OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

Dataparallel C: A SIMD Programming Language for Multicomputers

Michael J. Quinn
Ray J. Anderson
Computer Science Department
Oregon State University
Corvallis, OR 97331

Philip J. Hatcher
Anthony J. Lapadula
Robert R. Jones
Department of Computer Science
University of New Hampshire
Durham, NH 03824

90-80-6

Dataparallel C: A SIMD Programming Language for Multicomputers

Philip J. Hatcher†, Michael J. Quinn‡, Anthony J. Lapadula†, Ray J. Anderson‡, and Robert R. Jones†

† *Department of Computer Science, University of New Hampshire, Durham, NH 03824*
‡ *Department of Computer Science, Oregon State University, Corvallis, OR 97331*

Abstract

Dataparallel C is a SIMD extension to the standard C programming language. It is derived from the original C* language developed by Thinking Machines Corporation. We have nearly completed a third-generation Dataparallel C compiler, which transforms Dataparallel C programs into SPMD-style C code suitable for compilation and execution on NCUBE multicomputers. In this paper we elaborate on the characteristics and strengths of data-parallel programming languages. We summarize the syntax and semantics of Dataparallel C, present six benchmark programs, and document the performance of these programs executing on the NCUBE 3200 multicomputer. Our work demonstrates that SIMD source programs can achieve reasonble speedup when compiled and executed on MIMD computers.

Key Words

compiler, data parallel, hypercube, MIMD, multicomputer, programming language, SIMD

1. Introduction

The typical multicomputer programming language is an imperative language, such as C or FORTRAN, augmented with a variety of message–passing constructs. The difficulty of programming multicomputers using such languages is well known [1, 2, 3, 4, 5]. In response to this problem, a large number of high-level parallel programming languages have been proposed, including Booster [6, 7], C-Linda [8, 9], Crystal [10, 4], Coherent Parallel C [11], DINO [12, 13], Kali [14], Parallel Pascal [15, 16], and SEYMOUR [17].

In this paper we describe the high-level language Dataparallel C, which is derived from the original C* language developed by Thinking Machines Corporation [18]. Dataparallel C is a data-parallel programming language: the only mechanism by which parallelism is achieved is through the simultaneous application of a single operation to an entire data set [19]. The fully synchronous semantics and the local view of the computation give Dataparallel C a number of important advantages over languages without these attributes. Synchronous execution eliminates race conditions and makes Dataparallel C programs deterministic, greatly reducing the complexity of program debugging. Having a local view of the computation simplifies the introduction of data decomposition directives, which are essential in a distributed memory environment.

This paper describes experience with our third generation Dataparallel C compiler for a hypercube multicomputer. Our first compiler was based upon a general, but less efficient, control flow model [20]. Our second compiler introduced both efficient flow of control and a powerful communication optimizer, but was implemented in prototype form only [21, 22, 23]. The third, and current, compiler is intended to be usable by the general research community. It features a full implementation of the language, including general pointer-based communication and support for separate compilation. The compiler included new optimizations and utilizes an improved set of communication primitives.

In the remainder of this paper we summarize the characteristics we have chosen for our high-level language, give an overview of Dataparallel C, and present the results of compiling and executing a set of benchmark Dataparallel C programs on a 64–node NCUBE 3200 hypercube.

Few high-level parallel programming environments are available to those who want to solve problems on parallel computers or design new parallel algorithms. We hope that this compiler, which can produce code for workstations and multicomputers, will stimulate further research in parallel computing.

2. Design Choices for a High-Level Language

The parallel programming community has recognized the need for better languages, and dozens of alternatives have been proposed. An examination of these languages reveals that the fundamental design questions have been answered in virtually every possible way. Should the language be imperative, functional, or based on logic programming? Should the parallelism be implicit or explicit? Should the computation be expressed in terms of global activity or from the view of one of the processes? Should the processes execute a single instruction stream or multiple instruction streams? Should memory be viewed as distributed or shared?

We anticipate that there will be a variety of successful higher-level parallel programmng languages available to programmers of multicomputers in the next decade. However, we have chosen to concentrate on parallel languages with the following characteristics:

Imperative style. Imperative languages are more familiar to most programmers and can be compiled more efficiently. We believe imperative languages are so well established that they will continue to be used, even if the interesting work being done in the areas of logic programming and functional programing languages leads to efficient implementations on parallel machines.

Explicit parallelism. The programmer and compiler must work as a team to produce good parallel code. It is ridiculous for a programmer writing a new application to hide the parallelism inside sequential control structures and then ask the compiler to extract parallelism from the sequential code. The compiler-writer's job is hard enough—why add unnecessary complexity?

