches discussed in this paper do not illustrate all of the programming issues that are involved. A recent workshop considered these two and many others [4].

2 ALGORITHMS, LANGUAGES, AND EXISTING CODES

An improved algorithm can result in a greater than linear speedup, whereas a 10 processor machine will be limited to a maximum speedup of "only" 10. An extreme example is the calculation of the global minimum energy that a protein molecule can achieve as a function of its bond angles in three-dimensional space. The best current algorithms have runtimes that are of order n**2, where n is the number of atoms in the protein. At present it is possible to solve structures (i.e., calculate tertiary structure de novo, not molecular dynamics) that have under 100 atoms if one is willing to invest many hours of time on a very fast computer. Although there are molecules of this size whose solution is very useful, it is common for a protein to have over 2,000 atoms and these molecules could not be rigorously solved in a decade of compute time on presently available machines. In nature, the largest proteins are able to find their global energy minimum, starting from a completely extended form, in a few seconds. Obviously, the global search algorithm mentioned here is not the best one possible. Thus, algorithm improvements represent the area where the greatest decreases in compute time can be realized.

Even an "inefficient" algorithm like the chemistry example given above can be adapted to benefit from parallel execution. In other cases, imperative languages like Fortran and C seem inadequate. The human computer, i.e. the human brain, has a clock rate of ca. 200 Hz (a neuron may send out 200 impulses per second with a peak performance of ca. 500 per second in special cases) yet can outperform the best traditional computers extant in such areas as pattern recognition. The recent work on neural net algorithms [5], which adapt readily to some parallel architectures, represents an attempt to more closely approach the mechanism and speed of nature's parallel "biocomputers" that have been under development for millions of years.

Many of the production programs presently in use represent algorithms that have been instantiated in Fortran over a period of years. These programs have been modified and improved and much effort has been expended on testing and debugging the code, if not completing its docu-

mentation. For many applications, however, most of the decrease in execution times achieved since initial code development has been the result of hardware improvements. Today the owners of such codes frequently are reluctant to extensively rewrite such "trusted" codes so as to take advantage of parallel execution, particularly if the gains are relatively small or portability may be lost. Another reason for this inertia is the lack of a clear migration path for parallel codes—scientists feel, with some justification, that if they were to convert their codes for parallelism, that form of the code would very likely have to be abandoned in the near future in order to take advantage of a newer parallel architecture or programming paradigm.

Before the advent of parallel architectures, the development of parallel algorithms offered no advantages. Fortran does allow code to be written in a somewhat object-oriented style, which can ease the allocation of work to different processors, but most of today's "dusty deck" Fortran codes were not constructed in this manner. Neither were they written using a message-passing paradigm, which maps naturally for the distributed-style computing discussed below. Unfortunately, Fortran possesses several features that create difficulties when converting codes to take advantage of parallelizing preprocessors, dependency analyzers, and compilers that search for structures to rewrite for concurrency. The use of GO TO, SAVE, EQUIVALENCE, ENTRY statements, Hollerith fields, and certain mixed typing definitions are all permitted in Fortran, and although they may be less utilized today, when present can limit the effectiveness of the evaluation by modern parallel analysis tools. Although many UNIX programmers feel that the language C may have fewer of such problems, current versions of C have their own difficulties [6]. These problems are part of the driving force for the creation of new languages more suitable for parallelization.

Parallel machines that require one specific language have already been offered commercially. For example, the FPS T series parallel processor [7] when first delivered supported only the Occam language. Such offerings are at a disadvantage because there are not a lot of preexisting codes and, in addition, the possibility of porting to other machines is severely limited.

3 COMPILER IMPROVEMENTS

Several compilers are available that can detect parallelism at the loop level and distribute loop iterations among a machine's multiple processors. Compilers will not allow the distribution if a temporal or runtime dependency is detected, but usually allow programmers to override, at their own risk, and force the compiler to distribute the iterations of a particular loop. For example, a problem arises when a subroutine is called from within a loop. Most compilers at this time cannot guarantee that such calls would not violate a data dependency or result in several processes contending for the same variable. In some cases, such determinations may only be possible at runtime.

