Integrating Task and Data Parallelism in UC†

Maneesh Dhagat
6704A Boelter Hall,
Computer Science Department,
University of California at Los Angeles,
Los Angeles, CA 90024
(310)825-4885 (phone)
manu@cs.ucla.edu

Rajive Bagrodia
4531E Boelter Hall,
Computer Science Department,
University of California at Los Angeles,
Los Angeles, CA 90024
(310)825-0956 (phone)
rajive@cs.ucla.edu

Mani Chandy
Computer Science Department,
California Institute of Technology,
Los Angeles, CA 91125
(818)395-6559 (phone)
mani@vlsi.cs.caltech.edu

Abstract

UC is a set-based parallel language that has been implemented on synchronous and asynchronous parallel architectures. Primary constructs of UC include a set data-type, reductions, program composition operators with expression- and block-level synchronization granularity, and data mappings. This paper describes the addition of a channel data-type to the language to support asynchronous composition of data-parallel modules. Operations supported on channels include the standard send, receive and check-status functions; in addition, a channel data type may be mapped or decomposed like other aggregate UC data-structures.

A prototype UC compiler is available for the IBM SP1. The compiler has been used to implement integrated task and data-parallel programs for several applications including FFTs and other numerical applications, biological simulations, and image processing examples. For these applications, we have found a decrease in execution time of as much as 75% with the combined task/data-parallel implementation as compared to the purely data-parallel implementation.

†This research was partially supported under NSF PYI Award No. ASC-9157610, ONR Grant No. N00014-91-J-1605, Rockwell International Award No. L911014, and NSF Center for Research in Parallel Computing Contract CCR-8809615.

1 Introduction

A large number of parallel languages and notations have been designed using the imperative programming paradigm. [3] is a recent survey. Most existing languages adopt either the task or data-parallel paradigm. Task-parallel languages provide constructs to create and destroy asynchronous threads and to allow the asynchronous threads to communicate and synchronize as needed. As each thread typically operates on its private data, the data space of the program must be explicitly sub-divided among the multiple threads. Data-parallel languages use globally addressable memory together with a synchronous programming model where the computation is implicitly synchronized at some typically pre-determined level of granularity. The global data may be optionally mapped on the memory hierarchy of the parallel architecture using language directives.

Paradigm Integration Although data parallelism has been applied successfully to solve a large number of scientific problems, a number of applications (e.g. biological simulations and global climate models) may be more naturally and sometimes more efficiently designed using both forms of parallelism. Also for some data-parallel applications that can be programmed naturally using data parallelism, the computation granularity of an application or the communication latency of an architecture may favor a decomposition of the program into a task-parallel collection of data-parallel components.

Synchronization Granularity The synchronization granularity specified by a language determines the computation model presented to the programmer and the efficiency with which the languages can be implemented on asynchronous architectures. Thus, SIMD languages that are synchronized at the expression level have simple semantics, as the programmer has access to the correct global state of the program before executing any operation. However, implementing such a strict synchronous notation on an asynchronous distributed memory architecture can, in general, be expensive as a barrier may potentially be required before the execution of every operation. In contrast, a language with a weaker synchronization model forces the programmer to restrict remote data access to specific points in their code. Although this increases the programmer's burden of maintaining a coherent view of the shared data, an implementation of this model on an asynchronous architecture requires a smaller number of barrier synchronizations as compared with the previous model. Reducing barrier synchronizations improves execution efficiency by reducing both the blocking and communication times. Although optimizing compilers minimize the number of barriers for SIMD languages [14, 20], the optimizations are typically applicable to programs where array subscripts expressions may be evaluated at compile-time.

Approach We propose to integrate task and data parallelism in an extension of C called UC. In particular, it is shown that the addition of a single *channel* data type to a data-parallel language is sufficient to exhibit a wide variety of task-parallel behaviors. Additionally, the parallel composition operators provided by UC allow a programmer to syntactically specify the synchronization granularity of a set of statements to be at the expression-level, at the level of compound statements, or at the function level where multiple threads are synchronized only at the point of call and return and the multiple instances of the function execute asynchronously. Supporting synchronization at multiple levels of granularity promotes an **iterative approach** to program design where an initial program can be designed using expression-level synchronization and can subsequently be refined for efficient implementation on asynchronous architectures.

UC has been used to design a diverse set of applications that include search algorithms, graph algorithms, image processing, computational fluid dynamics, and computational mathematics. UC has been implemented on synchronous architectures like the CM-2 and on asynchronous architectures like the IBM SP1. The next section is a brief description of the language. Section 3 discusses the role of channels in specifying task parallelism. In section 4, we present several examples and their performance. Implementation of UC on the IBM SP1 is briefly discussed in section 5. Section 6 presents related work and section 7 is the conclusion.

2 Sets and Quantification

The central concepts of the UC notation are the well-known mathematical concepts of sets and quantification which have been used in other programming languages [25, 5, 23]. This section gives a brief description of concepts relevant to the proposal; a complete description can be found in [1].

2.1 Set and Range

Although UC notation supports specification of general relations and sets[1], for brevity we restrict our attention in this paper to dense sets, also referred to as ranges. A *range* is a special case of a set of integers with constant lower and upper bounds on values of members of the set; the bounds are specified at the point of declaration of the range. The declaration of a *range* is of the form:

range `I:i = {0..N-1};`

which declares `I` to be a *range*, and `i` to be a dummy variable that runs over `I`, where all members of `I` lie in the closed interval `[0, N-1]`. Set operators may also be applied to ranges with the restriction that all elements of a range must always lie within the constant bounds specified in its declaration.

2.2 Quantification

A quantification is of the form [5]:

$$\langle op\ \mathtt{i} \in \mathtt{I} : \mathtt{b(i)} : \mathtt{s(i)} \rangle$$

where *op* is an associative, commutative operator, `I` is a range, `i` is a dummy variable, `b(i)` is a boolean expression, and `s(i)` is an expression or statement. The notation supports addition, multiplication, maximum, minimum, universal quantification, and existential quantification over sets. For example,

$$\langle \forall\ \mathtt{i} \in \mathtt{I} : \mathtt{0} \leq \mathtt{i} < \mathtt{3} : \mathtt{a[i]} = \mathtt{0} \rangle$$

means that for all `i` in set `I` such that $0 \leq \mathtt{i} < 3$, the equation `a[i] = 0` holds. Likewise,

$$\langle +\ \mathtt{i} \in \mathtt{I} : \mathtt{0} \leq \mathtt{i} < \mathtt{3} : \mathtt{a[i]} \rangle$$

is the sum of `a[i]` over all `i` in set `I`, where $0 \leq \mathtt{i} < 3$. The notation also has a nondeterministic operator that returns an arbitrary member of a set[1]. Quantification is extended to control-flow operators that are associative and commutative. For example,

$$\langle \mathbf{par}\ \mathtt{i} \in \mathtt{I} : \mathtt{0} \leq \mathtt{i} < \mathtt{3} : \mathtt{s(i)} \rangle$$

where `s(i)` is an assignment statement with parameter `i`, is the parallel execution of `s(i)`, for all `i` in set `I`, where $0 \leq \mathtt{i} < 3$. The **par** operator is discussed later. Operations on subsets are obtained using operator **select**:

$$\langle \mathbf{select}\ \mathtt{i} \in \mathtt{I} : \mathtt{b(i)} : \mathtt{e(i)} \rangle \ = \ \{\mathtt{e(i)} | (\mathtt{i} \in \mathtt{I}) \wedge \mathtt{b(i)}\}$$

[1] This operator is denoted with a comma (,).

UC Notation In programs, we use the keyword **in** for $\in$, and the logical *and* operator `&&` for $\forall$, the logical *or* operator || for $\exists$, $<$ for minimum, and $>$ for maximum. We use:

`$`*op*`(i` **in** `I  st  (b(i)) s(i))`   for $\langle op\ \mathtt{i} \in \mathtt{I}\mathtt{:b(i):s(i)}\rangle$

where *op* is an arithmetic, boolean, or selection (min, max, select) operator. As a syntactic convenience, if the dummy element has been declared together with the corresponding range, it may be omitted in the statement. For instance, given the declaration

range `I:i = {0..N-1};`

the above expression may be written as `$`*op*`(I st (b(i)) s(i))` and if the boolean expression `b(i)` is true for all `i`, it can be omitted, as in `$`*op*`(I; s(i))`.