Local view of the computation. The language associates a virtual processor with the fundamental unit of parallelism, and the computation is expressed in terms of the

operations performed by the virtual processors. A key problem in generating code for multicomputers, which have no shared memory, is determining how to distribute the data among the individual memories of the physical processors. The compiler's job is simplified enormously when the programmer expresses the computation in terms of the actions of the virtual processors.

Synchronous execution of a single instruction stream. Inside parallel code the virtual processors execute the same instructions in lock step. In other words, the language is SIMD (single instruction stream, multiple data stream).

Global name space. Memory is distributed among the virtual processors, but every virtual processor can access the values of any other virtual processor. Processor interaction is through expressions, rather than explicit messages.

Dataparallel C has these characteristics. As a result, it has the following desirable attributes:

Versatility. Data parallelism is the natural paradigm for a large fraction of problems in science and engineering. In his study of 84 separate applications in the areas of biology, chemistry and chemical engineering, geology, earth and space science, physics, astronomy, computer science, and other disciplines [24], Fox has found that "the source of parallelism is essentially always domain decomposition or data parallelism; a simple universal technique to produce high performance scaling parallel algorithms" [25].

Practicality. It is easy to convert sequential C programs into Dataparallel C code, because Dataparallel C allows arbitrary control structures within domain select statements. During a normal conversion process, many functions and the interior portions of many loops can be lifted from the C program and inserted into the Dataparallel C program with very few changes.

Programmability. Data-parallel programs are easier to write than programs using lower-level parallel constructs, because the synchronous model of execution means that there is only one locus of control: race conditions and deadlock are impossible. The illusion of a global name space means that the programmer does not have to get involved with explicit message passing, even if the underlying architecture does not have a global name space. The existence of virtual processors simplifies the data partitioning task.

Together, the synchronous model of execution, global name space, and virtual processors cause Dataparallel C programs to be much shorter than programs written in languages with low-level parallel constructs. In fact, Dataparallel C programs are usually about the same length as the corresponding sequential program written in C. The extra code for defining parallel data structures and delimiting parallel program sections is compensated for by an elimination of `for` loops that sequentialize inherently parallel operations.

Dataparallel C programs are easier to debug, too. Debugging MIMD programs is difficult, because interacting asynchronous processes can exhibit deadlock, race conditions, and nondeterminism. The execution of data-parallel programs is easily made deterministic, and the synchronous processes interact through shared variables in a predescribed manner—deadlock and race conditions are impossible.

Portability. Because Dataparallel C is based on a high-level abstract model of parallel computation, Dataparallel C programs are more machine-independent than programs written in a language closer to the underlying hardware. As long as novel parallel computer architectures continue to appear with regularity, portability will be an especially valuable commodity.

Reasonable performance. Our benchmark suite of Dataparallel C programs demonstrates that data-parallel programs can achieve high speedups on MIMD computers. In most cases our compiler does not generate code that executes as fast as programs hand written in a lower-level language. Karp and Babb have called programming languages with low-level parallel constructs "the equivalent of machine language" [2], and we feel the analogy is a good one. A proficient assembly language programmer can usually construct a program that executes faster than one automatically compiled from a higher-level language. If assembly language programs execute faster, why does anybody use a higher-level language? Because other qualities have value, such as programmer productivity, code portability, and maintainability. In the general-purpose computing arena, execution time is not the only criterion. As parallel computing enters the mainstream, extracting every possible parallel cycle will become less important, and other issues will get the attention they deserve.

3. Overview of Dataparallel C

The Dataparallel C programming language is very similar to the original C* language designed by Rose and Steele [18]. We have added the notion of *virtual topologies*, extended the specification of pointers, and made array assignment a part of the language.

The conceptual model presented to the Dataparallel C programmer is that of a front-end uniprocessor attached to an adaptable back-end parallel processor. The sequential portion of the Dataparallel C program (consisting of conventional C code) is executed on the front end. The parallel portion of the Dataparallel C program (delimited by constructs not found in C) is executed on the back end.

The back end is adaptable in that the programmer selects the number of processors to be activated. This number is independent of the number of physical processors that may be available on the hardware executing the Dataparallel C program. For this reason the Dataparallel C program is said to activate *virtual* processors when a parallel construct is entered.

Virtual processors are allocated in groups. Each virtual processor in the group has an identical memory layout. The Dataparallel C programmer specifies a virtual processor's memory layout using syntax similar to the C `struct`. A new keyword `domain` is used to indicate that this is a parallel data declaration. Figure 1 contains a partial domain declaration for the mesh points of a hydrodynamics simulation. As in C structures, the names declared within the domain are referred to as *members*.