Tools are emerging that can analyze codes to determine the data dependency, such as VecPar_77 [8] (a commercial product), Forge [9] (a commercial product), and Toolpack [10] (nonsupported), which interact with the programmer to detect opportunities to vectorize and/or parallelize Fortran. "Express" is a commercial package that offers a complete analysis, programming, performance monitoring, and debugging environment [11]. The interactive capability allows the added benefit of human input regarding variables and dependencies. It also has the effect of shifting responsibility for correctness to the user. No tool currently available is able to modify the underlying algorithm, but rather renders the current code more readable and clarifies dependencies.

The allocation of loop iterations and any new process creation does incur a certain amount of overhead and it is not rare for the compiler, using default parameters, to produce parallel code that executes more slowly than the scalar version. The programmer is usually allowed the option of allowing or restricting the automatic parallelization of any particular loop and can set criteria (e.g., which loops to parallelize, amount of unrolling, roundoff restrictions, the maximum and minimum number of iterations assigned to each processor, local inlining of subroutines, etc.) for maximum performance. A Fortran parallelizing precompiler named "KAP" is available commercially [12] for the Sequent Symmetry, Digital VMS and Ultrix machines, and Sun. Compilation time is increased as a result of the additional analysis these new compilers must accomplish. Most owners of scientific codes readily accept such increases as a requirement to obtain faster runtimes. Optimizing loop iterations can improve runtime performance, but it is not often that this will even halve the runtime of large scientific programs. This is about as far as programmers can presently go if they do not wish to invest the time to rewrite the entire code.

```
#define   MAX_SLAVES  15

real_main()
{
   int     slaves = 0, nrecs = 0, section, me;
   double  tpi = 0.0, total = 0.0, pi, h;
   double  partial_pi();

   while (nrecs < MAX_SLAVES)
      {
         printf("\nNumber of rectangles is ...");
         scanf("%d", &nrecs);
         if (nrecs < 0) exit();
      }
   while ((slaves < 1) || (slaves > MAX_SLAVES))
      {
         printf("\n\nNumber of processors wanted is ...");
         scanf("%d", &slaves);
      }
   start_timer();
   section = nrecs/slaves;
   h       = 1.0 / nrecs;

   for (me = 0; me <= slaves - 1; me++)
      eval("results", partial_pi(me, h, section, slaves, nrecs));

   for (me = 0; me <= slaves - 1; me++)
      {
         in("results", ?tpi);
          /* ? is a wild card */
         total = total + tpi;
      }
   timer_split();
   printf("\nCalculated pi is ...
    %27.25f\n", total);
   print_times();
}

double partial_pi(me, h, section, slaves, nrecs)

int    me, section, slaves, nrecs;
double h;
{
   int    lower, upper, j;
   double x, sum = 0.0;

   lower = (me * section) + 1;
   upper = lower + section - 1;
   if (me == (slaves - 1))
      upper = nrecs;
   for (j = lower; j <= upper; ++j)
      {
         x = (j - 0.5) * h;
```

```
52              sum = sum + (4.0/(1.0 + x * x));
53          }
54      return (h * sum);
55  }
```

4 TWO APPROACHES TO CONCURRENCY

A comprehensive review of all the tools available for creating new parallel code is beyond the scope of this work. Linda [2] is a programming environment based on the C language (a Fortran-based version is now available). P4 [3] is a programming environment based on a message-passing paradigm with either C or Fortran. Neither tool makes any attempt to maximize the throughput or load balancing of the machine, but, rather, takes the position that machine resources cannot be managed from the user language level. Linda is a commercial product, whereas P4 is freely available public domain software. Neither example shown here will show all the attributes available to the user, but both examples can be executed on a variety of distributed memory and shared memory architectures without changing any of the code.

4.1 Linda

The Linda coordination language [13] is hardware independent and has been implemented on a variety of architectures (Intel, Encore, Sequent, Cray YMP, Sun, and IBM RS6000). Linda can be viewed as a precompiler that adds only six new primitives to the C language. Only the command line initiation differs between environments. Each Linda program creates an associative shared memory referred to as a "tuple space." Tuples are ordered collections of fields (one tuple may have mixed types) that can have any of the types associated with the underlying language and can be any length. The tuple space is a collection of tuples that can be accessed by any process on any machine in a distributed network.