If *op* is a control-flow operator we use:

op`(i` **in** `I)  st  (b(i)) s(i);`   for $\langle op\ \mathtt{i} \in \mathtt{I}\mathtt{:b(i):s(i)}\rangle$

Again, if the dummy variable `i` has been declared with the range `I`, the above can be rewritten as *op*`(I) st (b(i)) s(i);`. If the boolean expression `b(i)` is true for all `i`, it can be omitted, as in *op*`(I) s(i);`.

Lastly, note that the boolean expression `b(i)` in the quantification is used to optionally select a subset of set `I` for the corresponding operation; `b(i)` must be evaluated for all `i` *prior to* the execution of any `s(i)`.

Example: The UC program in figure 1 implements an algorithm to calculate prime numbers using the sieve method of Eratosthenes. It illustrates the use of quantification. Given a sequence of integers starting with 2, at each iteration of the loop the algorithm is as follows: compute the smallest integer in the sequence and mark it as prime. Then remove all multiples of it from the sequence. This is repeated until the sequence is empty.

```
#define N 1024
int main (void) {
   int prime[N];           /* initialized to 0 */
   range I:i = {2..N-1};   /* assume N > 2 */
   int nextp = 2;

   while (nextp <= N) {
      prime[nextp] = 1;                     /* prime[i] is 1 if i is prime */
      I = $select(I st (i%nextp != 0) i);   /* I = {i | (i mod nextp) ≠ 0 } */
      nextp = $<(I; i);                     /* nextp = minimum element of I */
   }
}
```

Figure 1: UC program to implement the sieve method of Eratasthenes.

2.3 Asynchronous Composition

The keyword **arb** (for arbitrary) is the operator for composing programs asynchronously. The operation s **arb** t, where s and t are statements, specifies concurrent execution of s and t using *arbitrary interleaving.* The operator **arb** is associative and commutative.

For example, { `a = y;` } **arb** { `b = z;` } assigns the values of `y` and `z` to `a` and `b` respectively, if the memory locations of `a` and `b` are different from each other, and different from those of `y` and `z`. However, the effect of the execution of {`z = y;`} **arb** {`y = z;`} is not specified, because the atomicity and interleaving of operations between the concurrently executing statements `z = y;` and `y = z;` are not specified. All of the following interleavings are permissible and each leads to a different result: (1) `z = y; y = z;` , (2) `y = z; z = y;` , and (3) `z,y = y,z;`.

Example 1: A program to transpose a matrix `a` is:

```
range I:i = {0..N-1}, J:j = I;
arb (I,J) tmp[i][j] = a[i][j];
arb (I,J) a[i][j] = tmp[j][i];
```

This program fragment consists of sequential composition of two statements, each of which is an asynchronous composition of N^2 threads each consisting of two assignments[2]. The first copies matrix `a` to matrix `tmp`, and the second assigns the transpose of `tmp` to `a`. On the other hand, the result of the fragment

```
arb (I,J) a[i][j] = a[j][i];
```

is not specified because the atomicity and interleaving of each of the N^2 statements is not specified. Sequential composition and parallel asynchronous composition do not commute. For instance, the following fragment has a different effect than the fragment in example 1, since here no barrier exists between the two statements.

Example 2:

```
range I:i = {0..N-1}, J:j = I;
arb (I,J) {
   tmp[i][j] = a[i][j];
   a[i][j] = tmp[j][i];
}
```

2.4 Parallel Assignment Composition

Assignment statements can be composed using the parallel composition operator **par**. The execution of { `x =` e; } **par** { `y =` f; } is as follows: the expressions e and f are first evaluated, and then their values are assigned to `x` and `y` in parallel. Thus the values of `x` and `y` after the execution of this is the same as their values after the execution of:

```
{ tmp_x = e;} arb { tmp_y = f;} ;
/* barrier */
{ x = tmp_x;} arb { y = tmp_y;} ;
```

where `tmp_x` and `tmp_y` are distinct fresh variables not named in expressions e and f.

The parallel composition operator is associative and commutative. We use the convention that the parallel composition operator has higher precedence than the asynchronous composition operator. Parallel composition is a convenience rather than a necessity, since parallel composition can be implemented in terms of **arb** and sequential composition.

We adopt the convention that the body of a **par** statement can be a block consisting of variable declarations followed by a sequence of assignment statements. Each assignment statement in the block is evaluated synchronously. For instance, the following fragment

```
par (I,J) {
   a[i][j] = a[j][i];
   a[i][j] = a[(i+1)%N][j];
}
```

is equivalent to

```
par (I,J) a[i][j] = a[j][i];
par (I,J) a[i][j] = a[(i+1)%N][j];
```

[2] We assume that `a` and `tmp` are N^2-size matrices.

A **par** statement may contain multiple clauses, which are executed asynchronously. For instance, the following fragment sets the even elements of `a` to 0 and the odd elements to 1.

```
par (I)
    st (i%2 == 0) a[i] = 0;
    st (i%2 != 0) a[i] = 1;
```

The **par** statement may contain nested **if** statements. The semantics of such a statement is identical to that of a **par** statement with multiple clauses, where each clause is equivalent to an arm of the corresponding **if** statement.

```
par (I) {                                        par (I)
    if (b[i]) s1;        is equivalent to            st (b[i]) s1;
    else s2;                                         st (!b[i]) s2;
}
```

The **par** quantification may include iterative constructs like the **while** statement:

```
par (I) st (b[i])                                while ($||(I; b[i]&&c[i]))
    while (c[i]) s;      is equivalent to            par (I) st (b[i]&&c[i]) s;
```

For every iteration, the loop condition is evaluated for all elements in `I`. The loop is terminated if the condition evaluates to false for every element; otherwise statement *s* is executed synchronously for every enabled element. The meaning of other iterative constructs is similar.

If a **par** statement includes a function call, multiple instances of the function execute asynchronously. The program fragment

```
                                                 arb (I) {
                                                     copy_x[i] = x[(i+1)%N];
par (I)                  is equivalent to            copy_y[i] = y[i];
    f(x[(i+1)%N], y[i]);                         };
                                                 arb (I) f(copy_x[i], copy_y[i]);
```

Note that asynchronous execution due to **arb**, **par** **st**-clauses and parallel function calls will cause non-determinism if two threads *interfere*. Two threads are said to interfere if a variable modified in one thread is referenced by another thread. A parallel function can cause interference if it modifies shared variables. Sharing of variables by parallel instances may occur either by sharing global variables or by passing pointers or array parameters. A compiler may be able perform a conservative check and issue an appropriate warning if interference of any of the preceding forms occurs.

Example: The UC program in figure 2 illustrates the use of set operations on ranges, reductions, and the **par** statement. It implements the LU decomposition phase of the Gaussian elimination scheme for solving a set of linear equations of the form $Ax = b$, where A is a square matrix of coefficients, and x and b are column vectors of unknowns and constants, respectively. The two reductions inside the **for**-loop find the position of the maximum element in the $\mathtt{z}^{th}$ column of `A`. The **par** statement that follows performs a row swap. The set delete operations, indicated with the minus operator in UC, are used to limit the parallelism is the subsequent **par** statements and the reductions.

2.5 Data Distribution

UC provides data mapping constructs to describe various types of data distributions[2]; these data mappings are similar to the mappings provided by HPF. Three types of data mappings are supported: **perm** to align a data structure with respect to another; **fold** to decompose aggregate data structures

```
int main(void) {
   double A[N][N], tmp[N];
   range L:l = {0..N-1}, I:i=L, J:j=L;
   int z, pivot;
   double max;

   for (z = 0; z < N; z++) {
      max = $>(I; fabs(A[i][z]));
      pivot = $<(I st (max == fabs(A[i][z])) i);    /* good pivot chosen */

      if (pivot < N)
         par (L) {
            tmp[l] = A[z][l];
            A[z][l] = A[pivot][l];
            A[pivot][l] = tmp[l];
         }
      else break;

      I = I - z;  J = J - z;
      par (I) A[i][z] /= A[z][z];    /* Calculation of Multiplier Column */
      par (I,J)  A[i][j] -= A[i][z] * A[z][j];     /* Elimination */
   }
}
```

Figure 2: UC program to implement the LU decomposition phase of Gaussian elimination.

