The Parallel C Preprocessor*

EUGENE D. BROOKS III, BRENT C. GORDA, AND KAREN H. WARREN

Massively Parallel Computing Initiative, Lawrence Livermore National Laboratory, Livermore, CA 94550

ABSTRACT

We describe a parallel extension of the C programming language designed for multiprocessors that provide a facility for sharing memory between processors. The programming model was initially developed on conventional shared memory machines with small processor counts such as the Sequent Balance and Alliant FX/8, but has more recently been used on a scalable massively parallel machine, the BBN TC2000. The programming model is *split-join* rather than *fork-join*. Concurrency is exploited to use a fixed number of processors more efficiently rather than to exploit more processors as in the fork-join model. Team splitting, a mechanism to split the team of processors executing a code into subteams to handle parallel subtasks, is used to provide an efficient mechanism to exploit nested concurrency. We have found the split-join programming model to have an inherent implementation advantage, compared to the fork-join model, when the number of processors in a machine becomes large.

1 INTRODUCTION

Shared memory multiprocessors, wherein a small number of processors access a common monolithic system memory, have been around for a long time. The frequently used parallel programming model on these systems is the fork-join model, where one processor starts out executing the serial code and additional processors are acquired when a parallel construct is encountered. Although the fork-join model is serving well in the form of vendor supplied implementations on machines with a small number of processors such as the Cray YMP and the Convex C-2 series, fundamental shortcomings in its implementation begin to surface when the number of processors grows. These shortcomings have become apparent through our experiences with the vendor supplied fork-join extension of Fortran [1] supplied by BBN on their TC2000 multiprocessor.

It is no longer possible to build a monolithic shared memory system that can sustain the bandwidth and latency demands of hundreds or thousands of processors when the number of processors is scaled up. Large local memories are introduced in order to provide an acceptable level of memory performance in such systems. Use of the shared memory facility in large parallel systems is best relegated to those times when communication is absolutely required *by the application*, as the bandwidth and latency of the shared memory are likely to be an order of magnitude short of the performance required to keep a processor running at full speed. The problem with the fork-join model is that the process of acquiring and relinquishing processors requires highly contended accesses for the shared memory, or some specialized hardware mechanism to handle processor dispatch. Although the fork-join program-

* Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48.

Received February 1992.

Revised April 1992.

Scientific Programming, Vol. 1, pp. 79–89 (1992)

CCC 1058-9244/92/010079-11$04.00

ming style is comfortable, the high overhead of its implementation quickly becomes burdensome.

An alternative to the fork-join model is the split-join programming model. In the split-join paradigm, all the processors that the code will ever have will enter the main program in a *live* manner at the start of the job. Processors are not fetched from and returned to a pool. This involves accesses to shared memory to implement. PCP [2] is an implementation of the split-join programming model that provides a good fit to massively parallel machines offering a shared memory facility in addition to the local memory that such machines usually have. Its constructs, though simple, provide powerful and efficient flow control in the form of parallel loops and team splitting. The programming model is a relatively straightforward extension of the conventional C programming model. An arbitrary number of processors executes the code stream as a single processor executes a serial program.

Nested concurrency is exploited through the structured mechanism of team splitting, in which the team of processors divides up into subteams in order to address independent but parallel tasks at a lower level. These subteams are relatively independent of one another, and are free to execute different code modules, until they rejoin the parent team in a block structured manner.

The PCP programming language places the issues of scheduling, communication, and synchronization directly into the hands of the programmer. It does this in a simple manner by providing useful constructs to handle these complexities. PCP allows the user to specify which portions of the program are to be executed in parallel, which are to be executed by subteams, and which are to be executed by one processor only. PCP allows any number of processors to be allocated to a job, the number of processors being a run time parameter that is set by the end user of the code when the program is actually executed. Through the use of a compile time flag, the PCP preprocessor can also produce serial code that does not contain the run time synchronization overhead required for parallel execution. This feature is often used to provide both a sanity check and a point of performance comparison for parallel executions.

The split-join parallel programming paradigm is independent of the target language. Examining the basic programming model, it is quickly concluded that any procedural programming language could be extended with a split-join programming model. This is indeed the case and on the BBN TC2000 we have developed a Fortran implementation, the Parallel Fortran Preprocessor (PFP). We discuss only the C syntax for the split-join programming model for the sake of brevity (see Warren et al. [3] if you are interested in the Fortran syntax).