The six new functions are operations on tuples and the tuple space. The command "out" places a named tuple in the tuple space and "in" removes it. The "rd" command tries to find a tuple in the space that matches the query tuple. The latter can have wildcard fields that match any field of the same type. Thus the command

```
rd(1, 2, "foo", ?x, ?y)
```

tries to find a tuple in the global tuple space where the first field is a 1, the second is a 2, the third field is the string "foo", and the fourth and fifth fields are integers (assuming x and y were initially types as integers). The matching tuple (or the first match, if several tuples match) is copied to the program and x and y are given the values of the corresponding fields. The "in" and "rd" commands have variants that try to locate a matching tuple and return a 0 if they fail, or a 1 if they succeed, that is blocking and nonblocking variations of in and rd.

The command "eval" is similar to "out" but if a field is a function, a new process is started immediately, on a separate processor, and the function is evaluated before the tuple is made available for access. Once created, a tuple exists even after the process that created it has exited. Process initiation and termination are invisible to the programmer. Tuples can be accessed similar to those in a data base. A simple but computationally intense program that calculates the value of pi will serve to illustrate the use of Linda tuples to coordinate parallel processing. The program [14] produces an approximation to the value of pi by using the rectangle rule to compute successive approximations to

(formula) integral from 0 to 1 of 4/(1 + (x * x))

The C-Linda code (interspersed with commentary) for this problem is given on page 26.

Because this code is intended to execute both on a network of workstations and on shared memory architectures, the constant MAX_SLAVES on Line 1 could refer either to the maximum number of processors or the number of workstations. All Linda programs use `real_main()` rather than `main()` and the file must end with a suffix of `.cl` rather than `.c`. Line 5 defines an integer "me" that will have a unique value for each spawned process evaluating the function `partial_pi()`. Thus, each process will calculate part of the integration and all contributions will be summed by `real_main()`.

Lines 9 through 19 allow the user to input the number of rectangles and the number of processes. Linda will initiate each new process on a separate processor if enough are available, otherwise the processes are queued. MAX_SLAVES is used in Line 9 to prevent the user from having a value of "nrecs" less than the number of "slaves". If it were less, a slave would have no work to do and such an input is not expected. The user provides a list of machines available (potential slaves) in a separate file.

Linda provides a timing function that is initialized at Line 20. The variable "section" is the number, or very close to the number, of rectangles for each process. Each rectangle has a thickness of "h".

Each iteration of the loop at Lines 24 and 25 spawns a new process and each new process (the number of new processes will be equal to the value of "slaves") will get a list of arguments. All of the arguments are the same for each process, except for the integer "me", the loop iteration, which will be used to set the integration limits for each process. The `partial_pi()` function will return a variable that has been typed as a "double" in Line 7. At the completion of all the spawned processes, the tuple space will have "slaves" number of tuples each one of the form

("results", instantiated variable of type "double")

The string "results" is not required, but such strings can be used for clarity and serve to decrease the time required for searches of very large numbers of tuples. The `real_main()` program does not wait for the spawned process to complete, but rather immediately starts execution of the next loop iteration.

There are "slaves" number of results to be collected. The function `in()` is blocking, so the program counter for `real_main()` will remain at Line 29 until one process exits and has created an appropriate tuple (see Line 54 below). The ? in Line 29 is a wild card, therefore, any tuple will be removed (not just read) from the tuple space where the first field is a string "results" and the second field is a double precision number (no tuple would be input if the variable `tpi` has been typed in Line 6 as an integer or a float). Each successive contribution from each process is collected at Line 30 and after "slaves" number of tuples are input (the order of collection is not important) the internal time clock is terminated at Line 32 and the elapsed time is printed at Line 34.

Lines 37 to 55 show the function that each of the "slave" processes evaluates. Each process has a different value for "me" from 0 to `slaves - 1` and in Lines 45 to 48, the upper and lower limits of integration are set. Lines 47 to 48 are required for the case in which the division of "nrecs" by "slaves" leaves a remainder. If so, then one process (the one with me equal to `slaves - 1`) will do slightly more work than the remaining ones. The contribution to the total integration is in Lines 49 to 53.

Linda is a language where parallel processes are easily initiated and synchronized. Early versions of both runtime and postmortem debuggers are available, and a trace facility exists that allows tuple traffic to be monitored during execution, along with deadlock detection. The resulting language is very powerful and the parallel computing model is easily understood, but Linda does involve learning a new style of programming. One company [15] offers a UNIX clone operating system (product name of HARNESS) for workstations based on a simplified version of Linda.