Instances of a domain are declared using the C array constructor. Each domain instance becomes the memory for one virtual processor. The array dimension therefore indicates the

```
domain cell {double energy, density, temperature, pressure;};
```

Fig. 1. Declaring a domain.

```
#define KDIM 54
#define LDIM 54
domain cell mesh[KDIM][LDIM];
```

Fig. 2. Declaring virtual processors.

```
[domain cell].{
   double temp1;
   temp1 = calculate_temperature(energy,density);
   temperature = (temp1 > TFLR ? temp1 : TFLR);
   pressure = calculate_pressure(temperature,density);
}
```

Fig. 3. Activating virtual processors.

size of the virtual back-end parallel processor that is to be allocated. Figure 2 contains a domain array declaration. Note that domain arrays can be multidimensional. The number of virtual processors allocated is the product of the array dimensions.

Data located in Dataparallel C's front-end processor is termed *mono* data. Data located in a back-end processor is termed *poly* data.

Figure 3 illustrates the Dataparallel C *domain select* statement. The body of the domain select is executed by every virtual processor allocated for the particular domain type selected. The virtual processors execute the body synchronously. The domain members are included within the scope of the body of the domain select. These names refer to the values local to a particular virtual processor.

The code executing in a virtual processor of a Dataparallel C program can reference a variable in the front-end processor by referring to the variable by name. A variable that is visible in the immediately enclosing block of a domain select statement is visible within the domain select. The Dataparallel C compiler is responsible for making mono variables accessible at run time to the virtual processors.

Similarly, the members of a domain instance are accessible everywhere in a program. The members of one domain can be read and written from within a domain select statement for a different domain. Poly data can also be read and written from the sequential portion of the program. The syntax employed is to provide a full domain array reference followed by a member reference.

Dataparallel C, like C++, has a keyword `this`. In Dataparallel C `this` is a pointer to the domain instance currently being operated on by a virtual processor. Pointer arithmetic on `this` can be performed to access other virtual processors' members.

Dataparallel C provides a set of *reduction* operators, which accumulate poly values into a mono location. All C assignment operators are overloaded for this purpose. The language includes new operators to express reductions that compute the minimum and maximum of a set of poly values.

The sequential portion of a Dataparallel C program is just C code and executes according to the normal C semantics. Conceptually, the parallel sections of a Dataparallel C program execute synchronously under the control of a *master program counter* (MPC). A virtual processor's local program counter is either active, executing in step with the MPC, or inactive, waiting for the MPC to reach it.

For example, the MPC steps through an `if-then-else` statement by first evaluating the control expression, then executing the `then` clause, and finally executing the `else` clause. A local program counter would also proceed first to the control expression. However, if the expression evaluated to zero (*false* in C), then the local program counter would proceed to the `else` clause and wait for the MPC to reach it. If the expression evaluated to nonzero (*true* in C), then the local program counter would wait at the `then` clause for the MPC.

As well as being synchronous at the statement level, Dataparallel C is also synchronous at the expression level. No operator executes within a virtual processor unless all active virtual processors have evaluated their operands for the operator. Once the operands have been evaluated, the operator is executed as if in some serial order by all active virtual processors. This seemingly odd use of a serial ordering to define parallel execution is required to make sense of concurrent writes to the same memory location.

Our implementation of Dataparallel C allows additional information to be provided by the programmer in order to aid the compiler in the mapping of virtual processors to physical processors. The array dimension of the domain array establishes a *virtual topology*. A one-dimensional domain array is considered to be a ring of virtual processors. A two-dimensional domain array is considered to be a two-dimensional mesh of virtual processors with wraparound connections. In general, an n-dimensional domain array is considered to be an n-dimensional mesh of virtual processors with wraparound connections.

These virtual topologies establish a convention of *locality*. Virtual processors that are adjacent in a virtual topology should be mapped by a compiler to physical processors that are "near" each other. On some architectures this information will be of little value and can be ignored by the compiler. On other architectures this can lead to large efficiency gains if the programmer exploits the feature and the compiler effectively implements it.

For the common topologies (low dimension domain arrays), the compiler recognizes macros that provide convenient access to adjacent elements in the virtual topology. The macros return the address of the appropriate adjacent domain element. In the one-dimensional case macros called `successor` and `predecessor` provide access to ring neighbors; in the two-dimensional case the macros `north`, `south`, `east` and `west` provide access to mesh neighbors.

Additional keywords exist to aid the compiler in mapping a larger number of virtual processors to a smaller number of physical processors. The keyword `contiguous` indicates

that blocks of adjacent domain elements should be mapped to the same physical processor. The keyword `interleaved` indicates that domain elements should be assigned to physical processors in a round-robin fashion. In the two-dimensional case, the keywords `contiguous_row`, `contiguous_col`, `interleaved_row`, and `interleaved_col` exist to map rows and columns *in toto*. The keyword `userspec` indicates that the compiler should utilize user-written macros to implement an arbitrary mapping.

