PFP: A Scalable Parallel Programming Model *

Brent C. Gorda and Karen H. Warren
Lawrence Livermore National Laboratory
Livermore, CA 94550

Abstract

The Parallel Fortran Preprocessor (PFP) is a programming model for Multiple Instruction Multiple Data (MIMD) parallel computers. It provides a simple paradigm consisting of data storage modifiers and parallel execution control statements. The model is lightweight and scalable in nature. The control constructs impose no implicit synchronizations, nor do they require off-processor memory references.

The model is portable. It is implemented as a source to source translator which requires very little support from the back end compiler. The implementation has an option to produce serial code which can then be compiled for serial execution.

1 Introduction

Parallel programming is a complex task. Programmers are required to be aware of memory allocation, scheduling, communication, and synchronization of processors. Programming models can either hide these tasks, making them implicit, or leave them exposed so that the user has explicit control. Controlling implicit models is often difficult and requires knowledge of various tricks to take advantage of the subtle features.

In PFP we take the attitude that things should be explicit for the programmer. With simple tools in hand, the programmer is better able to utilize the efficiency of the hardware for an application. Abstractions of the serial model are built upon, giving the programmer a familiar concept with which to work.

PFP is a split-join [1] model, similar to Harry Jordan's Force [2] and the IBM SPMD [3] programming models, with added support for clustering of teams of processors. The concept of teams and team splitting, further explained below, permits efficient exploitation of nested concurrency with any number of processors.

Programmers are free from the usual burden of worrying about specific numbers of processors.

The programming model for PFP and PCP (Parallel C Preprocessor - the C equivalent) is scalable in nature. The control constructs contain no implicit serialization nor synchronization. All such primitives are placed directly in the hands of the applications programmer. Typical applications require relatively few instances of these mechanisms. They are generally used to guarantee that data dependencies are met before execution of a section of code. Programs which have no data dependencies are free to execute without the restrictions imposed by them.

The model contains lightweight parallel execution mechanisms in the form of looping constructs, parallel sections, and nested parallel team semantics. Under the assumption that non-local memory references are expensive relative to local references, it is important to note that PFP control constructs involve 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 which were previously deemed not heavy enough to amortize the overhead.

The memory model presented to the user is straightforward and simple. Automatic data is private to each process in the group. Non-automatic data is either shared or private, an option which is controlled by storage attributes in the source code as well as being a preprocessor option. Teamprivate data also exists and is data which is accessible to a restricted group of processors acting as a team. The program state, required for parallel constructs, is also saved in private memory.

Many of the actual implementation vehicles, including the most recent 128 processor BBN TC2000, have a caching mechanism. Private memory and stack memory are cachable in these instances. In addition, the state required for the control parallelism constructs is cached. This further reduces the expected overhead involved in using the control constructs.

---

*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.

U.S. Government Work Not Protected by U.S. Copyright
2 Split-Join Programming

In the traditional fork-join parallel programming model a single processor enters the user's main program. The program uses directives to acquire more processors as concurrency is encountered in the code. Examples of this model are PCF [4] in which the system may utilize more processors for designated program sections if it desires, and the Cray autotasking model [5] which automatically distributes loop iterations to a number of processors determined by the system and which itself obviously is not portable.

In contrast, the PFP and PCP programming models are split-join in nature. In the split-join paradigm, the expense of acquiring and releasing processors is minimized by never releasing the processors during execution. All of the processors dispatched on a job enter the main program together and are then under direct user control. The number of processors is determined at run time; recompilation when the number changes is unnecessary.

This concept of a group, or team of processors, executing the entire code causes programmers to think about the application from a different viewpoint than with other parallel models. In the PFP model the programmer thinks of a team executing the code in the same way that a single processor executes a serial program. Similarly the entire program is done if any member calls exit or encounters an exception. Thus, the model encourages programmers to view the entire program as executing in parallel. The programmer looks for sections which then need to be serialized. Compared to the traditional approach of searching for parallelism in an application, this methodology has proven beneficial.

The PFP model utilizes the concept of team splitting. The programmer is able to split off groups of processors (teams) to execute logically distinct code sections. All parallel constructs work at the team level. A team has access to the parallel looping constructs, the barrier, locks and even further team splitting. The total number of processors is conserved in the team splitting process; 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 the processors within a team complete their work, they rejoin the parent team with no serialization.

Nested concurrency can be easily exploited via team splitting. The split construct takes an optional weight parameter which controls the number of processors to send into each team. In this manner, load balancing is placed under user control.

Splitting the size of the team into smaller subteams as nested concurrency is encountered is counter intuitive. The goal, however, in exploiting 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 programming model is its full featured, bottleneck free, implementation through a highly portable preprocessor.