One problem with the current implementation of Linda arises when the tuple space is composed of many large tuples. The network Linda runtime system does not maintain a copy of all tuples on every machine, rather it distributes the tuples across the network in a manner not controllable by the programmer. A large overhead can be incurred when one workstation needs to access tuple(s) on a distant machine. The programmer could possibly try to assign the various tuples to particular machines in order to lessen this overhead, but at the present time Linda does not have this capability. A language has been devised to circumvent some of these problems [16]. However, it is not generally available to the programming community for evaluation at this time.

4.2 Parallel Programming System (P4)

P4 is essentially a point-to-point message-passing library in which the programmer need not be concerned with varying definitions of basic data types across vendors (i.e., Alliant, Intel, Sun, Cray XMP, NeXT, DEC, Silicon Graphics, Encore Multimax, Sequent Symmetry, IBM RS6000, Stardent, BBN). The message-passing functions of P4 can utilize if necessary the xdr software package [17] to perform data conversion (of basic types and arrays, but not user-defined structures)

```
 1  #include "p4.h"
 2  #define  MAX_SLAVES  15
 3
 4  main(argc, argv)
 5  int    argc;
 6  char **argv;
 7  {
 8    int    section, slaves, type, from, size, nrecs = 0, me, i;
 9    int    start, stop;
10    double *partial_pi, total = 0.0, h;
11    struct slave_data
12      {
13        int   nrecs;
14        int   section;
15        float h;
16      } data;
17    p4_initenv(&argc,argv);
18    p4_create_procgroup();
19    while (nrecs < MAX_SLAVES)
20       {
21          printf("\nNumber of rectangles is ...");
22          scanf("%d", &nrecs);
23          if (nrecs <= 0) goto end;
24       }
25    while ((slaves < 1) || (slaves > MAX_SLAVES))
26       {
27          printf("\n\nNumber of processors wanted is ...");
28          scanf("%d", &slaves);
29       }
30    start = p4_clock();
31    data.nrecs   = nrecs;
32    data.section = nrecs/slaves;
33    data.h       = 1.0/nrecs;
34    for (i = 1; i <= slaves; i++)
35      p4_send(100, i, &data, sizeof(data));
36    type 200;
37    partial_pi = NULL;
38    for (i = 1;i <= slaves; i++)
39      {
40        from = -1;
41        p4_recv(&type, &from, &partial_pi, &size);
42        total = total + *partial_pi;
43      }
44    end:
45    stop = p4_clock();
46    printf("\nCalculated pi is ... %27.25f\n", total);
47    printf("Time is %d msecs\n", stop - start);
48    p4_wait_for_end();
49  }
```

---

if the machines involved in a network have different data representations. In addition, P4 has functions for shared memory architectures.

Above is the same program as before implemented utilizing the P4 library.

The structure "slave_data" defines those pa-

rameters that will be sent to the slave nodes. Line 9 defines two variables that are used by a timing routine. The identifier "partial_pi" refers to a pointer to a type double, rather than a function as before. Each slave is sent a program (shown below) that is compiled separately from the `main()` program. The other variables have the same meaning as before.

The p4 functions initialize the p4 system and provide for the passage of command line arguments. P4 allows for command line arguments setting a "debug level" for both host and node programs. Print statements will be output or passed over depending on their user-defined debug level, a variable accompanying each p4 print function (not shown here). The master process is assigned a process id of "0" and the slaves have their own unique id numbers starting with "1".

Lines 19–29 as before allow the user to request the number of "nrecs" and the number of "slaves".

Line 30 initiates the timing and Lines 31 to 33 define the variables to be sent to the slave nodes. As with Linda a separate file (named on the command line) lists the names of the slave machines and the path of the executable to be utilized. Thus, the user can send the appropriate compiled program to the proper architecture.

The for loop sends nonblocking messages of type 100 to the slaves whose id numbers are `1...slaves`. If the function `p4_sendr()` had been utilized instead, it would wait for an acknowledgment before proceeding. If the second argument in Line 34 is −1, the message is broadcast.

The slaves will return the results in a message of type 200. The variable that will be instantiated must be initialized to "NULL".