(e.g. to specify block partitioning for arrays), and **copy** to replicate parts of a data structure. Any declaration block of a UC program may contain a data distribution statement, allowing UC mappings to be dynamic; data structures can be redistributed in deeper scopes. In the interest of brevity, we demonstrate some simple uses of mappings below:

```
int a[N], b[N], c[N], d[N], e[N];
perm (I) a[i] :- b[(i+2)%N];      /* alignment */
copy (I,J) c[i] :- c[i][j];       /* replication */
fold (I) {
   d[i] :- d[i%B];                /* block partioning */
   e[i] :- e[i/C];                /* cyclic partioning */
}
```

In the permute mapping the $\mathtt{i}^{th}$ element of array a is aligned with the $(\mathtt{i}+2)^{th}$ element of array b. The copy mapping replicates array c along an additional axis. The fold mapping puts size-B blocks of array d on each processor and cyclically places elements of array e on C processors.

Multiple Synchronization Granularity UC constructs allow programmers to specify program synchronization at different levels of granularity: the **par** construct can be used to specify fine-grain synchronization at the expression level; coarser-grain synchronization at the compound-statement level can specified using **par st**-clauses, **arb** or parallel function calls. This provides the programmer considerable flexibility: an application can initially be designed as a data-parallel program using the expression level synchronization of a **par** statement and can subsequently be refined by iteratively weakening the synchronization granularity to improve its performance. As a final step, the program can be refined into a task-parallel program composed of two or more data-parallel subprograms.

3 Task Parallelism

Although the asynchronous constructs provided in UC (such as **par st**-clauses, **arb**, and parallel function calls) provide coarse granularity synchronization, their semantics are well-defined only if a variable modified in one thread is not accessed in concurrent threads. This is overly restrictive and we need a construct to specify a weaker form of synchronization within these asynchronous constructs. We use message-passing channels for this purpose.

3.1 Channel Variables

A *channel* type is a queue of messages. The messages in the queue can be of arbitrary type. The primary operations supported on a channel include depositing a message and retrieving a message. The two operations are implemented respectively by the **send** and **receive** functions. The function **send**(*c,expr*) is an asynchronous send that deposits a message containing *expr* on channel *c*. The function **receive**(*c*) provides a blocking receive which retrieves items from channel *c*. In addition to the send and receive operations, two other boolean functions, **empty**(*c*) and **full**(*c*) are supported with their standard semantics.

A channel is a type and can be used in most places where types are used. For instance, it is possible to declare an array of channels. A channel can be mapped and aligned using data mappings in exactly the same way as arrays. Channels are also subject to the same scope rules as other variables in the program.

Each channel is associated with a buffer size. If unspecified, the buffer size of a channel is assumed to be infinite. A finite size may however be specified in its declaration as in:

channel `x[N][N]` **size** *expr*;

The preceding fragment indicates that all channels in variable `x` have a maximum capacity of *expr* bytes. If a channel is declared to have an infinite capacity, the send operation is always non-blocking (limited by the total local memory). In this case, all data are buffered at the processor where the channel has been mapped. If a channel is declared with a finite capacity, the underlying implementation will ensure that a send operation is blocked until the buffer has sufficient capacity to store the data.

3.2 Semantics

As the send and receive operations on a channel variable are provided as function calls, they can occur anywhere in a UC program where they do not violate other syntactic restrictions. The following table presents the semantics of channel operations within **par** and **arb** statements. Both scalar channel variables (**channel** `sx;`) and array channel variables (**channel** `ax[N];`) are considered:

assignment	semantics
par`(I)` **send**`(sx, a[i]);`	illegal
par`(I) a[i]=`**receive**`(sx);`	value dequeued from `sx` is broadcast to all `a[i]`
arb`(I)` **send**`(sx, a[i]);`	in arbitrary order, each `a[i]` is enqueued on `sx`
arb`(I) a[i]=`**receive**`(sx);`	in arbitrary order, each `a[i]` is assigned a dequeued value from `sx`
par`(I)` **send**`(ax[i], val);`	copy of `val` is enqueued on each `ax[i]`
par`(I) val=`**receive**`(ax[i]);`	illegal
arb`(I)` **send**`(ax[i], val);`	copy of `val` is enqueued on each `ax[i]`
arb`(I) val=`**receive**`(ax[i]);`	each `ax[i]` is dequeued once and values are assigned to `val` in arbitrary order

The **par** statements are illegal because they allow multiple values (`a[i]`, **receive**`(ax[i])`) to be written to a single variable (`sx`, `val`).

Channels may be used inside loops; for instance, in the following fragments the loop with the **send** will deposit successive elements from array `a` in channel `sx` and the loop with the **receive** will remove elements (as they become available) from channel `sx` and place them in successive elements of array `b`:

```
for (i=0;i<N;i++)                  for (i=0;i<N;i++)
   send(sx, a[i]);                    b[i]=receive(sx);
```

Receive operations may also be used in reduction expressions as in the following:

```
c = $+(I; receive(ax[i]));
d = $,(I; receive(ax[i]));
```

The first reduction removes an element from each channel and returns their sum in `c`; the operation suspends until all channels are non-empty. The second reduction will non-deterministically select one of the channels, remove an element and return it in variable `d`; the operation will suspend while the selected channel is empty.

As channel variables are untyped the number of bytes dequeued by a **receive** operation depends on the type of the destination. For example,

```
struct my_struct { int x; float y; } my_var;
my_var =  receive(sx);
```

dequeues **sizeof**(`my_var`) bytes from `sx` and assigns it to `my_var`.

Channel variables may be passed as arguments to functions. Like arrays they are passed by reference.

Example: The UC program in figure 3 illustrates the use of channels in connecting data-parallel modules. The program implements iterated polynomial multiplication which consists of three data-parallel subproblems: two modules which perform forward FFT's on two polynomials, and one module which takes their output, performs pointwise multiplication on them and calculates the inverse FFT of the product. Polynomial coefficients, which are complex numbers, are represented in the program by two arrays - for the real and imaginary parts. The three **st**-clauses of the main **par** execute asynchronously. Routine `fft`, not shown for brevity, is a data-parallel program that reads in a polynomial and performs a FFT on it. Similarly, routines `pw_mult` and `inv_fft` implement pointwise multiplication and inverse FFT respectively. Channel variables `a_x_{r,i}` and `b_x_{r,i}` are used by the first two threads to communicate with the third thread. Note that, although channel variables are being modified in more than one thread, due to channel variable semantics determinism is preserved.

3.3 Discussion

Our view of channels has advantages and disadvantages. The advantage is that the programmer can write code with less synchronization cost by using channels within asynchronous constructs. Channels also allow the programmer to separate the storing of data from the point where it is needed. This allows the programmer to improve performance of the code as data transmission and computation can proceed concurrently. Finally, deterministic execution can be guaranteed for a UC program by ensuring that channel operations are not used within an **arb** statement or in a reduction.

A disadvantage is that as channels are universally addressable, inefficient programs can be written. If the mapping is mis-specified, it may take additional hops for a message to get where it is needed. The programmer must take special care in mapping channel variables. Also as **send** and **receive** operations are provided by the language, the programmer must take additional care to avoid writing a deadlock into the program by mistake.

For programs whose communication patterns are known at compile time, it is possible to use compiler transformations to replace global synchronizations by point to point communications

```
float a_r[N], a_i[N], b_r[N], b_i[N], c_r[N], c_i[N], l;
float tmp_a_r[N], tmp_a_i[N], tmp_b_r[N], tmp_b_i[N];
channel a_x_r[N], a_x_i[N], b_x_r[N], b_x_i[N];
range T:t = {0..2}, I:i = {0..N-1};

int main(void) {
   par (T)
      st (t == 0)
         for (l = 0; l <= L; l++) {
            fft(a_r, a_i);
            arb (I) { send(a_x_r[i], a_r[i]);  send(a_x_i[i], a_i[i]); }
         }
      st (t == 1)
         for (l = 0; l <= L; l++) {
            fft(b_r, b_i);
            arb (I) { send(b_x_r[i], b_r[i]);  send(b_x_i[i], b_i[i]); }
         }
      st (t == 2)
         for (l = 0; l <= L; l++) {
            arb (I) {
               tmp_a_r[i] = receive (a_x_r[i]);  tmp_a_i[i] = receive (a_x_i[i]);
               tmp_b_r[i] = receive (b_x_r[i]);  tmp_b_i[i] = receive (b_x_i[i]);
            }
            pw_mult(c_r, c_i, tmp_a_r, tmp_a_i, tmp_b_r, tmp_b_i);
            inv_fft(c_r, c_i);
         }
}
```

Figure 3: Combined task/data-parallel UC program to implement iterated polynomial multiplication using three FFT's.

using channel variables[20, 14]. For programs with dynamic communication structures however, programmer-specified transformations may be significantly more efficient.