4. Benchmark Programs and Results

In this section we present a selection of short Dataparallel C programs that serve as a preliminary benchmark suite for the compiler. We are currently in the process of implementing a variety of nontrivial applications in Dataparallel C, including the SIMPLE benchmark from Lawrence Livermore Laboratory, an irregular mesh computation, and an underground contaminant transport model.

Many parallel programming environments have been reported in the literature, but there is a decided lack of hard performance data to facilitate comparisons between the various approaches. We invite comparisons between the performance of our compilers and other portable parallel programming environments, and to that end the programs in this section are completely self-contained. Every program generates whatever input data it needs and prints some sort of check sum result.

A. Numerical Integration

The area under the curve $4/(1+x^2)$ between 0 and 1 is π. One way to compute the value of π, then, is to approximate the integral by calculating the sum

$$\pi \approx \frac{1}{N} \sum_{i=0}^{N-1} \frac{4}{1+x_i^2}$$

using the rectangle rule, where $x_i = (i+1/2)/N$ is the midpoint of the ith interval. This numerical integration algorithm is the basis for two works contrasting the programming environments on a variety of parallel architectures [1, 2].

The problem is amenable to a data-parallel solution. We associate one virtual processor with every interval. Each virtual processor computes the height of its rectangle, and then all the individual heights are added to form a global sum. The total area is determined by multiplying the total height by the rectangle width. The Dataparallel C implementation of the π-finding algorithm appears in Figure 4.

B. Relatively Prime Number Computation

The problem is to count the number of integer pairs (i, j) such that i and j are relatively prime, for $2 \leq i, j \leq N+1$. Two numbers are relatively prime if they have no prime factors in common; i.e., if their greatest common divisor is 1. For example, 3 and 5 are

```
/* Computation of pi using rectangle rule  */

#define INTERVALS 400000
#define ID        (this-chunk)

domain span { char dummy; } chunk[INTERVALS];

main ()
{
   double sum;        /* Sum of areas */
   double width;      /* Width of interval */

   width = 1.0 / INTERVALS;

   [domain span].{
      double x;      /* Midpoint of rectangle on x axis */
      x = (ID+0.5)*width;
      sum = (+= (4.0/(1.0+x*x)));
   }
   sum *= width;
   printf ("Value of pi is %14.12f\n", sum);
}
```

Fig. 4. Dataparallel C program to compute π using numerical integration.

relatively prime, since their greatest common divisor is 1, while 6 and 15 are not relatively prime, since their greatest common divisor is 3. The greatest common divisor of two integers can be found using Euclid's algorithm, which repeatedly replaces the larger value with the difference between the values until the two values are equal.

This algorithm can be characterized as "embarrassingly parallel," since all candidate pairs may be examined simultaneously. It would be a mistake, however, to assume that simply because the virtual processors do not interact, a parallel algorithm will achieve high speedup. The first difficulty we encounter has to do with establishing the fundamental unit of parallelism. If we choose the integer pair to be the fundamental unit of parallelism and declare a two-dimensional array of domain instances to represent all possible pairs, then only those instances above the diagonal will be active (since we cannot count both (i, j) and (j, i)). Creating twice as many virtual processors as needed wastes memory and reduces the maximum size problem we can solve. If we choose all integer pairs beginning with element i to be the fundamental unit of parallelism and declare a one-dimensional array of domain instances (one for every possible value of i), then there will be a high-level load imbalance between the virtual processors. We have decided to take another course: we define one virtual processor for each of the $N(N-1)/2$ unique pairs. This option solves the memory utilization and load-balancing problems; its disadvantage is the number of arithmetic operations needed for a virtual processor to determine which integer pair it is responsible for.

An interesting feature of the problem is the variance in the amount of time needed to determine the greatest common divisor of two integers using Euclid's algorithm. For

example, determining that 6 and 12 are not relatively prime requires two comparisons and one subtraction, while determining that 6 and 13 are relatively prime requires eight comparisons and seven subtractions. The distribution of virtual processors to physical processors can determine whether all physical processors have a reasonable mix of "easy" and "difficult" pairs.

It is important to note, however, that MIMD computers have an advantage over SIMD computers when emulating virtual processors in blocks whose execution times have large variances. On a SIMD computer, no virtual processor may continue execution beyond the end of a `while` loop until all processing elements have completed execution, because the system supports only a single instruction stream. A MIMD computer can execute a program that is semantically equivalent, yet not so tightly synchronized, taking advantage of the ability of every processor to execute its own instruction stream. Since there are no interactions among the virtual processors, CPUs on a MIMD computer can race ahead on the easy pairs and lag behind on the difficult pairs. In other words, every CPU executes at full speed throughout the computation, and the only inefficiency occurs at the end of the computation, when the CPUs must synchronize. For this reason MIMD computers can achieve higher efficiency than SIMD computers when executing embarrassingly parallel programs.