The loop collects results from each slave and adds it to the total. The variable "from" must be initialized to each loop iteration (−1 receives messages from anyone) because upon receipt of the message "from" is set to the id of the sender.

The timer is stopped in Line 45 and the resultant value calculated for pi and the elapsed time is output. Line 48 exits the p4 environment after the slave processes have ended.

Each slave process is sent to a separate machine listed in the command line file. As before, if not enough machines are available, the jobs will be queued up. The slave code is shown below.

```
 1  #include "p4.h"
 2
 3  slave()
 4  {
 5     int    section, slaves, lower, upper, nrecs, type, from, size, me, j;
 6     double partial_pi, h, x, sum = 0.0;
 7     struct slave_data;
 8      {
 9        int    nrecs;
10        int    section;
11        float  h;
12      } *data;
13
14     me = p4_get_my_id() - 1;
15     slaves = p4_num_total_slaves();
16     type = 100;
17     from = 0;
18     data = NULL;
19     p4_recv(&type, &from, &data, &size);
20     nrecs   = data -> nrecs;
21     section = data -> section;
22     h       = data -> h;
23     lower = (me * section) + 1;
24     upper = lower + section - 1;
25     if (me == (slaves - 1))
26        upper = nrecs;
27     for (j = lower; j <= upper; j++)
```

```
28        {
29          x = (j - 0.5) * h;
30          sum = sum + (4.0/(1.0 + x * x));
28        }
29      partial_pi = h * sum;
30      p4_sendr(200, 0, &partial_pi, sizeof(partial_pi));
31  }
```

Line 14 defines a local variable "me", which is the slave id number minus one.

The slave process can ask how many slaves there are (Line 15) and each slave anticipates a message of type 100 coming from the master (whose id is 0).

Lines 20–29 perform the same compute intensive calculations as Lines 37 to 55 of the Linda code.

The individual contribution of each slave is sent with a type of 200 (cf. Line 41 of the host code) to the specified instance of the spawning process.

It is possible to use P4 with the dbxtool accompanying the SunView window environment in addition to the P4 provided debug levels mentioned above.

5 LINDA, P4, AND AN LU FACTORIZATION ALGORITHM

In order to study a larger, nontoy application fragment, an LU factorization code [18] was implemented in Linda and P4 utilizing a right-looking submatrix algorithm [19, 20]. The resulting codes were about 3,700 and 4,800 lines of code, respectively.

6 PERFORMANCE COMPARISON

Performance results (rounded off to nearest second) for six SPARC 1 workstations for the pi program are shown in Table 1. Not surprisingly, there is no significant difference between timings for the two tools for this low communication program. If eight or more slaves are utilized for execution with an input of only 1.0E6, the time required is slightly greater than it is for the case of six slaves. The overhead of starting up eight processes is greater than the gain of compute power for such a small data set. With large input data, the pi program is essentially linear in processors.

The results of the LU factorization program (Table 2) for P4 show that if the number of processes is doubled, the execution time does not approach half until the size of the matrix is increased. The efficiency (peak/actual) of Linda execution (640 size) was 50%. In addition, as the matrix size increases, the machine resources required increase more than linearly. For larger matrices than the ones shown, greater than eight machines were required to keep in bounds the resources required on any one slave. The six machines utilized for the calculations shown in Tables 1 and 2 were all under the control of one super user and all were diskless nodes connected to a single server. At the time of the experiments reported here, the machines were idle, i.e., no other users were running jobs. The net connecting the machines is part of a larger net and had an unknown amount of traffic. Repetitive runs afforded similar results. We have begun a series of experiments with much larger networks that involve the use of workstations where we do not have root privileges and that are separated by gateways from our original group of six. Moreover, if some of the machines are not in use, our remote processes compete with the screenlock programs invoked by absent owners. It is also harder to restore allocated resources or kill "broken" processes created by coding errors or by the fact that the machines in our pool had different amounts of

Table 1. A Calculation of Pi by the Rectangle Rule with a Network of SPARC (Sun 4) Workstations (in Seconds)

Rectangles	Slaves	P4	Linda
1.0E6	2	4	4
	4	3	2
	6	2	1
1.0E7	2	45	41
	4	21	21
	6	14	14
2.0E7	2	80	84
	4	43	42
	6	28	29

Table 2. LU Factorization with SPARC (Sun 4) Workstations (in Seconds)

	Four Slaves		Eight Slaves
Matrix Size	Linda	P4	P4
640	46	57	40
1,280	321	373	245
1,600	444	767	433
1,920	864	1,317	711

such system resources as swap space. Load balancing is also more difficult to achieve with larger groups of machines where the demands of users and the resources available are harder to ascertain.