4 Performance Studies

Several data-parallel and combined task/data-parallel examples were coded in UC. These are described below and performance results are presented. UC programs were compiled into C with calls to IBM SP1's message passing and collective communications library, and were run on a 24-node IBM SP1 using the high-performance switch with the lightspeed communications protocol. All parallel arrays in the following programs were block mapped along the first axis; all other axes were stored serially.

4.1 Small Data-Parallel Examples

A number of small examples were coded using only the data-parallel portion of the compiler. The speedup obtained with these examples was computed as a function of the number of processors and the size of the dataset. Among the algorithms coded in UC is a systolic matrix multiplication algorithm due to Cannon[17] shown is figure 4. This algorithm uses barriers to ensure that dependencies are not violated. The performance is shown in figure 5(a). Figure 5(b) shows that slightly better performance can be achieved by using channel communication to replace the barriers, as communicating pairs are known at compile-time. Additionally the Gaussian elimination[8] program discussed earlier (figure 2) and an all-pairs shortest path algorithm (figure 6) were implemented. Their performance graphs are shown in figure 7(a) and figure 7(b)[3].

```
int main (void) {
   int a[N][N], b[N][N], c[N][N], k;
   range I:i = {0..N-1}, J:j=I;

   par (I,J) {
      a[i][j] = a[i][(i+j)%N];          /* Alignment Phase */
      b[i][j] = b[(i+j)%N][j];

      c[i][j] += a[i][j] * b[i][j];     /* Multiplication Phase */
      for (k = 1; k < N; k++) {
         a[i][j] = a[i][(j+1)%N];
         b[i][j] = b[(i+1)%N][j];
         c[i][j] += a[i][j] * b[i][j];
      }

      /* Re-alignment Phase -- omitted */
} }
```

Figure 4: UC program to implement systolic matrix multiplication.

4.2 Combined Task/Data-Parallel Examples

Three applications that could benefit from an integrated task/data-parallel approach were programmed in UC. Each consists of two or more subproblems that form stages of a pipeline. The stages execute concurrently as data-parallel subprograms in a task-parallel program.

Polynomial Multiplication (FFT) This application was discussed briefly earlier. We treat it in more detail here. The program is a pipelined computation that performs a sequence of polynomial multiplications using three discrete Fourier transforms. The transformations are performed using a data-parallel version of the fast Fourier transform algorithm. The task is to multiply pairs of polynomials of degree $n-1$, where n is a power of 2. Input is a sequence of pairs of polynomials, each polynomial is represented by its n coefficients. Output is a sequence of product polynomials of degree $2n-1$, each represented by $2n$ coefficients. Finding the product of two polynomials can be broken down into subproblems as follows:

1. Extend the first polynomial to degree $2n-1$ by padding with zeros. Convert it to point-value representation by evaluating this extended polynomial at the $2n$ $2n^{th}$ roots of unity. These can be computed efficiently using FFT.

2. Do the same as above the second polynomial.

3. Multiply the two resulting $2n$-tuples of complex numbers element-wise. This represents the product polynomial evaluated at the $2n$ roots of unity. Convert the tuple to coefficient form by applying an inverse FFT.

[3] All matrices in the examples are of size N^2. For Gaussian Elimination the coefficient matrix was chosen such that pivoting would be required at every iteration.

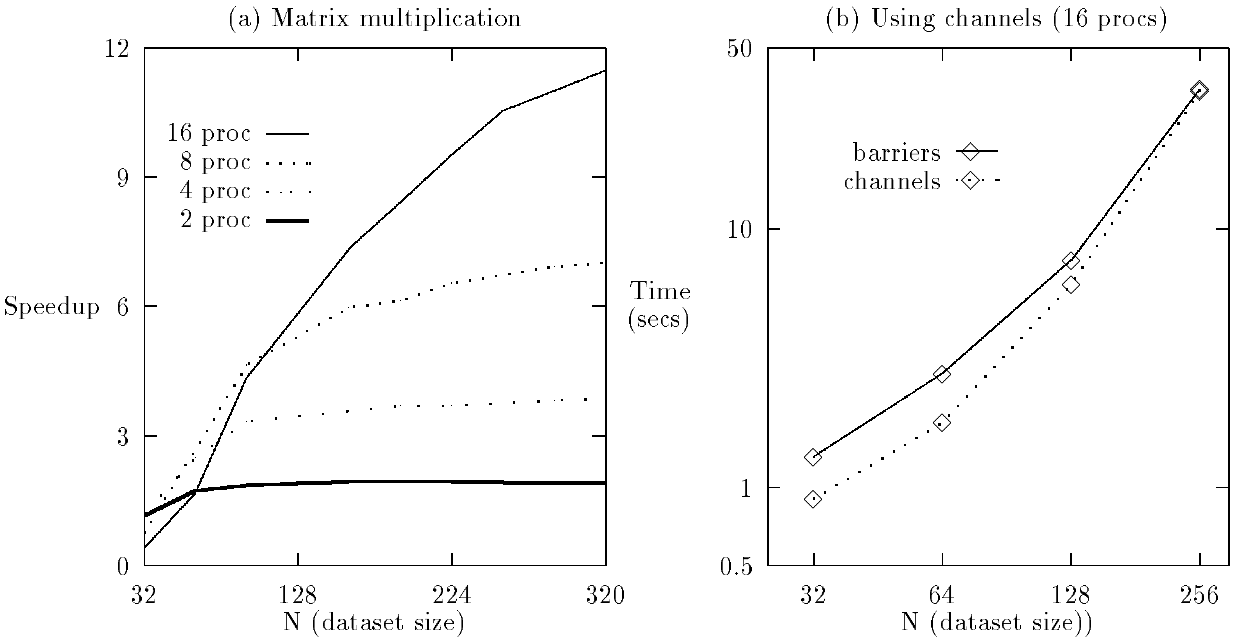

Figure 5: (a) Performance of systolic matrix multiplication on IBM SP1 and (b) using channel communication to reduce barriers.

For brevity, the details of computing FFT in parallel are not presented here; a recent reference is [8].

The UC program for the combined task/data-parallel implementation was presented earlier in figure 3. The purely data-parallel program is shown in figure 8. A complete listing of both programs in presented in the appendix. In the data-parallel implementation each subproblem executes on all processors, whereas in the combined implementation each subproblem executes on a third of the processors. The program for each implementation multiplies 20 polynomials. The execution time for the two implementations was plotted as function of the number of processors[4]. These curves are shown in figure 9 for three different dataset sizes.

The graph shows that the combined implementation performs better for these dataset sizes. This can be explained as follows. Each FFT module is communication intensive, performing a series of all-to-all communications. The communication overhead for each module dominates the local computation. FFT modules benefit from being executed at a smaller number of processors as they send fewer (although longer) messages. The purely data-parallel implementation performs each of three FFT's on all processors, whereas the combined implementation performs each FFT on a third of the processors. The combined implementation shows better performance as it benefits from pipelining of FFT modules.

Pattern Recognition Moments constitute an important set of parameters for image analysis; for example, the lower-order moments indicate the location, size, and orientation of the object in the

[4] The combined code executes on numbers of processors that are multiples of three, whereas for the purely data-parallel code the number of processors is a multiple of two.

```
int main (void) {
    int k, dist[N][N]; /* initialized appropriately */
    range I:i = {0..N-1}, J:j = I;
    for (k = 0; k < N; k++) {
        /* ⟨∀i,j : i ∈ I ∧ j ∈ J : dist[i][j] = min(dist[i][k] + dist[k][j], dist[i][j]);⟩ */
        par (I,J)
            dist[i][j] = min(dist[i][k]+dist[k][j], dist[i][j]);
    }
}
```

Figure 6: UC program to calculate the length of the shortest path between each pair of nodes in a graph.

image[6]. The $(K, L)^{th}$ moment of an image is defined by

$$m_{KL} = \sum_{i=1}^{N} i^K \sum_{j=1}^{N} j^L x_{ij}$$

where x_{ij} represents the gray-scale level at pixel (i, j) in an image of size N^2. In a straight-forward implementation, this computation can be divided into two stages. The first stage, for a particular value L, performs the inner summation over the j^{th} axis. If this axis is stored serially within each processor, this summation loop can be executed locally. The matrix x is then transposed and given for processing to the second stage. The second stage performs the outer sum-reduction over the i^{th} axis which is now stored serially. One pass over the two stages calculates m_{kL} for $k = 1..N$. The two stages are looped over $L = 1..N$ to calculate all moments.