The Dataparallel C program to count relatively prime integer pairs appears in Figure 5.

C. Matrix Multiplication

Matrix multiplication is a tempting target for a case study. The standard sequential algorithm has time complexity $\Theta(N^3)$, and there are no data dependencies, so any reasonable parallel algorithm will have a good grain size.

Matrix multiplication may be parallelized in a variety of ways, but the Dataparallel C program we benchmark is based upon a row-wise decomposition of the first matrix. A row-wise decomposition makes use of the following property: when multiplying two matrices **A** and **B** to yield matrix **C**, the ith row of **C** is the product of the ith row of **A** and the entire matrix **B**. All of these vector-matrix products may be computed simultaneously.

Our Dataparallel C program assigns one row of **A**, one column of **B**, and one row of **C** to every virtual processor. The program appears in Figure 6.

D. Warshall's Algorithm

The transitive closure of an $N \times N$ binary relation can be found through $\lceil \log N \rceil$ matrix multiplication-like steps, where multiplication is replaced by logical "and" and addition is replaced by logical "or." The time complexity of the resulting algorithm is $\Theta(N^3 \log N)$. Warshall observed that by pulling the innermost loop to the outermost level, only a single iteration is required, reducing the complexity of the algorithm to $\Theta(N^3)$.

Although Warshall's algorithm bears a superficial similarity to the standard matrix multiplication algorithm, it is not "embarrassingly parallel," since there are data dependencies between variables set and used in different iterations of the outermost loop.

```
/* Program to count number of relatively prime pairs */

#include <math.h>

#define N       128
#define ID      (this-x)
#define FIRST   (((1 + (int) sqrt((double) (8*ID+1)))/2)+1)
#define SECOND  (ID - (((i-1)*(i-2))/2) + 1)

domain cell { char dummy; } x[N*(N-1)/2];

main()
{
   int count = 0;

   [domain cell].{
      if (gcd(FIRST,SECOND) == 1) count += (poly) 1;
   }
   printf ("Number of relatively prime pairs is %d\n", count);
}

int gcd(i, j)
   int i, j;
{
   while (i != j)
      if (i > j) i -= j;
      else j -= i;
   return (i);
}
```

Fig. 5. Dataparallel C program to count number of relatively prime integer pairs.

A Dataparallel C version of Warshall's algorithm appears in Figure 7. Although the speed of the program can be increased by adding a test before the innermost `for` loop, we have omitted this test in order to make the execution time of the algorithm less data dependent.

E. Gaussian Elimination

Gaussian elimination is an $\Theta(N^3)$ algorithm used to solve a system of linear equations **Ax=b** for the vector **x**. The principle behind Gaussian elimination is to reduce the number of unknowns from a system of linear equations by adding multiples of rows to other rows. After matrix **A** has been reduced to upper triangular form, back substitution is used to diagonalize the matrix. Once all of the off-diagonal elements have been reduced to zero, the elements of **x** may be found directly.

Since Gaussian elimination is row-oriented, we choose to make the row the fundamental unit of parallelism. The algorithm requires `N-1` iterations to reduce an `N`×`N` system. During iteration i the column i value in every unmarked row is driven to 0 by taking a linear combination of that row and the pivot row. For numerical stability the pivot row chosen is

```
/* Matrix multiplication in Dataparallel C */

#define N   128
#define ROW (this-x)
domain row { float a[N], btrans[N], c[N]; } x[N];

main()
{
   float checksum;
   int   j, k;

   [domain row].{
      for (j = 0; j < N; j++) {     /* Initialize matrices */
         a[j] = 1.0 + N * ROW + j;
         btrans[j] = 1.0 / (1.0 + N * j + ROW);
      }
      for (j = 0; j < N; j++) {     /* Multiply matrices */
         float tmp, bcol[N];
         tmp = 0.0;
         bcol = x[j].btrans;
         for (k = 0; k < N; k++)
            tmp += a[k] * bcol[k];
         c[j] = tmp;
      }
   }
   checksum = 0.0;                  /* Print result */
   [domain row].{
      for (j = 0; j < N; j++)
         checksum += c[j];
   }
   printf ("Check sum is %13.6e\n", checksum);
}
```

Fig. 6. Dataparallel C program to multiply two $N \times N$ matrices.

the unmarked row whose column i value has the greatest magnitude. Once a row is used as a pivot row, it is marked, meaning that it is never again reduced or considered as a pivot row candidate. After the Gaussian elimination step the problem of solving **Ax=b** has been reduced to the problem of solving **Ux=c**, where **U** is an upper triangular matrix.