PCP was originally written to solve the problem of the portability of C language based parallel programs among several different shared memory multiprocessors. The PCP preprocessor is machine independent; the amount of machine dependent run time support is small (200 lines of C on the BBN TC2000) and can be easily implemented with fast inline code. PCP has been used successfully on the Alliant, Sequent, Cray, SGI, and Stellar machines.

The sections of this paper are as follows. The split-join model and its memory model are described in Sections 2 and 3. Details on how the implementation of these models take advantage of the architecture are included. The PCP team state is discussed in Section 4. The synchronization primitives offered in PCP are discussed in Section 5. The actual PCP syntax is shown in Section 6. In Section 7 we discuss the issues of implementing and using PCP on a scalable machine such as the BBN TC2000. A general discussion follows in Section 8.

2 THE SPLIT-JOIN MODEL

In the traditional fork-join parallel programming model, a single processor starts the execution of the program and acquires more processors as concurrency is encountered in the code. The fork-join programming model has been quite useful on tightly coupled shared memory machines with relatively few processors. Some architectures such as the Alliant FX/8 and the Convex C2 provide special hardware to make the dispatch of slave processors happen as quickly as possible. Scalable machine architectures are not as tightly coupled and the cost of communication between processors, heavily used in process of dispatching processors in the fork-join model, is relatively high. The BBN TC2000, described in more detail in Section 7, is a realistic example of what might be expected in this regard. The latency of a cache hit on local memory is 3 clocks (pipelined at a rate of one per clock) whereas the latency of a remote memory reference is roughly 40 clocks (not pipelined). If one must deal with a 40 clock latency for every memory reference required in the code used

to dispatch processors, even an efficient spanning tree implementation can have substantial overhead.

In the split-join paradigm we deal with the high cost of processor dispatch and communication between processors by minimizing their occurrence in the fundamental constructs of the programming model. All of the processors the job will ever acquire are dispatched at the start of the program and are immediately placed under the control of the programmer. This group of processors, which loosely follow each other through the code, is referred to as a *team* of processors. In a serial program a single processor enters the **main**( ) routine and executes code until it either returns from **main**( ), calls **exit**( ), or encounters an exception. In the PCP model we generalize this in a natural way, having a team of processors enter **main**( ) and having the job end when any team member returns from **main**( ), calls **exit**( ), or encounters an exception. A team of processors consists of a team *master* and zero or more other processors, which travel through the code almost in unison.

The split-join parallel programming model is very similar to Harry Jordan's Force [4] and the IBM SPMD [5] programming model, the most significant difference being the support for team splitting and the arbitrary nesting of concurrency constructs. The PCP concept of team splitting allows an arbitrary subdivision of the team of processors executing the code and allows each subteam to execute arbitrarily different codes within the constraint of block structure. This is a flexible extension of the SPMD programming model that supports the exploitation of nested concurrency for both subroutines and nested loops.

The PCP concept of teams is dynamic and well suited for massively parallel machines. The user synchronizes the members of a team, designates tasks for individual team members, and splits up the team into smaller sized teams to execute logically distinct code sections. All parallel constructs apply to teams. Working within any team the programmer has access to the parallel looping constructs, the barrier, locks, and even further team splitting. In the team splitting process the total number of processors remains constant. The processors temporarily become members of new subteams. As this happens, the processors save their old state for restoration at the end of the split construct. As processors within a team complete their work, they then rejoin the parent team with no implicit serialization.

Team splitting is used to exploit nested concurrency with a fixed number of processors. In the split construct the user explicitly marks off separate blocks of work that can be executed independently (each block of work may itself be a job consisting of subtasks that can be executed in parallel). The user may also indicate the relative amount of work in each block of code. The PCP split construct takes an optional weight parameter that controls the fraction of processors to send into each team. This form of load balance control is effective if the workload can be accurately predicted at execution time, as is the case for the two concurrent linear system solves below. The team splitting weights, which occur in an iterative loop during execution, might also be corrected by using a real time clock to detect imbalance in a prior iteration and adjusting the weights of the next iteration appropriately. This second strategy would be effective if the relative load of two split tasks varied slowly with the iteration index.

Splitting the team into smaller subteams to exploit nested concurrency is counter intuitive. The goal, however, in executing nested concurrency is to use a *fixed* number of processors more efficiently, not to use *more* processors. The split-join programming model is in some sense the *dual* of the fork-join model. One finds that one can usually accomplish the task at hand with either programming model. The advantage of the split-join model is its bottleneck free implementation through a highly portable preprocessor. As will be seen in Section 4, the process of team splitting is accomplished in a few instructions, independent of the size of the team, which only access local memory.