The numbers reported here are not absolutes, and there are continual upgrades to both the tools utilized. The numbers should serve to illustrate the tradeoffs and speedups that can be obtained. The fact that the Linda implementation was somewhat faster than the P4 one does not imply the same will hold for other algorithms. The preference between any two paradigms will also be dependent on the architectures supported, ease of learning, debug tools, and programmer's preferences. Two other approaches to point to point message-passing paradigms are PICL [21] and PVM [22]. Isis [23] is yet another package for distributed computation. If one machine in a network should become inoperable or not respond, the net reconfigures itself so as to continue to function.

The allocation of tasks could assess the current workload of the nodes in the net in order to assign work efficiently. Toward this goal, network Linda, if given a list of machines greater than the number of "eval" functions, will utilize the least loaded machines first, based on the response to a UNIX "uptime" command.

7 RELATED WORK

Because the message-passing paradigm is unfamiliar to many programmers and is relatively low level, attempts have been made to devise languages at higher levels of abstraction that better match the way programmers tend to view their algorithms. Several languages have been developed for distributed MIMD multiprocessors in which the message passing is invisible to the programmer. The user defines a virtual machine and then maps distributed data structures to that virtual machine. An example is the DINO (Distributed Numerically Oriented) language [24] package, which is in an early stage of development. It supports only C and is available for the Intel i860 machine. There are presently no debugging tools available, but there are some provisions for functional parallelism.

Recently, a public domain software package named Distributed Queuing System (DQS) [25] has become available for a number of UNIX-based machines. DQS is not a parallel tool, per se, in that its function is to distribute batch workloads to various available idle machines. If the idle machine subsequently receives any input (e.g., keyboard), the DQS job is suspended (but still resides on the machine) and restarts after the machine in question is idle for a predetermined length of time. The DQS software has the capacity of launch parallel PVM programs and it can be adapted to launch Linda and P4 programs as well. Thus, DQS along with parallel packages can be utilized with a goal to harnessing the combined compute power of scattered machines. Presently, DQS merely consults a list of available machines. Future implementations of this and similar packages (commercial products are appearing) could consider relative compute power of a heterogeneous network, availability times, process migration to another machine, and even the relative need to "idle" a process as opposed to "migrating" it.

A compendium comparing 12 different parallel Fortran implementations of the pi program example discussed above has been published [26]. This was based on earlier work that also discusses the parallel hardware on which the programs were executed [14]. A compendium that gives a brief classification and summary of parallel programming tools can be found in Chang and Smith [27] and is a source of further references.

8 SCIENTIFIC PARALLEL PROCESSING

The commercial success of a particular computer or of a particular software package is dependent on the timing of its release, its utility, and the quality of its documentation. Regarding this last point, whereas the ratio of programmers to manual readers is likely much greater than one, many vendors do not seem to appreciate the value of complete, clearly written, and carefully indexed documentation—with examples. This situation is made worse by the lack of agreement among vendors and users for standardizing parallel language extensions or naming conventions. Linda and P4

are sufficiently easy to learn that the accompanying documentation is at least satisfactory.

A great deal of attention has been dedicated to "load balancing" issues for expensive multiprocessors. With today's trend toward cheap parallel capable workstations, an idle central processing unit (CPU), per se, is less of a worry. Surrendering the use of a processor must be balanced with the overhead incurred if the use of that processor must be requested later on. In the case of early hypercube architectures, the large overhead of sending messages had a great effect on the algorithms that could be effectively utilized. Later versions of hypercubes have greatly reduced this problem, thus allowing the overhead of message passing to become a major consideration rather than an overwhelming characteristic.