The back substitution algorithm solves **Ux=c**. Given a system of size N, the algorithm has N-1 iterations. During each iteration the values above the diagonal in another column are reduced to zero, so that at the end of the reduction all elements above the main diagonal are zero. At this point the problem has been reduced to **Dx=d**, where **D** is a diagonal matrix, and the problem can be solved directly.

A Dataparallel C program that generates and solves a system of linear equations appears in Figure 8. The function to generate pseudo-random numbers does not appear; our code is adapted from Pascal functions appearing in [26].

```
/* Warshall's algorithm in Dataparallel C */

#define N      512
#define SQUARE 64
#define ID     (this-x)

domain row { char a[N]; } x[N];

main()
{
   int checksum, j, k;

   [domain row].{
      for (j = 0; j < N; j++) a[j] = 0;  /* Initialize */
      if (!((ID+1) % SQUARE)) a[ID-SQUARE+1] = 1;
      else a[ID+1] = 1;

      for (k = 0; k < N; k++) {          /* Compute closure */
         char row[N];
         row = x[k].a;
         for (j = 0; j < N; j++)
            a[j] |= (a[k] & row[j]);
   }
   checksum = 0;                         /* Print result */
   [domain row].{
      for (j = 0; j < N; j++)
         checksum += a[j];
   }
   printf ("Check sum is %d\n", checksum);
}
```

Fig. 7. Dataparallel C implementation of Warshall's algorithm.

F. Prime Number Sieve

A prime number has exactly two factors: itself and 1. A number is composite if it is not prime. The Sieve of Eratosthenes begins with a list of natural numbers 2, 3, 4, ..., N, then gradually weeds composite numbers from the list by marking multiples of 2, 3, 5, and successive primes. After the multiples of all primes up to $\lfloor\sqrt{N}\rfloor$ have been removed, the numbers remaining on the list are all prime numbers.

Our Dataparallel C implementation of the Sieve of Eratosthenes associates with every virtual processor a block of natural numbers. When a new prime number v is found, every process strikes every vth element of its block, beginning with the first block element that is a multiple of v. The program appears in Figure 9.

G. Results

We present the results of executing the six benchmark programs in Table 1. For each execution we include the speedup of the parallel program on 64 processors over our best

```
/* Dataparallel C program to solve a system of linear equations */

#define       N 512
#define EPSILON 1.0e-08
#define   FALSE 0
#define    TRUE 1
#define      ID (this-r)

double fabs(); float random();
domain system {
   double a[N+1];
   int    id, marked, pivot;
} r[N];

main()
{
   double coeff, temp_array[N+1], temp_element;
   int    i, mono_i, picked, solution;

   [domain system].{
      int k, seed;
      id = ID;
      marked = 0;
      seed = id * id;
      a[N] = 0.0;
      for (i = 0; i < N; i++) {
         a[i] = random(&seed);
         a[N] += i * a[i];
      }
      solution = TRUE;
      for (i = 0; (i < N-1) && solution; i++) {
         double tmp;
         if (!marked)
            if (max_tournament(fabs(a[i]), id, &picked)) {
               marked = 1;     /* Mark pivot row */
               pivot = i;      /* Remember permuted position */
            }
         temp_array   = r[picked].a;
         temp_element = temp_array[i];
         if (fabs(temp_element) < EPSILON) solution = FALSE;
         else if (!marked) {
            tmp = a[i] / temp_element;
            for (k = i; k < N+1; k++)
               a[k] = a[k] - temp_array[k] * tmp; k++;
         };
      }
      if (!marked) pivot = N-1;
      if (solution) {
         for (i = N-1; i >= 0; i--) {
            if (pivot == i) picked = id;
            temp_array = r[picked].a;
            coeff = temp_array[N] / temp_array[i];
            if (pivot < i) a[N] -= coeff * a[i];
         }
      }
   }
   if (solution) {
      for (mono_i = 0; mono_i < N; mono_i++) {
         int temp_int;
         double temp_dbl;
         temp_int = r[mono_i].pivot;
         temp_dbl = r[mono_i].a[N]/r[mono_i].a[temp_int];
         printf ("x[%d] = %10.6f\n", temp_int, temp_dbl);
      }
   } else printf ("No solution\n");
}
```