Teams may be split both statically and dynamically. Static team splitting is specified using the keywords **split** and **and**. As an example, consider the routines **foo**( ) and **bar**( ), which perform an equal amount of work and can be called concurrently. In the PCP model the concurrent execution of the two routines is arranged through the syntax:

```
split
{
  foo( );
}
and
{
  bar( );
}
```

where the amount of work performed by **foo**( )

and **bar**( ) is assumed to take the same amount of time. The team encountering the split divides into two subteams of equal size. The first subteam calls **foo**( ) while the second calls **bar**( ) concurrently, and the two subteams join again at the end of the **and** block. The tasks performed in **foo**( ) and **bar**( ) must be independent. If there is only one processor in the team that encounters the split, or if an implementation limit for team splitting has been reached, the encountering team calls **foo**( ) and then **bar**( ) sequentially in an implementation dependent order. For this reason, one must view the execution of **foo**( ) and **bar**( ) to be completely asynchronous. The algorithms implemented by **foo**( ) and **bar**( ) should not be designed in a way that requires simultaneous progress in both routines. This could be done, but the code would not be portable to multiprocessors for which the team size entering the split block is unity.

In addition to the static splitting described above wherein the amount of work performed in each block of code is assumed to be the same, weights may be assigned to the blocks of work. If the user has provided accurate loading information, via the *weight* parameters for team splitting (see Section 6.5) that determine the subteam sizes, the processors in the subteams finish their work and rejoin to become the parent team nearly simultaneously. The total number of processors is conserved in the team splitting process. As an example of team splitting with weights, consider the case of two concurrent linear system solves that are for potentially differing dimensions. (For a description of the Gauss elimination implementation in this model see Warren et al. [6])

```
int dim1, dim2;
double **matrix1, **matrix2;
double *rhs1, *rhs2;
split dim1*dim1*dim1 {
      dgauss(matrix1, rhs1, dim1);
}
and  dim2*dim2*dim2 {
      dgauss(matrix2, rhs2, dim2);
}
```

Here the routine **dgauss**( ) performs a linear system solve leaving the result in the vector rhs provided as an argument. The operations required scale like the cube of the dimension and this is noted by the weight expressions included on the **split** and **and** lines. The weight expressions are used to compute the sizes of the two subteams, dividing the parent team into two subteams having relative sizes that match the ratio of weights as closely as possible. By specifying weights for the concurrent blocks of work some measure of load balancing can be achieved, subject to the algebraic restrictions caused by the fact that the number of processors is finite.

Split constructs are not limited to the static binary form shown above. Static splits may have more parallel blocks specified by concatenating and blocks.

Team splitting may also be treated dynamically. The construct:

```
splitall(int i = 0; i < imax; i += 1)
{
     <work dependent on the index, i>;
}
```

specifies that the body of the loop is to be executed with the indices **i = 0, 1, . . . , imax-1**, parceling out the indices to a collection of subteams that are split off from the team that encounters the split. The actual number of subteams is determined at run time, and is possibly influenced by a compile time flag. There may not be a split at all or the team may be split to individual processors, using the best heuristic algorithm that can be conjured up. Extra parameters (see Section 6.6) inside the **splitall** header can be used to establish firmer control of the team splitting mechanism.

To give a trivial application of the **splitall** loop, consider the parallel computation of a set of matrix vector products.

```
double **result;
double ***matrices;
double **multplcnd;
int dim;
int number;
splitall(int i = 0; i < number; i += 1) {
  mvprod(result[i], matrices[i],
                    multplcnd[i], dim);
}
```

If a team split would be profitable, the team encountering the **splitall** block is divided into subteams, each subteam handling a subset of the indices **i**. The library routine **mvprod**( ) is designed for team entry and contains parallel language constructs designed to efficiently exploit the parallelism of each matrix vector product. If the team that enters the **splitall** loop has 100 processors and the number and dimension of the matrix-

vector products is 5 and 20, respectively, we see that the use of team splitting will have a substantial impact on program performance.

3 THE MEMORY MODEL

PCP allows the user to designate the memory class for all data. In the split-join programming paradigm itself, three types of memory are provided to fully exploit the notion of team splitting. These are:

1. Memory that is private to a processor, *private* memory
2. Memory that is shared among all processors, *shared* memory
3. Memory that is shared among the members of a given team or grouping of processors, but private to the team, *teamprivate* memory

Private memory is implemented on the processor that requires access to it. Shared memory is implemented in the interleaved shared memory facility (see Section 7) of the BBN TC2000. There are many situations where a user would like to have static data that is not shared by all processors, but is shared by all of the processors within a given team. This type of data is declared using the storage class modifier *teamprivate*. Teamprivate memory is allocated as an array in the interleaved shared memory, indexed by a *team descriptor* that is unique to a given team. More details on the team descriptor are given in Section 4.