In the purely data-parallel implementation each stage executes on all processors. In the combined task/data-parallel implementation each stage uses half the processors. The execution time of the two implementations was plotted as a function of the problem size, N, where the input image is a matrix of size N^2. In figure 10 the performance of the two implementations is compared for two different machine sizes: 16 processors and 8 processors.

The graphs show that the combined implementation performs better for smaller problem sizes (N $\leq$ 256), whereas the purely data-parallel implementation performs better for larger problems sizes. It is informative to analyze the results. In the combined implementation the transpose operation is merged with the transfer of data between the two stages. The purely data-parallel implementation performs the matrix transpose over twice as many processors as the combined implementation. The latter sends fewer messages, although they are longer. For smaller problem sizes communication overhead dominates computation cost and hence the combined implementation performs better. For larger problem sizes, local computation dominates communication overhead and the purely data-parallel implementation performs better.

Mosquito Control Simulation Biological simulation is a class of problems that can benefit from an integrated task/data-parallel approach. Computer simulation of mosquito control is useful in determining optimal insecticide treatment schedules and in minimizing development of resistance in mosquitos. A biological model for the *Culex quinquefasciatus* mosquitos for simulation is described in [12]. Parallel computer simulations of this model have been used successfully to control mosquito populations in Orange County, California[11]. This model was coded in UC and executed on the IBM SP1. The program consists of two coupled subprograms: the weather simulation and the population simulation.

The simulation environment consists of a grid of locations representing a geographical map. Each location contains 4 numbers representing the population sizes of eggs, larvae, pupae, and

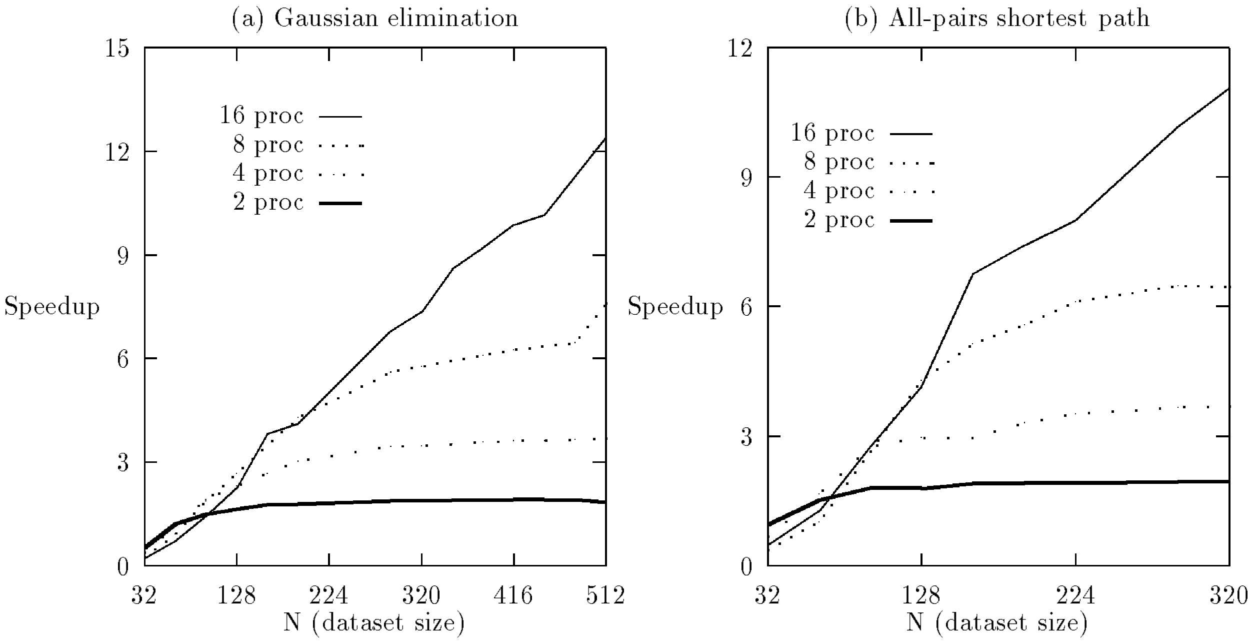

Figure 7: Performance of (a) Gaussian elimination and (a) all-pairs shortest path on IBM SP1.

adults resident in that location. Each location is close to one of 5 weather stations. The simulation is divided into two phases: weather and population calculations. The weather module reads the daily air temperature, and precipitation data for each weather station. From these it computes the water temperature, humidity, days since rain, development rates for eggs, larvae, pupae, and the adult survival rates. These rates are used by the population module to age, reproduce, and kill off mosquito populations.

In the purely data-parallel implementation a copy of the weather simulation is replicated on each processor. This is followed by the population simulation on the locally stored portion of map. In the combined implementation, the weather simulation is executed on a single processor and the population simulation is distributed over the rest. Each implementation executes for 30 simulation days. The execution time is plotted as a function of the number of processors[5]. These curves are shown in figure 11 for three different dataset (geographical map) sizes.

The performance can be explained as follows. Initially (for 2 processors), the purely data-parallel implementation performs better. This is because the population model, which is the predominant computation, executes on both processors in the purely data-parallel implementation, whereas in the combined implementation it executes on only half the machine. With increasing number of processors, the effect of the extra processor used for the weather simulation becomes less significant, and the combined implementation starts to perform better. In the combined implementation each processor does not have to repeat the weather calculations – it simply gets the results from the single processor executing the weather model. However, as processors increase further, the combined implementation again becomes worse due to the increasing overhead of broadcasting the weather information to each processor.

[5]The purely data-parallel implementation executes on a number of processors which is a power of two, whereas the combined implementation executes on a number processors which is a power of two plus an additional processor for the weather model.

```
float a_r[N], a_i[N], b_r[N], b_i[N], c_r[N], c_i[N], l;

int main(void) {
   for (l = 0; l <= L; l++) {
      fft(a_r, a_i);
      fft(b_r, b_i);
      pw_mult(c_r, c_i, a_r, a_i, b_r, b_i);
      inv_fft(c_r, c_i);
   }
}
```

Figure 8: Purely data-parallel UC program to implement iterated polynomial multiplication using three FFT's.

5 Compilation

In this section we discuss compilation of UC programs. To initiate this study we have implemented a prototype compiler for the data-parallel portion of UC on the IBM SP1. The compiler is implemented as a source-to-source transformation which converts data-parallel UC notation into C with calls to EUI-H[16], a MPI-like message passing library which utilizes IBM SP1's high-performance switch. This target code with minor annotations by hand can approximate the target code of an integrated task/data-parallel compiler. Below we discuss the data-parallel implementation and issues related to an integrated task/data-parallel compiler.

5.1 Data-parallel Implementation

Array variables in UC are classified as serial or parallel, determined by their usage. Parallel arrays are distributed over the processors and serial variables are replicated on each processor. By default, parallel arrays are distributed by blocking along the first axis and keeping rest of the axes serial. This distribution can be changed by using mapping directives. Any serial C computation operating solely on serial variables is replicated on each processor. Additionally, the compiler supports the following six categories of parallel operations: guarded local assignment to parallel arrays, assignment to serial variables, reduction on a parallel array, spread, send to a channel, and receive on a channel. Together these constructs describe a sufficiently complete subset of UC such that any UC program can be translated into the subset using simple source-to-source transformations.

Local assignments to parallel arrays are simply converted into assignment loops over the local portion of the array. Assignment to a serial variable from a parallel array is followed by a broadcast to maintain consistency. Reduction is performed over a single axis at a time. If the axis is stored serially, an accumulation loop over the local portion of the array is performed. If the axis is distributed, the accumulation loop is followed by a reduction over the processors. Spreads are implemented correspondingly. General purpose communication is implemented as channel sends and receives. Each channel variable becomes a buffer implementing a queue. Communication takes place by sending to a non-local channel variable. Message aggregation is used to improve communication performance.