Fig. 8. Dataparallel C program to solve a system of linear equations.

```
/*
 *      Sieve of Erotosthenes in Dataparallel C
 */

#define NUMPROCS 64
#define VPRATIO  125000
#define N        (NUMPROCS * VPRATIO)
#define FALSE    0
#define TRUE     1
#define ID       (this-x)

domain natural { char prime[VPRATIO];
                 int first, i, localsum, lowvalue; } x[NUMPROCS];

main()
{
   int candidate;   /* Index of prime number candidate */
   int count;

   [domain natural].{
      for (i = 0; i < VPRATIO; i++) prime[i] = TRUE;
      if (!ID) prime[0] = prime[1] = FALSE;
      lowvalue = VPRATIO*ID;
   }
   candidate = 2;                              /* First prime number */
   while (candidate*candidate < N) {
      [domain natural].{
         if (!ID) first = 2 * candidate;    /* Mark multiples of current prime */
         else {
            first = lowvalue % candidate;
            if (first) first = candidate - first;
         }
         for (i = first; i < VPRATIO; i = i + candidate) prime[i] = FALSE;
         if (!ID) {                          /* Find next prime number */
            i = candidate + 1;
            while (prime[i] == FALSE) i = i + 1;
            candidate = i;
         }
      }
   }
   count = 0;

   [domain natural].{
      localsum = 0;
      for (i = 0; i < VPRATIO; i=i+1)
         if (prime[i] == TRUE) localsum++;
      count += localsum;
      if (!ID)
        printf ("There are %d primes less than %d\n", count, N);
   }
}
```

Fig. 9. Dataparallel C program to count prime numbers.

sequential version of the program executing on a single processor. When measuring the execution times of the sequential and parallel programs, we do not include the time needed

Program name	Problem size	Execution time (seconds)	Speedup
pi	400,000	0.41	38.67
relprime	128	0.04	23.46
relprime	256	0.16	30.48
matmult	64	0.42	10.05
matmult	128	1.96	17.82
matmult	256	11.76	29.01*
matmult	512	83.51	31.85*
warshall	64	0.22	9.92
warshall	128	0.79	21.53
warshall	256	4.02	34.55*
warshall	512	26.74	42.31*
linearsystem	128	6.47	2.35
linearsystem	256	24.54	5.22*
linearsystem	512	109.80	9.35*
primes	1,600,000	0.82	33.11*
primes	4,800,000	1.92	46.02*
primes	8,000,000	2.95	50.91*

Table 1. Performance of compiled C* benchmark programs on a 64–processor NCUBE 3200. Speedup figures marked with a * are scaled speedup, because the problem is too large to be solved on a single processor.

to generate the input data or print the results. We have been careful to choose fast sequential algorithms as standards against which our parallel code is compared. To further document the performance of the compiled programs, we have included their absolute execution times.

Some of the programs are too large to solve on a single processor of the NCUBE, due to the 512 Kbyte primary memory size. In these cases the table records the scaled speedup of the parallel algorithm; that is, the time required by the parallel algorithm divided into our best estimate of the time that would be needed to execute the sequential algorithm on a single processor, if it had enough primary memory.

In some cases the speedups are satisfactory, given the parallel algorithm. For example, in the case of the Gaussian elimination program, the phase that reduces the sytem to upper triangular form achieves a speedup of 15.57 over the sequential program, but the back substitution phase, with its small number of computations per communication, actually runs more than 10 times slower than the sequential program, leading to an overall speedup of 9.35 on 64 processors.

In other cases the compiler could generate better code. For example, the compiler does not yet generate efficient code for nearest neighbor communications. Once this functionality has been added, we will be able to improve the speed of matrix multiplication significantly by implementing a systolic version of the algorithm in which the processors form a mesh and pass blocks of A and B to mesh neighbors.

We want to emphasize that these speedups are real results, not projections. As we continue to refine our compiler and learn how to write more efficient code, the performance of the compiled Dataparallel C programs will improve.

In another paper we have shown that Dataparallel C programs can be compiled for efficient execution on shared-memory multiprocessors [27]. Taken together, these results demonstrate that it is possible to execute programs written in a high-level parallel programming language on two radicaly different parallel architectures and achieve reasonable speedups on both machines.

5. Summary

The typical commercial parallel programming language is too low-level and machine dependent. Vendors have been slow to recognize that programming environments can be even more important than processing speed. As a result, there is a large gap between relatively sophisticated hardware and relatively primitive systems software.

Given the dizzying pace at which new parallel computers are released, the gap between hardware and software will continue to increase, unless architecture-independent programming environments are developed. Programmers are far more likely to develop parallel programs, if they know that the programs will not become worthless when the next generation machine appears. The data-parallel model of computation offers programmers an abstract machine with a single flow of control, virtual processors, and a global name space. This model is sufficient to solve many problems in science and engineering.

We have implemented a compiler for the language Dataparallel C. The compiler generates C code suitable for cross-compilation and execution on the NCUBE 3200 multicomputer. The performance of our compiler on a set of benchmark programs demonstrates that data-parallel programs can be compiled for reasonably efficient execution on multicomputers.