By default, all statically allocated data is shared and thus accessible by all of the processors. Stack, or **auto**, data is private to a processor and is stored in a processor's local memory if it exists. The choice of default for statically allocated data is a holdover from the days of running PCP codes on shared memory machines such as those manufactured by Sequent or Alliant, and is not that appropriate for a scalable machine such as the BBN TC2000. Fortunately, the default can be switched to private through the use of a compile time flag.

4 TEAM STATE

The PCP concept of teams is implemented using a small amount of local memory. The *team state*, which is carried by the processors, is made available to the programmer who may use it to construct parallel language extensions that are not directly supported. The current features of PCP were not postulated before its first implementation. These features evolved over time as users constructed some of their own using the team state variables. Those features that were found to be generally useful have been standardized and elevated to constructs directly supported by the Parallel C Preprocessor itself. Further evolution of the parallel programming model in this manner will reduce the need to directly use the team state, but the team state will always remain accessible to the programmer both for backward compatibility and to facilitate the creation of new language constructs that might eventually find their way into PCP.

The team state consists of five values that are carried along by the processors. Two of these values are implicitly *read only* (not to be touched by user programs). They are:

1. **_NPROCS**, that is, the number of processors that execute the program. It is the size of the team that enters main.
2. **_IPROC**, that is, the processor index. It has a value unique to each executing processor in the range from 0 to **_NPROCS-1**.

These two values are set by the run time system before **main**( ) is entered and under no conditions should they be changed. In implementations where the target multiprocessor does not support local memory directly, the processor index is used to index an array to simulate local memory. To support the concept of team splitting three more variables were added to the set carried along by the processors, and the name *team state* was coined. Unlike the processor index and the number of processors, these values are *read/write* and are manipulated by the language constructs of PCP:

1. **_TSIZE**, that is, the team size. If no team splitting has occurred, this will be equal to _NPROCS.
2. **_TINDEX**, that is, the index of the member within the team. It must have a unique value within the team in the range, 0 to **_TSIZE-1**. The team master is the processor with a team index of zero.
3. **_TDESC**, that is, the team descriptor, a non-negative value unique to the team.

Teams are a dynamic association of physical processors and because of this teams can be created and destroyed as these associations change during split-join operations. The team descriptor is a small positive integer used to identify the team and provide for teamprivate memory, which is private to the team but shared among the team members. Arrays in shared memory are indexed by this value to simulate the notion of privacy. Without the team descriptor distinct teams could not have independent barrier operations.

PCP quickly calculates the new team descriptor for team splits in local memory by shifting the current team descriptor left n times where n is the log (base 2) of the number of new subteams and 'oring' in an integer from 0 to (the number of subteams -1). Teamprivate memory is not initialized. As team descriptors are reused, initializations of teamprivate data are almost guaranteed to have been corrupted. The team descriptor and team size are identical for all of the members of a given team but each team member carries its own private copy in order to prevent hot spots [7] as the team state is accessed.*

Team splitting is handled in a block structured way. Each time a processor becomes a member of a new subteam, it computes a new team descriptor and its position in the new team without accessing any shared memory or synchronization resources. This leads to an efficient bottleneck free implementation of team splitting, the cost of which is completely independent of the number of processors in the team. As the processor computes a new team descriptor, it pushes the old one onto a private stack for recovery when it reaches the end of its share of the work in the split block.

Since a processor carries the team descriptors of all its antecedent teams on a stack, it has access to the teamprivate memory of a parent team. This can be very useful in a situation where the tasks in the split blocks are to compute some results required by all the members of the parent team, but for which the use of the top level shared memory would pose an access hazard due to nested use of team splitting in a reentrant way. We have not given the syntax for this here, as the notion of accessing the teamprivate data of a parent team is still undergoing exploratory use. We expect changes in syntax and functionality as we learn from the experience of users.