The prototype compiler only supports **arb** clauses. Any **par** clause can be rewritten as a series of **arb** clauses using temporaries. A barrier is generated at the end of each **arb** clause.

5.2 An Integrated Compiler

The target code produced by the data-parallel compiler can with minor hand-compilation be made to approximate the code generated by an integrated task/data-parallel compiler. This technique was

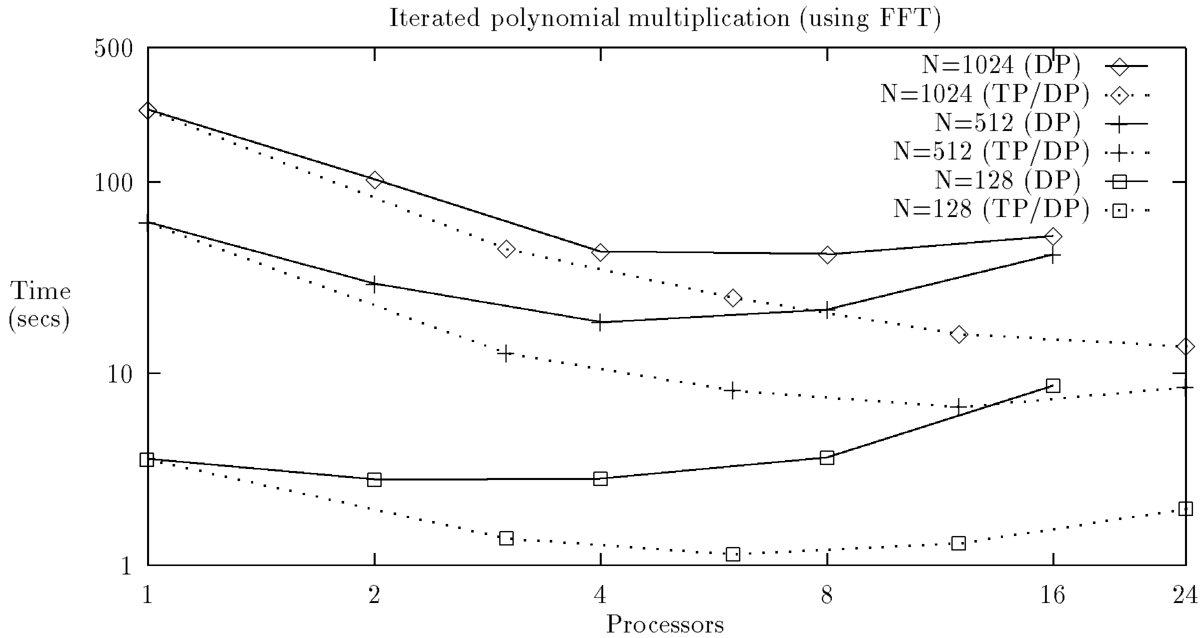

Figure 9: Performance of iterated polynomial multiplication using three FFT's on IBM SP1.

used to compile three applications which were described in detail in section 4.2. Here we describe the hand annotations needed and identify some implementation problems which require more general solutions that are the topic of our current research.

Thread allocation Parallel threads are distributed equally among the processors. If there are fewer threads than processor, a thread may occupy several processors. Similarly, if threads out number the processors, many threads may execute on each processor. For the former case, MPI-like libraries provide the concept of processor groups; each thread executes on a group of processors with all collective communication operations, such as barriers, spreads, and reductions being performed only within the group. Consider again the iterated polynomial multiplication program of figure 3. Each thread shares a third of the processors. In order to properly allocate processors, the boolean **st** guards for **par** must be simple so that they may be analyzed at compile-time. Also note that each of the three threads is a data-parallel program which uses the *owner-computes* rule, whereas the top-level task-parallel program should move data to the processor-group of the thread that modifies it. Such complex information can not always be extracted at compile-time; user annotations can improve performance.

In the three combined task/data-parallel applications described earlier, processor group declarations and data-to-thread mappings were hand-inserted.

Multitasking When several threads execute on each processor, there is a need for multitasking. This is important to avoid deadlocks where all processors are blocked on a **receive** on one of their threads. Multitasking is also important for performance, as it allows for overlapping communication and computation by switching to a non-blocked thread.

With compile-time analysis in some cases it may be possible to determine the communicating pairs of threads and to schedule the operations strategically at compile-time such that no blocking

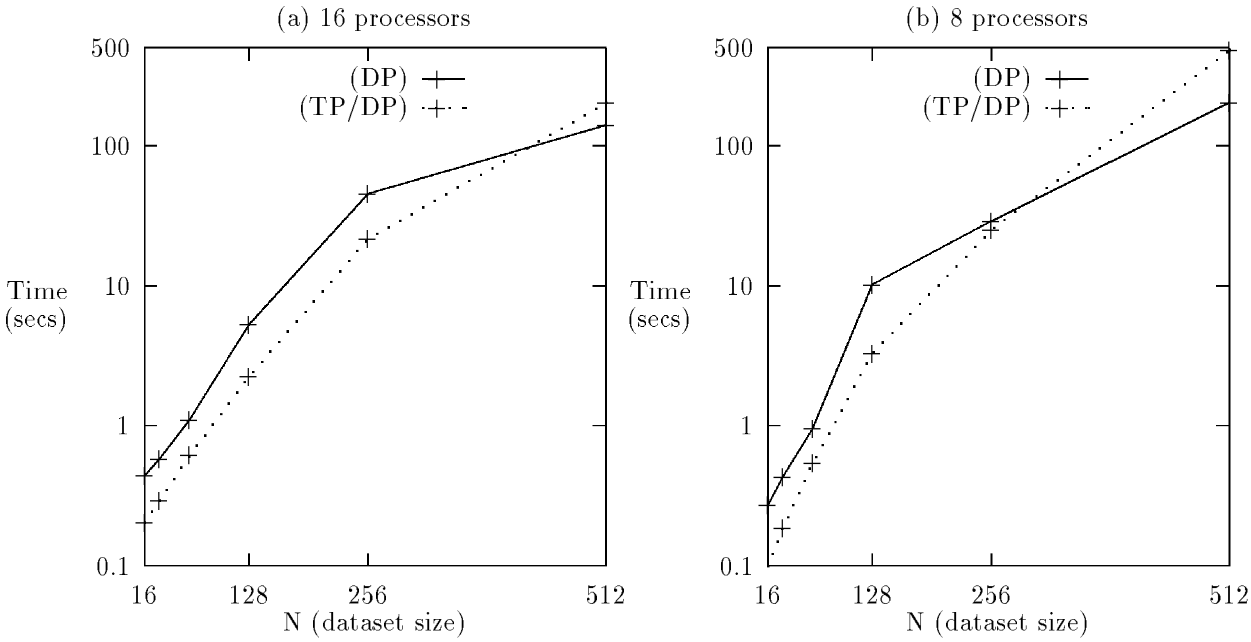

Figure 10: Performance of pattern recognition (computing moments) on IBM SP1 for two machine sizes - (a) 16 processors and (b) 8 processors.

occurs. For cases when compile-time analysis cannot be used, run-time techniques similar to the *inspector-executor* method of [24, 19] may be used, where an inspector loop at run-time determines a schedule of operations which is used by the executor loop. An alternative approach is to fork a process for each UC thread. The "weight" of the process has a crucial effect on performance. Lightweight processes such as those provided by libraries such as Chant[13] and Nexus[9] are implemented at the user-level and have less context switching overhead than heavyweight processes that are implemented at the kernel-level in the operating system.

In the combined task/data-parallel examples presented earlier, there were fewer top-level threads than processors, so no multitasking was needed.

Channel Variables A message queue associated with a channel variable is implemented by maintaining postings of non-blocking receives equal in number to queue length. A channel receive blocks until a receive completes; at this point another non-blocking receive is posted. Different message types are used to distinguish receives due to different channel variables. Mapping of channel buffers may have a significant effect on performance. In the absence of user annotations, such mappings must be deduced.

Function instances may communicate via channel variables passed as arguments. The processor on which the channel buffer lies is passed to the function as the channel argument.

For the combined task/data-parallel examples presented earlier, channel buffers were hand-placed at the destination processor; additionally, proper transformations of channel arguments were inserted by hand in the target code.