Tools that profile runtime performance (e.g., percent breakdown of time for each task, subtask, etc.) are a great aid in determining whether parallelization should even be considered for a particular module. Surprisingly, portability of parallel code does not seem to be a high priority issue for many scientific programmers. It seems sufficient for some installations (e.g., academic installations) that programs produced in their environments be executable only on the hardware they possess or are likely to possess in the foreseeable future. For other environments (e.g., corporations) portability is a high priority issue (e.g., building portable libraries).

Debugging parallel codes has all of the problem of verifying correctness as scalar ones, along with some additional features. Language extensions built on older compilers may not place restrictions on how the parallelization is achieved and cannot enforce correct usage. As a result, it is common for an improperly constructed program not to exit with an error message, core dump, or bus error. The program may be in memory but there is no indication of normal execution. It may be waiting for a message that is not forthcoming. Sending interrupts to such programs can leave "broken" processes and resources scattered throughout a net.

Even the output of correct answers does not necessarily guarantee that the program is correct. It is quite possible to write a parallel code that executes correctly 99 successive times and fails on run number 100 due to a race condition or variable contention. A race condition could cause code to fail ONLY when not in debug mode (or vice versa). Actually, very little progress has been made developing tools for debugging distributed computation. The postmortem analyzers and graphic displays presently available (e.g., HeNCE [22]) are far more useful for performance analysis, for example, showing bottlenecks.

9 SUMMARY

For the foreseeable future, Fortran will likely continue to be the workhorse language for parallel scientific computing. Neither Fortran nor C supports parallelization directly without at least one extension. An international committee has decided what features Fortran ought to include (Fortran 90 standard). However, in the United States, Fortran 77 is also being retained as a standard. For example, Fortran 90 has recursion and includes files, whereas Fortran 77 supports neither. Each commercial Fortran compiler adds its own set of extensions. Recently, the Parallel Computing Forum (PCF) [28] finished a draft standard for shared memory parallel processing in Fortran and C to support portability of parallel programs between vendors. The last meeting occurred in April 1990 and the results of the PCF efforts are undergoing further standardization activities by a committee (X3H5) of the American National Standards Institute (ANSI). Presently, that committee has not yet produced a set of standards for Fortran or C. The X3J3 committee of ANSI is also reviewing the recommendations of the High Performance Fortran Forum [29] for writing SIMD-style data parallel programs.

Distributed memory architectures offer a solution to avoiding memory and variable contention, are perhaps more easily scalable, but have the disadvantage that the speed of implementing message passing is growing more slowly than the speed of the processors that perform the node computations and send the messages. Most of the early research on parallelization has been for shared memory architectures that are suitable for a wide range of granularities, are usually considered easier to debug, but suffer from limited scalability due to bus bottlenecks.

Current parallel machines range from designs with a small number of very fast processors with a large available memory (e.g., Cray) suitable for large grained parallelization, to machines with a very large number of slower processors (massively parallel) with a small amount of memory per processor (e.g., connection machine [30]). The connection machine attempts to use order (N) processors to process problems of size N [31]. Recently,

another possibility has surfaced: A machine has been announced that can contain up to 16,000 RISC chips. A model with 1,024 CPUs would cost about 25 million dollars [32]! We have already seen the commercial appearance and disappearance of several parallel machines (e.g., Denelcor HEP, FPS T series, Multiflow, Loral LDF, and the BBN Butterfly), which for one reason or another are no longer available.

What level of abstraction will enjoy the greatest success for scientific programming? As the programmer is partially removed from the programming "loop", that is as programmers rely more on tools and code generators, they may lose sight of the fundamentals of parallel code implementation. This has already happened in the area of graphics, where many users of graphics tools do not know, or care about, the operation of these tools at a fundamental level.

The need for speedup must be balanced against program development time, the debug tools and maintenance available, and the expected lifetime of the parallel tools employed and the parallel architectures they target.

The help and contributions of James Patterson, Robert G. Babb II, and Cleve Ashcraft to this work are gratefully acknowledged.

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture, A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann Publishers, 1990.

[2] Linda, Scientific Computing Assoc., Inc., 246 Church St., Suite 307, New Haven, CT 06510 (It is not the intention here to rigorously define the differences between language extensions, precompilers, and programming environments. The last phrase is this vendor's description of Linda).

[3] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens, *Portable Programs for Parallel Processors*. New York, NY: Holt, Rinehart, and Winston, 1987.