The sophisticated user will find the team index and team size useful when a custom scheduling algorithm for a segment of code is desired. By customizing the scheduling algorithms for certain tasks the need for barrier synchronization can often be reduced, with an attendant increase in efficiency. Programmers must be careful to design the custom algorithm so that it will work properly for any team size. The more aggressive programmer could also manipulate the team state values, perhaps creating custom splitting algorithms.† Simple heuristic techniques are used for team splitting and sometimes a customized heuristic is warranted for special circumstances. Programmers must be careful to preserve the integrity of the team state. The correct functioning of nearly all the parallel language constructs is dependent on the team state being consistent on each processor.

5 SYNCHRONIZATION

Barrier synchronization and the notion of locks are provided in the PCP implementations of the split-join programming model.

In barrier synchronization, all of the processors in a given team are forced to wait at the barrier until the last processor arrives. A bottleneck free software implementation [8] is used, requiring 30 to 40 microseconds to synchronize 32 processors. The execution time of the barrier scales as the log of the processor count. Each team has its own unique barrier.

A lock is used to provide for critical region access to data. A processor attempting to acquire a lock spin-waits until the lock is unlocked and then indivisibly locks it. When the processor unlocks the lock it is available for others immediately. Locks may be declared by the user in either shared memory or teamprivate memory. When in shared memory, the lock is visible to all the processors, regardless of the team to which they belong. Locks declared in teamprivate memory are visible only within a team.

In addition to the use of barriers and locks, the user may implement event notifications by simply spin-waiting on a location in shared or teamprivate memory to change. On a machine supporting coherent shared memory caches this is particularly effective and has no negative impact.

* Hot spots are shared memory locations for which many processors are contending.

† It was, in fact, through such activity that the notion of team splitting was invented initially.

If the machine lacks this support, as is the case for the BBN TC2000, users must be careful about the possibility of generating adverse impact on available memory bandwidth through the introduction of a hot spot.

6 PCP SYNTAX

A well structured C code requires very few changes to make effective use of parallelism. While this typically does not take into account data locality issues, simply getting a program running correctly with the split-join programming model is usually not a difficult task. This has been our experience with PCP on systems providing a monolithic shared memory. On systems with a hierarchical memory structure, such as the BBN TC2000, further work optimizing the code to exploit data locality is required to get the expected performance out of the hardware. In this section, we describe the lightweight parallel excecution mechanisms of PCP.

Under the assumption that nonlocal memory references are expensive relative to local references, PCP control constructs generate fast inline code involving only local references. The typical control construct requires only a few local memory references for execution. The absence of overhead in this area has allowed programmers to exploit parallelism in sections of their applications that were previously deemed not heavy enough to amortize the overhead. In addition, the control structures contain no implicit barrier synchronization. If there is no data dependency explicitly involved with the construct, processors are free to run ahead in the execution of subsequent code.

A short summary of the PCP syntax follows. For more detailed specification, see the PCP user's manual [9]. Anyone interested in the equivalent PFP syntax for Fortran should refer to the PFP user's manual [3].

6.1 master

Within the context of a specific team, that processor whose current team index is 0 executes the code delimited by a **master** block. Arbitrary PCP constructs may be enclosed by a **master** block. A **master** block is often used in the portion of the program that performs initialization as well as input, output, and memory allocation. At a much smaller scale of granularity, **master** blocks are used to initialize shared data such as accumulators, which all team members will access.

```
master {
    <declarations>
    <executable code>
}
```

Note that a **master** block does not provide a serial critical region in the sense that most people think. If team splitting has occurred, several teams exist each with its own master and each executing its own task. Multiple teams may be in **master** blocks concurrently. A race condition could exist within a **master** block for access to shared data if it is possible for the teams to encounter the block asynchronously. (A simple global semaphore is all that would be required to protect the region). Thus **master** blocks do not necessarily have the Amdahl's law* impact that they might be otherwise expected to have.

6.2 forall

The **forall** loop is the PCP concurrent equivalent of the C language **for** loop. It achieves a fine-grained parallelism by dividing the passes of the **for** loop among the members of the team:

```
forall (int i = <start>; <cond>;
                        i += <step>)  {
    <work dependent on the index, i>;

}
```

The indices of the loop are interleaved among the members of the executing team. The loop index variable must be declared in the **forall** statement. We have borrowed this syntax from C++† to remind the user that the loop index is not defined after the closing brace of the loop body. The **<start>** and **<step>** expressions are currently restricted to simple constants or variables. The **<cond>** expression is unrestricted and not checked for sanity. **forall** loops may be nested arbitrarily. The team index inside the loop body is set to 0 and the team size to 1. This makes a team of size 1 for the scope of the loop iteration, ena-

* Amdahl's law states that a program's performance is dominated by its slowest component, typically a serial section.