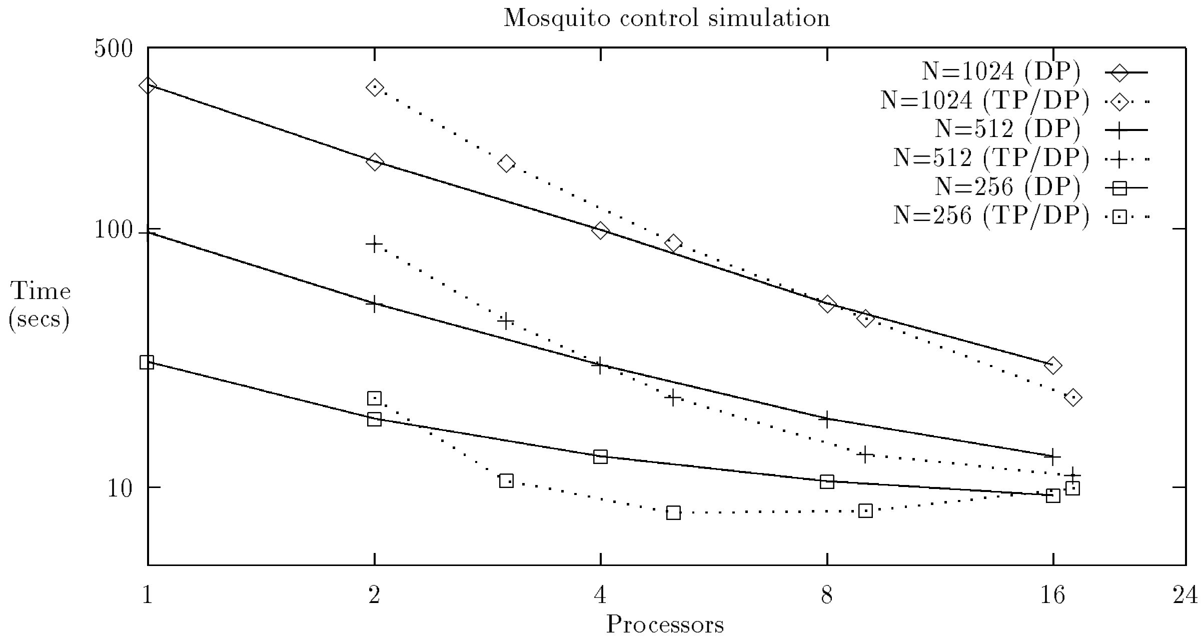

Figure 11: Performance of mosquito control simulation on IBM SP1.

6 Relationship to prior work

UC is related to collection-oriented languages like SETL and Paralation Lisp as well as data-parallel languages like C∗, DINO, FortranD, and Kali. This work is also related to some recent work in paradigm integration. We compare these to the approach presented in this paper below.

Data-parallel languages use the universally addressable memory model and typically provide optional data distribution primitives to specify how the program data are distributed over the memory hierarchy of the parallel architecture. Some languages like C∗[21] specify strict synchronization at the expression level, while other languages weaken the synchronization granularity and are synchronized at the block level. The code executed between synchronization points is not allowed to access non-local data. FortranD[15] and Kali[19] are good examples of such languages, which are sometimes referred to as SPMD languages. DINO[22] provides a more flexible SPMD model similar to UC, but unlike UC it requires local and remote data references to be distinguished statically. The primary difference between UC and existing data-parallel languages is its support for variable granularity of synchronization and its use of sets to support data and task parallelism.

SETL[25] is a high-level prototyping language that uses dynamic, heterogeneous sets as its primary data type. Control-flow in SETL is specified using the standard sequential constructs; presumably parallelizing compilers may be used to parallelize SETL programs much like they were used with Fortran loops, although the dynamic typing may make compile-time dependence analysis less effective for SETL loops. Paralation Lisp[23, 4] is a data-parallel language which is based on the notion of a group of related collections, called a *paralation*. A single communication mechanism is provided, and it is syntactically divided into two parts: setting up a communication pattern and actual data movement. In UC, the nature of the communication – whether it is a neighbor communication, broadcast, reduction, etc. – is usually visible at compile-time by inspecting array subscripts or the parallel construct being used.

Recently some researchers have explored multiparadigm integration. Most have focused on task-parallel coordination of data-parallel components. For example, Quinn et al.[26] propose a system for the Intel iWarp which is used to connect Dataparallel C computations. They extend the concept of C stream I/O to include inter-module communication channels. Their system requires each module to be written as a separate program instead of allowing computational phases to be different functions in the same program.

Subhlok et al.[27] propose a compile-time method for exploiting task and data parallelism also for the iWarp. The input program uses a FortranD-like notation with directives exposing dependencies, parallel sections, and data distributions. The compiler generates a task graph which is mapped onto the machine. However, tasks may only communicate at entry and exit points of functions. Our approach is more flexible, but also requires more programmer intervention.

Massingill[18] proposes a mechanism that uses PCN to coordinate SPMD programs written in message passing C. The SPMD programs may only communicate at entry and exit points, hence this approach is more restrictive than ours. Foster et al.[10] build on previous work in task-parallel and data-parallel Fortran compilers by using the task-parallel language Fortran M to coordinate data-parallel HPF tasks. Fortran M is designed to support dynamic task parallel programs where the identity of task is determined only at run-time and is run-time multithreaded. However data-parallel languages like HPF assume a dedicated machine of known size at compile-time. This raises the problem of integrating the run-time systems of the two compilers. Additionally, the Fortran M and HPF compiler may have different internal data representations. Data being transferred between modules may first have to be converted to representation used by the destination module.

Cheng et al.[7] use a data-flow based model through a commercially available visualization system (AVS) on the CM-5 to integrate programs written in different paradigms. This may be useful in connecting existing programs written in different paradigms, but our concern has been to provide integrated task and data parallelism within a single language.

7 Conclusion

Most existing languages adopt either the task or data-parallel paradigm. Task-parallel languages provide constructs to create and destroy asynchronous threads and to allow the asynchronous threads to communicate and synchronize as needed. Data-parallel languages use globally addressable memory together with a synchronous programming model where the computation is implicitly synchronized at some typically pre-determined level of granularity. Although data parallelism has been applied successfully to solve a large number of scientific problems, a number of applications may be more naturally and sometimes more efficiently designed using both forms of parallelism. In this paper, we have proposed the integration of task and data parallelism in an extension of C called UC. It has been shown, through performance studies on the IBM SP1, that a diverse set of applications can benefit for an integrated task/data-parallel approach.

References

[1] Austel, V., Bagrodia, R., Chandy, K.M., and Dhagat, M. Reductions + Relations = Data-Parallelism. Technical Report 930009, University of California, Los Angeles, CA 90024, April 1993.

[2] Bagrodia, R. and Mathur, S. Efficient implementation of high-level parallel programs. In *4th International Conference on Architectural Support for Programming Languages*, pages 142–153, April 1991.

[3] Bal, H.E., Steiner,J. and Tanenbaum,A.S. Programming Languages for Distributed Computing Systems. *ACM Computing Survey*, 21(3):261–322, September 1989.

[4] Blelloch, G.E. and Sabot, G.W. Compiling Collection-Oriented Languages onto Massively Parallel Computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[5] Chandy, K.M. and Misra, J. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Massachusetts, 1988.

[6] Chen, K. Efficient Parallel Algorithms for the Computation of Two-Dimensional Image Moments. *Pattern Recognition*, 23(1/2):109–119, 1990.

[7] Cheng, G., Fox, G.C. and Mills, K. Integrating Multiple Programming Paradigms on Connection Machine CM5 in a Dataflow-based Software Environment. Technical report, Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY 13244, 1994.

[8] Cormen, T.H., Leiserson, C.E., and Rivest, R.L. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1991.

[9] Foster, I., Kesselman, C., Olson, R. and Tuecke, S. Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems. Technical Report version 1.4, Argonne National Laboratory, Argonne, IL 60439, January 1993.

[10] Foster, I., Xu, M., Avalani, B. and Choudhary, A. A Compilation System That Integrates High Performance Fortran and Fortran M. In *1994 Scalable High Performance Computing Conference*, May 1994.

[11] Fry, J. and Taylor, C.E. Mosquito Control Simulation on the Connection Machine. *Proceedings of California Mosquito and Vector Control Association*, 58:207–213, 1992.