3 Memory Model

Memory class modifiers exist to control where data is allocated in a program. In the split-join programming paradigm three types of memory are provided to fully exploit the concept of team splitting. These are:

- memory which is private to a processor, private memory,
- memory which is shared among all processors, shared memory,
- memory which is shared among the members of a given team or grouping of processors, but private to it, teamprivate memory.

Private memory is implemented on the processor which requires access to it. On the BBN TC2000 shared memory is implemented in the interleaved shared memory facility. Teamprivate memory is allocated as an array in the interleaved shared memory, indexed by a team descriptor which is unique to a given team.

By default, all statically allocated data is private; a copy exists on each processor. Stack, or auto, data is also private to a processor and is stored in the processor's local memory.

4 Synchronization

Both barrier synchronization and locks are implemented in PFP. Barrier synchronization forces all of the team members to wait at the barrier until the last processor arrives. A bottleneck free software implementation [6] is used, requiring 30 to 40 microseconds to synchronize 32 processors on the BBN TC2000.

The notion of a lock is provided as a base type in the model. A processor attempting to acquire a lock spawait until the lock is unlocked and then indivisibly
locks it. Locks may be declared by the user in either shared memory or teamprivate memory.

5 Flow Control

In the basic PFP model, a team of processors is executing the program at a time. Similar to a single processor execution, the entire team knows the bounds on any loops as well as the values of shared variables. Each member of the team thus knows the number of iterations in a shared loop, the bounds for a split, and the location of locks.

In PFP, each team member is responsible for finding its own path through the code in the presence of loops and splits. Each processor has the required information in its memory, and can thus execute the constructs with no remote memory penalties.

5.1 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 PFP code 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
       <executable statements>
    endmaster

Note that a master block does not provide a serial critical region in the usual sense. If team splitting has occurred, several teams exist and is executing asynchronously with its own master. A race condition could exist for access to shared data within a master block if it is possible for the two teams to encounter the block asynchronously. Because of team splitting, master blocks do not necessarily have the Amdahl's law impact that one might otherwise think they would have.

5.2 Doall

The doall loop is the PFP concurrent equivalent of the Fortran language do loop. It achieves a fine-grained parallelism by dividing up the passes of the do loop among the members of the team:

    doall i = <start>,<stop>,<step>
       <executable statements>
    enddoall

The indices of the loop are interleaved among the members of the executing team. Doall loops may be nested arbitrarily. The team index inside the loop body is set to 0 and the team size to 1 to permit enclosed PFP constructs to work correctly.

5.3 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:

    call 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 doall loop, to ensure that the preceding work is complete before any processor is allowed to continue on. A fast algorithm [6] which has no hot spots or critical regions has been implemented for PFP runtime support.

5.4 lock, unlock

Concurrency must be inhibited in a statement that reads, modifies, and then writes a variable to which many processors are modifying. 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.

PFP offers spin wait locks that are implemented by variables of the lock data type which has the two states locked and unlocked.

    shared lock lock1

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

    lock(lock1);
    <critical section>
    unlock(lock1);
The lock variable may be declared in any of the memory segments of the language. If the lock variable is shared, then the critical section is global. If teamprivate, then the critical section is local to the team. A lock declared private will not be visible to other processors.

5.5 Split

To divide up a number of tasks, which is known at compile time, among sub-teams which are split from the parent, one uses static team splitting:

```plaintext
split [weight1]
    <executable statements for task 1>
and [weight2]
    <executable statements for task 2>
...
and [weightn]
    <executable statements for task n>
endsplit
```

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

5.6 splitall

The dynamic version of team splitting is the splitall loop:

```plaintext
splitall i = <start>, <stop>, <step>
    <executable code>
endsplitall
```

When a team encounters a splitall loop, it disassociates into a number of sub-teams to which the indices of the loop are interleaved.

6 Summary and Conclusions

PFP is a unique, versatile and powerful programming tool for MIMD parallel processors. Not only is it portable, but it adds only a few constructs to a serial code, and at the same time allows the user to take advantage of data locality which becomes an important factor on massively parallel machines.

The solution to high performance on the BBN TC2000 and any future scalable system supporting shared memory is two-fold. For memory references, the efficient exploitation of data locality is key. The PCP split-join model makes memory locality an explicit attribute with which the user must be involved.

The other requirement for a successful scalable system is that scalable control structures be implemented. The PFP control constructs contain no implicit serialization nor synchronization.

PFP, with its pre-scheduled loops, team association and absence of implicit barrier synchronization, is capable of scaling to thousands of processors with no increase in overhead. The notion of programming for a team of processors is also convenient as the user may detach himself from the concern of exactly how many processors are at his disposal. Thus the program will make effective use of 10, 100 or 1000 processors with no change in coding.

References