† In C++, the index **i** would be defined outside of the loop body. We borrowed the syntax, but not the semantics.

bling any enclosed PCP constructs to work correctly.

6.3 barrier

The team of processors executing the code freely run through it unless explicit synchronization primitives are encountered. One basic and frequently used form of synchronization is the *barrier*: **barrier;**.

A barrier requires all members of the team to arrive at the barrier before any are allowed to continue. Each team has its own distinct barrier. A barrier is often used after a **master** block or a **forall** loop to ensure that the preceding work is complete before any processor is allowed to continue. A fast algorithm[8] that has no hot spots or critical regions has been implemented for PCP run time support.

6.4 lock, unlock

Concurrency must be inhibited in a statement that modifies a variable that many processors are accessing for both reading and writing. To prevent processors from destructively interfering with each other, entrance to a critical section of a code must be restricted so that only one processor may execute it at a time. This is accomplished by using a lock.

PCP offers spin-wait locks that are implemented by variables of the **lock** data type, which has the two states **locked** and **unlocked.** A **lock** variable is a statically allocated and initialized PCP data type:

```
lock var = unlocked;
```

The functions that change the state of a lock are **lock**( ) and **unlock**( ), which take the pointer to the lock variable as an argument. **lock**( ), when passed a pointer to a variable of the data type, **lock,** waits until the **lock** is **unlocked** and then atomically sets it to **locked**. **unlock**, when passed a pointer to a lock, sets it to **unlocked**. A lock is used to protect a critical section in the following way:

```
lock(&var);
  <critical section>
unlock(&var);
```

The lock variable may be declared as either shared or teamprivate. If the lock variable is shared, then the critical section is global. If the lock is declared teamprivate, then the critical section is local to the team.

6.5 split

To divide a number of tasks, which is known at compile time, among subteams that are split from the parent, static team splitting is used:

```
split <weight1> {
    <task1>
}
and <weight2> {
    <task2>
}
. . .
and <weightn> {
    <taskn>
}
```

The tasks may be executed in any order, including sequentially if the team encountering the split statement cannot be split for some reason. If one task contains more work than another, weights may be assigned to the blocks of work to achieve some measure of load balancing. The weights determine the fraction of the current team's processors that are split into each subteam.

6.6 splitall

The dynamic version of team splitting is the **splitall** loop:

```
splitall (int i = <start>; <cond>;
      i += <step>[;nteams[;tsize]]) {
   <work dependent on the index, i>;
}
```

When a team encounters a **splitall** loop, it dissociates into subteams to which the indices of the loop are interleaved. The number and size of the subteams may be determined by the optional integer expressions, *nteams* (for specifying desired number of teams) and *tsize* (for desired size of teams), or by compile time flags to PCP. If the appropriate number of processors are available at run time the user supplied directives are followed. Otherwise the number of teams and team size are picked by the implementation giving priority to the *nteams* parameter.

7 PCP AND THE BBN TC2000

On conventional shared memory multiprocessors, which provide a single monolithic shared memory system to the programmer, a great deal of effort does not have to be expended to restructure a serial code to exploit data locality in order to closely approach the performance limits of the machine. This situation is characteristic of the shared memory machines manufactured by Cray Research, Alliant, Convex, and Sequent. There is perhaps the slight exception of those systems that employ coherent caches in an attempt to insulate an inadequate shared memory system from the demands of the processors.

Massively parallel systems, on the other hand, can have a rather complex structure, which includes local memory, finely interleaved shared memory scattered across the processor boards, and shared memory, which all processors can access but which remains on a given processor board as the physical address is incremented. We call this last form of shared memory *block shared memory*, for lack of a better term. Block shared memory has the characteristic that the processsor on the same card as the memory accesses it without going through the interconnection network. The other processors access this memory through the interconnection network and face latency and contention problems in order to gain access. Machines with this complex memory structure have the advantage of being scalable to very high processor counts, but the programmer pays for this advantage with a much more demanding coding effort being required to obtain a reasonable fraction of the performance capability offered by the architecture.

On the BBN TC2000, accesses through the network have priority over local memory references. We have exploited this feature in the supplied PCP synchronization constructs by arranging that a processor that is busy polling does so in a memory location residing in the on–card block shared memory. This has been done in the barrier synchronization algorithm, the most heavily used synchronization operation. Doing this is a simple matter of allocating the barrier data structure used to control the progress of the processors in appropriate segments of block shared memory. For locks, the data structure representing the state of the lock resides in interleaved shared memory, but a linked list of pointers to locations in block shared memory can be used to control processors that are waiting for the lock. This is referred to as a queueing lock [10]. Using a lock data structure of this sort, users again arrange that busy polling is done in the on-card block shared memory at low priority on the BBN TC2000 so that the progress of processors accessing this memory through the switch in the course of performing real work is not retarded. We have not implemented queueing locks, as application codes have typically used locks in a way that avoids contention.