[12] Fry, J., Taylor, C.E., and Devgan, U. An Expert System for Mosquito Control in Orange County California. *Bulletin of the Soceity of Vector Ecology*, 14(2):237–246, December 1989.

[13] Haines, M., Cronk, D. and Mehrotra, P. On the Design of Chant: A Talking Threads Package. Technical Report 94-25, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23681-0001, April 1994.

[14] Hatcher, P.J. and Quinn, M.J. *Data-Parallel Programming on MIMD Computers.* Scientific and Engineering Computation Series. MIT Press, Cambridge, MA, 1991.

[15] Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., and Tseng, C.-W. An Overview of the Fortran D programming system. Report CRPC-TR91121, Center for Research on Parallel Computation, March 1991.

[16] IBM, Kingston, NY 12401. *IBM AIX Parallel Environment Parallel Programming Reference*, 1.0 edition, September 1993.

[17] Johnsson, S.L., Harris, T., and Mathur, K.K. Matrix Multiplication on the Connection Machine. Technical Report NA89-3, Thinking Machines Corporation, Cambridge, MA 02142, 1989.

[18] Massingill, B. Integrating Task and Data Parallelism. Master's thesis, Caltech, Pasadena, CA 91125, May 1993.

[19] Mehrotra, P. and Van Rosendale, J. Programming distributed memory architectures using Kali. Report 90-69, Institute for Computer Application in Science and Engineering, Hampton, VA, 1990.

[20] Prakash, S., Dhagat, M., and Bagrodia, R. Synchronization Issues in Data-Parallel Languages. In *Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, pages 76–95, Portland, Oregon, August 1993.

[21] Rose, J.R. and Steele Jr., G.L. C*: An Extended C Language for Data Parallel Programming. Technical Report PL-87.5, Thinking Machines Corporation, Cambridge, MA, March 1987.

[22] Rosing, M., Schnabel, R.B., and Weaver, R.B. The DINO parallel programming language. *Journal of Parallel and Distributed Computing*, 13:30–42, 1991.

[23] Sabot, G. *The Paralation Model: Architecture-Independent Parallel Programming.* The MIT Press, 1988.

[24] Saltz, J., Berryman, H. and Wu, J. Multiprocessors and runtime compilation. *Concurrency: Practice and Experience*, 3(6):573–592, December 1991.

[25] Schwartz, J.T., Dewar, R.B.K, Dubinsky, E., and Schonberg, E. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, New York, 1986.

[26] Seevers, B., Quinn, M. and Hatcher, P. A Parallel Programming Environment Supporting Multiple Data-Parallel Modules. In *Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multicomputers*, pages 44–47, Boulder, Colorado, September 1992.

[27] Subhlok, J., Stichnoth, J.M., O'Hallaron, D.R. and Gross, T. Exploiting Task and Data Parallelism on a Multicomputer. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–22, San Diego, California, May 1993.

A Polynomial Multiplication (FFT) UC code

```
#define PI 3.141592654
#define L 20
#define N 1024

range K:k={0..N-1};

void fft(float a_real[N], float a_imag[N], bool forward)
{
  int s, m, j, c = 1, ub = log(N)/log(2);
  float t_real[N], t_imag[N], u_real[N], u_imag[N];
  float tmp_real[N], tmp_imag[N], w_real_fact[N], w_imag_fact[N];
  float w_real, w_imag, wm_real, wm_imag, tmp;
  int dst[N], my_k[N], bitmap[N];
  channel x[N];

  /****** initialize *******/
  if (forward) read_input(a);

  /****** put a in bit-reverse order ******/
  arb (K) {
    my_k[k] = k; dst[k] = 0;
  }
  for (s = ub-1; s >= 0; s--) {
    arb (K) {
      bitmap[k] = 1;
      if (my_k[k] >= pow(2, s)) {
        bitmap[k] = bitmap[k]<<ub-(s+1);
        dst[k] = dst[k]|bitmap[k];
        my_k[k] -= pow(2, s);
      }
    }
  }
  arb (K) {
    x[dst[k]] = b_real[k]; b_real[k] = x[k];
    x[dst[k]] = b_imag[k]; b_imag[k] = x[k];
  }

  if (!forward) c = -1;

  for (s = 1; s <= ub; s++) {

    m = pow(2, s);
    wm_real = cos(c*2*PI/m); wm_imag = sin(c*2*PI/m);
    w_real = 1.0;           w_imag = 0.0;

    for (j = 0; j <= m/2-1; j++) {
      arb (K) st ((k-j)%m == 0) {
        w_real_fact[k] = w_real; w_imag_fact[k] = w_imag;
      }
      tmp    = w_real*wm_real - w_imag*wm_imag;
      w_imag = w_real*wm_imag + wm_real*w_imag;
      w_real = tmp;
    }

    arb (K) {
```

```
      /*******  t <- w * a[k + m/2] *******/
      x[(k-m/2+N)%N] = a_real[k]; t_real[k] = x[k];
      x[(k-m/2+N)%N] = a_imag[k]; t_imag[k] = x[k];
      tmp_real[k] = w_real_fact[k]*t_real[k] - w_imag_fact[k]*t_imag[k];
      t_imag[k] = w_real_fact[k]*t_imag[k] + t_real[k]*w_imag_fact[k];
      t_real[k] = tmp_real[k];

      /*******  u <- a[k] *******/
      u_real[k] = a_real[k]; u_imag[k] = a_imag[k];

      /*******  a[k] <- a[k] + t *******/
      if ((0 <= k%m) &&(k%m < m/2)) {
        a_real[k] += t_real[k];a_imag[k] += t_imag[k];
      }

      /*******  a[k + m/2] <- u - t *******/
      tmp_real[k] = u_real[k] - t_real[k];
      tmp_imag[k] = u_imag[k] - t_imag[k];
      dst[k] = (k+m/2)%N;
      x[(k+m/2)%N] = tmp_real[k]; tmp_real[k] = x[k];
      x[(k+m/2)%N] = tmp_imag[k]; tmp_imag[k] = x[k];
      if ((m/2 <= k%m) &&(k%m < m)) {
        a_real[k] = tmp_real[k]; a_imag[k] = tmp_imag[k];
      }
    }
  }

  if (!forward)
    arb (K) {
      /*******  a[k] <- a[k]/N *******/
      a_real[k] /= N; a_imag[k] /= N;
    }
}


int main(void)
{
  range P:p={0..2};
  int l;
  channel x_real[N], x_imag[N];
  channel y_real[N], y_imag[N];

#if DATA_PARALLEL
  {
    float a_real[N], a_imag[N];
    float b_real[N], b_imag[N];
    float c_real[N], c_imag[N];
    for (l = 0; l < L; l++) {
      fft(a_real, a_imag, true);
      fft(b_real, b_imag, true);
      /* pointwise multiply */
      c_real[k] = a_real[k]*b_real[k] - a_imag[k]*b_imag[k];
      c_imag[k] = a_real[k]*b_imag[k] + b_real[k]*a_imag[k];;
      fft(c_real, c_real, false);
    }
  }
#endif /* DATA_PARALLEL */
```

```
#if TASK_N_DATA_PARALLEL
  arb (P)
    st (p == 0) {
      float a_real[N], a_imag[N];
      for (l = 0; l < L; l++) {
        fft(a_real, a_imag, true);
        arb(K) {
          send(x_real[k], a_real[k]); send(x_imag[k], a_imag[k]);
        }
      }
    }
    st (p == 1) {
      float b_real[N], b_imag[N];
      for (l = 0; l < L; l++) {
        fft(b_real, b_imag, true);
        arb(K) {
          send(y_real[k], b_real[k]); send(y_imag[k], b_imag[k]);
        }
      }
    }
    st (p == 2) {
      float a_real[N], a_imag[N];
      float b_real[N], b_imag[N];
      float c_real[N], c_imag[N];
      for (l = 0; l < L; l++) {
        arb(K) {
          a_real[k] = receive(x_real[k]);
          a_imag[k] = receive(x_imag[k]);
          b_real[k] = receive(y_real[k]);
          b_imag[k] = receive(y_imag[k]);
          /* pointwise multiply */
          c_real[k] = a_real[k]*b_real[k] - a_imag[k]*b_imag[k];
          c_imag[k] = a_real[k]*b_imag[k] + b_real[k]*a_imag[k];
        }
        fft(c_real, c_real, false);
      }
    }
#endif /* TASK_N_DATA_PARALLEL */

}
```