[16] A. P. Reeves and D. Bergmark, "Parallel Pascal and the FPS hypercube supercomputer," in *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 385–388, 1987.

[17] R. Miller and Q. Stout, "An introduction to the portable parallel programming language SEYMOUR," in *Thirteenth Annual International Computer Software and Applications Conference*, IEEE Computer Society Press, 1989.

[18] J. R. Rose and G. L. Steele, Jr., "C*: An extended C language for data parallel programming," Tech. Rep. PL 87–5, Thinking Machines Corporation, 1987.

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

[20] M. J. Quinn, P. J. Hatcher, and K. C. Jourdenais, "Compiling C* programs for a hypercube multicomputer," in *Proceedings of the ACM/SIGPLAN PPEALS 1988*, pp. 57–65, 1988.

[21] M. J. Quinn and P. J. Hatcher, "Compiling SIMD programs for MIMD architectures," in *Proceedings of the 1990 International Conference on Computer Languages*, pp. 291–296, IEEE Computer Society Press, 1990.

[22] M. J. Quinn and P. J. Hatcher, "Data-parallel programming on multicomputers," *IEEE Software*, vol. 7, pp. 69–76, Sept. 1990.

[23] L. H. Hamel, P. J. Hatcher, and M. J. Quinn, "An optimizing compiler for a hypercube multicomputer," in *Languages, Compilers and Run-Time Environments for Distributed Memory Machines*, Elsevier, 1991.

[24] G. C. Fox, "What have we learnt from using real parallel machines to solve real problems?," in *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pp. 897–955, ACM Press, 1988.

[25] G. C. Fox, "1989—The first year of the parallel supercomputer," in *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, pp. 1–37, 1989.

[26] R. Sedgewick, *Algorithms*. Reading, MA: Addison-Wesley, 1983.

[27] P. J. Hatcher, M. J. Quinn, A. J. Lapadula, B. K. Seevers, R. J. Anderson, and R. R. Jones, "Data-parallel programming on MIMD computers," Tech. Rep. 90–80–4, Department of Computer Science, Oregon State University, 1990.

References

[1] R. G. Babb, II, *Programming Parallel Processors*. Reading, MA: Addison-Wesley, 1988.

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

[3] P. C. Miller, C. E. S. John, and S. W. Hawkinson, "FPS T Series parallel computer," in *Programming Parallel Processors* (I. R. G. Babb, ed.), pp. 73–91, Reading, MA: Addison-Wesley, 1988.

[4] M. C. Chen, "Very-high-level parallel programming in Crystal," in *Hypercube Multiprocessors 1987*, (Philadelphia, PA), pp. 39–47, SIAM Press, 1987.

[5] D. M. Pase and A. R. Larrabee, "Intel iPSC concurrent computer," in *Programming Parallel Processors* (I. R. G. Babb, ed.), pp. 105–124, Reading, MA: Addison-Wesley, 1988.

[6] E. Paalvast, "The Booster language," Tech. Rep. PL 89–ITI-B-18, Instituut voor Toegepaste Informatica TNO, Delft, The Netherlands, 1989.

[7] E. M. Paalvast and H. J. Sips, "A high-level language for the description of parallel algorithms," in *Parallel Computing 89* (D. J. Evans, G. R. Joubert, and F. J. Peters, eds.), (Amsterdam, The Netherlands), pp. 467–472, Elsevier Science Publishers, 1990.

[8] N. Carriero and D. Gelernter, "Linda in context," *Communications of the ACM*, vol. 32, pp. 444–458, Apr. 1989.

[9] N. Carriero and D. Gelernter, "How to write parallel programs: A guide to the perplexed," *ACM Computing Surveys*, vol. 21, pp. 323–357, Sept. 1989.

[10] M. C. Chen, "Crystal: A synthesis approach to programming parallel machines," in *Hypercube Multiprocessors 1986*, (Philadelphia, PA), pp. 87–107, SIAM Press, 1986.

[11] E. Felten and S. Otto, "Coherent Parallel C," in *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pp. 440–450, ACM Press, 1988.

[12] M. Rosing, R. B. Schnabel, and R. P. Weaver, "Dino: Summary and examples," in *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pp. 312–316, 1988.

[13] M. Rosing, R. B. Schnabel, and R. P. Weaver, "Expressing complex parallel algorithms in DINO," in *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, pp. 553–560, 1989.

[14] C. Koelbel, P. Mehrotra, and J. V. Rosendale, "Supporting shared data structures on distributed memory architectures," in *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 177–186, 1990.

[15] A. P. Reeves, "Parallel Pascal: An extended Pascal for parallel computing," *Journal of Parallel and Distributed Computing*, vol. 1, pp. 64–80, 1984.