As if having to deal with local, interleaved shared, and block shared memory is not complicated enough, the performance of the network used to support memory references between processors can be highly dependent on how the run time communication patterns clash with the topology of the network. If the interconnection network is a 2-D mesh, for instance, a remote memory reference to a neighboring processor is accomplished with a much lower latency and higher bandwidth than a remote memory reference to a random processor in the machine. Fortunately, we have not had to face this problem on the BBN TC2000 as the latency and bandwidth of the interconnection network used in this architecture are rather independent of the source and destination addresses for memory traffic flowing through the network.

The BBN TC2000 [11] is a scalable multiprocessor architecture that can support up to 512 Motorola 88100 RISC microprocessors running at 20 MHz. The processors, with their 16 megabyte memories, are interconnected to each other in a PE-to-PE model using a variant of a multistage cube network [12], which BBN refers to as the "butterfly switch." The BBN TC2000 supports local memory, block shared memory wherein successive cache lines reside on one processor card, and interleaved shared memory wherein successive cache lines are placed on successive cards and wrap around the machine. The aggregate bandwidth to interleaved shared memory scales linearly with the number of processor cards contributing to this memory pool.

The contribution of each node to the interleaved shared memory pool is made at boot time, set via device registers in the interface to the switch that connects the processors. Any number of processors can be configured to contribute to the interleaved shared memory pool and it is useful and convenient to set the number of contributing processors to a prime number to avoid bandwidth reductions when accessing arrays in a strided manner. The rest of the memory in each node can be used for either local memory or block

shared memory. The division between these two memory types is enforced by the memory management unit attached to the processor and is set at the time an application is run using operating system calls in a completely flexible way.

On the BBN TC2000, the memory latency for a cache hit is 3 clocks, the latency for a cache miss to the local memory card is 8 clocks, and the latency for a remote memory reference satisfied through the switch is 39 clocks. Cache hits are fully pipelined at a rate of one access per clock. Cache misses, or noncached accesses, cause the processor to stall until the request is satisfied. Because of the rather large performance penalty for accesses to interleaved shared memory compared to cache hits users are required to expend a considerable amount of effort in localizing data structures if they are to achieve a substantial fraction of the performance potential of the machine. Using the current PCP facilities, users have typically managed the interleaved and local memories [13] to accomplish this. The use of block shared memory has usually been implicit, *but critical*, having been hidden in the provided synchronization constructs. Some users have gone further by using block shared memory directly for their own data structures,[14] but have found doing this to be tedious.

8 DISCUSSION

PCP was originally developed on shared memory systems with small processor counts to solve the code portability problem for parallel applications written in C for this class of machines. During this period, which in fact has not yet ended, each vendor of a parallel machine developed and marketed its own parallel execution environment, which was incompatible with those of other vendors. The implementation of PCP as a preprocessor, with a minimal amount of machine dependent run time support, was critical to being successful in achieving this goal. The split-join paradigm, an extension of the simple SPMD model [5], allows nested concurrency to be exploited while still preserving the simple preprocessor implementation and high efficiency.

In the early days of using PCP on a 12-processor Sequent Balance or on an 8-processor Alliant FX/8, users did not have to concern themselves with exploiting data locality. Team splitting was used to improve the concurrency available and was used to reduce overhead by reducing the cost of a barrier within a team. PCP was used to obtain rather efficient execution on shared memory multiprocessors [15], but users had to be very careful about the quantization error involved when splitting a team of a few processors in size and this tended to restrict the use of team splitting to the trivial case of binary splits of an even number of processors for two equally sized subtasks.

With the BBN TC2000, we have up to 128 processors available* and the quantization error for team splitting is an order of magnitude smaller. This allows much more accurate matching of the team size to the work involved for the subtasks. The sheer number of processors available begins to support deeply nested team splitting. With the earlier shared memory systems possessing only a few processors, a programmer could essentially enumerate the task for each processor individually and keep track of them. With hundreds or more processors, the programmer ceases to think of individual processors and begins to think of teams as abstract aggregates of processors.