[4] Workshop on Cluster Computing. Florida State University, Tallahassee, FL, Abstracts available via anonymous ftp from ftp.scri.fsu.edu., 1992.

[5] B. Forrest, D. Roweth, N. Stroud, D. J. Wallace, and G. V. Wilson, "Neural network models," *Parallel Comput.*, vol. 8, pp. 71–83, 1988.

[6] T. MacDonald, "C versus Fortran-77 for scientific programming," Sci. Prog., vol. 1, pp. 99–114, 1992.

[7] FPS Computing, P.O. Box 23489 Portland, OR 97223.

[8] W. R. Cowell, "An Introduction of VecPar_77," Numerical Algorithms Group, Inc., 1400 Opus Place, Suite 200, Downers Grove, IL 60515, 1990.

[9] Forge, Pacific-Sierra Research Corp. 12340 Santa Monica Blvd., Los Angeles, CA 90025.

[10] W. R. Cowell, *User's Guide to Toolpack/1-Tools for Data Dependency Analysis and Program Transformation*. Argonne, IL: Argonne National Laboratory ANL-88-17, 1988.

[11] Parasoft Corp., 27415 Trabuco Circle, Mission Viejo, CA 92692.

[12] Kuck and Assoc., Inc. 1906 Fox Drive, Champaign, IL 61820.

[13] D. Gelernter and N. Carriero, "Coordination languages and their significance," Communications ACM, vol. 35, pp. 96–107, 1992.

[14] R. G. Babb II, *Programming Parallel Processors*. Reading, MA: Addison-Wesley, 1987.

[15] Superconcurrency System Solutions, Inc. 1100 NW Compton Drive, Suite 309, Beaverton, OR 97006.

[16] S. E. Zenith, *Programming with Ease; Semiotic Definition of the Language*. New Haven, CT: Yale University RR 809, 1990.

[17] Sun Microsystems, Inc. Sun Network Programming Manual, Part Two, Protocol Specification, 1988.

[18] G. H. Golub and C. F. Van Loan, *Matrix Computation*. Baltimore, MD: The Johns Hopkins University Press, 1989.

[19] C. Ashcraft, *The Distributed Solution of Linear Systems Using the Torus Wrap Data Mapping*. Bellevue, WA: Boeing Computer Services ECA-TR-147, 1990.

[20] C. Ashcraft, *A Taxonomy of Distributed Dense LU Factorization Methods*. Bellevue, WA. Boeing Computer Services ECA-TR-161, 1991.

[21] G. Geist, M. Heath, B. Peyton, and P. Worley, PICL: *A Users' Guide to PICL. A Portable Instrumented Communication Library*. Oak Ridge, TN: Oak Ridge National Laboratory TM-11616, 1990.

[22] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam, The PVM and HeNCE Projects. PVM and HeNCE may be obtained by sending the email message "send index from pvm" to netlib@ornl.gov.

[23] K. Birman and R. Cooper, *The ISIS Project: Real Experience with a Fault Tolerant Programming System*. Ithaca, NY: Cornell University 90-1138, 1990.

[24] T. M. Derby, E. Eskow, R. K. Neves, M. Rosing, R. B. Schnabel, and R. P. Weaver, *The DINO User's Manual*. Boulder, CO: University of Colorado CU-C5-501-90, 1990.

[25] DQS, *Distributed Queuing System*. Tallahassee, FL: Supercomputer Computations Research Institute, Florida State University, 1992.

[26] A. H. Karp and R. G. Babb II. "A comparison of 12 parallel Fortran dialects." IEEE Software, vol. 5, pp. 52–67, 1988.

[27] L. Chang and B. T. Smith. *Classification and Evaluation of Parallel Programming Tools.* Albuquerque, NM: University of New Mexico C590-22, 1990.

[28] B. Leasure. The chairman of the X3H5 committee can be reached at 1906 Fox Drive, Champaign, IL 61820.

[29] "High Performance Fortran Language Specification". Houston, TX: Rice University, Version 0.4, 1992.

[30] W. D. Hillis. *The Connection Machine.* Cambridge, MA: The MIT Press, 1985.

[31] W. D. Hillis and G. L. Steele, "*Data parallel algorithms.*" *Communications ACM*, vol. 29, pp. 1170–1183, 1986.

[32] Parallelogram International, December 1991–January 1992, p. 4.