PCP provides no frills for the programmer. No attempt is made to automatically parallelize serial code, and in particular, data dependence analysis is entirely up to the user. If users write code that allows a race condition, they are likely to get bitten by it sooner or later. Adding data dependence analysis to aid the user in spotting the rather trivial race conditions that often bite would be useful, and is most certainly technically possible, but would require a compiler development that is entirely beyond the scope of the effort to date.

PCP, in its current form, only offers easy management of data structures in private and interleaved shared memory. Users must allocate and manage data structures in block shared memory by hand, and as reported, this is profitable but very tedious [14]. Data distribution models supporting data parallel computation, which have recently been a very popular research area [16], provide a means of exploiting data structures allocated in block shared memory. We are developing such a programming model as a high level layer on top of our Fortran implementation of the split-join programming model, PFP. This will hopefully provide the ease of programming that Fortran 90 offers for array operations, while maintaining an efficient escape to the PFP split-join programming model for everything else. The

* The largest BBN TC2000 delivered to date is the 128 processor machine at Lawrence Livermore National Laboratory.

resulting Parallel Data Distribution Preprocessor (PDDP) is expected to be useful on the BBN TC2000, and will likely be even more useful on future machines of this general architectural class employing substantially faster processors. The latency of communication networks is not improving at the same rate as the speeds of microprocessors and this will make the successful exploitation of data locality even more important in the future.

REFERENCES

[1] TC2000 Fortran Reference, *BBN Advanced Computers, Inc.*, Cambridge, MA, 1991.

[2] E. D. Brooks III, *PCP: A Parallel Extension of C that is 99% Fat Free*, UCRL-99673, Lawrence Livermore National Laboratory, Livermore, CA, 1988.

[3] K. H. Warren, B. Gorda, and E. D. Brooks, III, *Programming in PFP*, UCRL-MA-107028, Livermore, CA, Lawrence Livermore National Laboratory, 1991.

[4] H. F. Jordan, "The force: A highly portable parallel programming language," *Proc. of the International Conference on Parallel Processing.* University Park and London: The Pennsylvania State University Press, August 1989, pp. II-112–II-117.

[5] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "A single-program-multiple data computational model for EPEX/FORTRAN," *Parallel Comput.*, vol. 7, no. 1, pp. 11–24, April 1988.

[6] K. H. Warren and E. D. Brooks III, "Gauss elimination: A case study on parallel machines," *Proc. of Compcon '91.* Los Alamitos, CA: IEEE Computer Society Press, February 1991, pp. 57–61.

[7] D. M. Dias and M. Kumar, "Preventing congestion in multistage networks in the presence of hotspots," *Proc. of the 1989 International Conference on Parallel Processing.* University Park and London: The Pennsylvania State University Press, August 1989, pp. I-9–I-13.

[8] D. Hensgen, R. Finkel, and U. Manber, "Two algorithms for barrier synchronization," *Int. J. Parallel Programming*, vol. 17, no. 1, pp. 1–17, 1988.

[9] B. Gorda, K. Warren, and E. D. Brooks III, *Programming in PCP*, UCRL-MA 107029, Lawrence Livermore National Laboratory, Livermore, CA, 1991.

[10] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Trans. Parallel and Distributed Syst.*, vol. 1, no. 1, pp. 6–16, January 1990.

[11] Inside the TC2000, BBN Advanced Computers Inc., Cambridge, MA, 1989.

[12] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing*, 2nd edition. New York: McGraw-Hill, 1990, pp. 113–174.

[13] L. H. Yang, E. D. Brooks III, and J. Belak, *A Linked-Cell Domain Decomposition Method for Molecular Dynamics Simulation on a Scalable Multiprocessor*, UCRL-JC-109752, Lawrence Livermore National Laboratory, Livermore, CA, 1992 (submitted to *Scientific Programming*).

[14] S. Picano, E. D. Brooks III, and J. E. Hoag, *Assessing Programming Costs of Explicit Memory Localization on a Large Scale Shared Memory Multiprocessor*, UCRL-JC-109751, Lawrence Livermore National Laboratory, Livermore, CA, 1992. *Scientific Programming*, vol. 1, no. 1, pp. 65–76, Fall 1992.

[15] E. D. Brooks III, "Effective use of shared memory multiprocessors," *Proc. Third International Conference on Supercomputing*, May 15–20, 1988, Vol. II, pp. 365–371, International Supercomputing Institute, Inc.

[16] G. Fox et al., "Fortran D Language Specification," Rice COMP TR90-141, Rice University, December, 1990.