PRESTO: A System for Object-oriented Parallel Programming

BRIAN N. BERSHAD, EDWARD D. LAZOWSKA AND HENRY M. LEVY
Department of Computer Science, FR-35, University of Washington, Seattle WA 98195,
U.S.A.

SUMMARY
PRESTO is a programming system for writing object-oriented parallel programs in a multiproces-
sor environment. PRESTO provides the programmer with a set of pre-defined object types
that simplify the construction of parallel programs. Examples of PRESTO objects are threads,
which provide fine-grained control over a program's execution, and synchronization objects,
which allow simultaneously executing threads to co-ordinate their activities.

The goals of PRESTO are to provide a programming environment that makes it easy to express
concurrent algorithms, to do so efficiently, and to do so in a manner that invites extensions and
modifications. The first two goals, which are the focus of this paper, allow a programmer to
use parallelism in a way that is naturally suited to the problem at hand, rather than being
constrained by the limitations of a particular underlying kernel or hardware architecture. The
third goal is touched upon but not emphasized in this paper.

PRESTO is written in C++; it currently runs on the Sequent shared-memory multiprocessor
on top of the Dynix operating system. In this paper we describe the system model, its applicability
to parallel programming, experiences with the initial implementation, and some early performance
measurements.

KEY WORDS Design Languages Measurement Performance Parallel computing Software parallelism
Speed-up Efficiency

INTRODUCTION
PRESTO is a programming system for writing object-oriented parallel programs in a multiproces-
sor environment. PRESTO consists of an object-oriented language (C++1),
a library of basic tools constructed in this language, a run-time system providing
efficient support, and, most important but least tangible, a programming methodology.

Our first goal in designing and implementing PRESTO was to apply our experiences
in building distributed object-oriented systems2, 3 to the world of multiprocessors. In
distributed systems, an object-oriented programming paradigm makes it easier to think
about and to express concurrent algorithms. Problem decomposition and run-time
synchronization details can be neatly described by an object model. Each object is
responsible for solving some small part of an overall problem, and each is responsible
for maintaining its own (and only its own) internal consistency. These are exactly the
qualities that are needed when building parallel applications for a multiprocessor.

0038-0644/88/080713-20$10.00
© 1988 by John Wiley & Sons, Ltd.

Received 16 October 1987
Revised 4 February 1988
Our second goal was to provide efficient concurrency and synchronization mechanisms. The primitives provided by many existing parallel programming systems are so expensive that they become the major factor in determining the structure of applications. (For example, one may be forced to design algorithms that use only as many threads of control as there are physical processors.) Our experience (and common sense) suggested that the construction of parallel applications — a daunting task under the best of circumstances — was made considerably more difficult by such constraints. PRESTO allows the programmer to use parallelism in the manner most natural to the problem at hand, with minimal external performance constraints arising from an underlying kernel or hardware architecture.

Our third goal was to provide an 'open' environment that could be used as a 'toolkit' for building efficient support for a variety of 'models' of parallel programming. Most parallel programming systems present themselves in terms of a fixed set of primitives running on top of a closed run-time kernel. The primitives together with the kernel define a 'model' of parallel programming that, while pleasing to the implementor, may not always be satisfactory to the application programmer. In PRESTO, it is possible to redefine the behaviour of the lowest level system primitives. New constructs (and thus new abstractions) for constructing parallel applications can be introduced quickly, and without the level of overhead normally associated with a layered system.

This paper concentrates on describing PRESTO in terms of the first two goals: its object-orientation and its performance, and the impact of these characteristics on parallel programming. The third goal, that of providing an open environment for building parallel programming systems, is fully described in a companion paper. The examples in this paper are drawn from what could be called the 'default' PRESTO programming model — a Mesa-like environment where threads, monitors and condition variables define the basic primitives for the programmer.

The remainder of this paper discusses the implementation language for PRESTO (C++), the use of objects in building parallel programs, a few of the default PRESTO objects, the use of PRESTO on a multiprocessor and some preliminary performance measurements.

**PRESTO AND THE C++ PROGRAMMING LANGUAGE**

PRESTO is implemented in C++. To quote from the C++ reference manual:

> C++ is a superset of the C programming language that retains the efficiency and notational convenience of C, while providing facilities for type checking, data abstraction, operator overloading and object-oriented programming.

We chose C++ for three reasons, each touched upon in the above quotation. First, we wanted an object-oriented programming language. Secondly, C++ is implemented as a preprocessor to C, making it portable to any system with a C compiler. (Although PRESTO exists now on only one machine, we intend to port it to other multiprocessors as they become available to us.) Thirdly, we wanted PRESTO to be widely used, and C++ is relatively easy to learn for C programmers.

Object-oriented programming in C++ is made possible by the concept of a *class*. A class is a user-defined data type allowing the programmer to specify an object in terms of its data representation and operations. As an example, the class definition for a simple stack of integers might appear as
// This is a comment.
class Stack {
    // Private Data
    int st_size;       // maximum size
    int st_sp;         // current stack pointer
    int *st_elements;  // data

    // Private Operations
    void st_growstack();  // make the stack larger

    // Public Operations
    Stack (int maxsize) { st_size = maxsize;
    st_sp = 0;
    st_elements = new int[st_size];
    }  // CONSTRUCTOR

    Stack() { delete st_elements; }
    int st_depth() { return st_sp; } // current depth

    virtual void (push(int newelement)
        { if (st_sp == st_size)
            st_growstack();
        st_elements[st_sp++] = newelement;
    } // ensure we have room

    virtual int pop()
        { if (st_depth() == 0) // nothing left?
            // ERROR HANDLER HERE
            else
                return st_elements[--st_sp];
    }
};

Each instance of a Stack has its own private version of the class's variables. An object's operations are shared by all instances of the object's class. The declarations in the private section specify those parts of the class accessible only from within the operations themselves. If another object creates a stack,

    Stack *S = new Stack(100);  // Invoke the CONSTRUCTOR; maxsize = 100

then only the operations

    S->push(x);
    x = S->pop();
    sz = S->depth();  // Invoke the DESTRUCTOR

can be performed on S. S's private operations and private data can be referenced from within these public operations. The process of performing an operation is called an
invocation.

The keyword virtual in the class definition indicates that push() and pop() can be redefined by any classes that are defined as a sub-class of Stack. In C++, a sub-class can be derived from a super-class, allowing classes to exist hierarchically. The sub-class's qualities are those it defines for itself, as well as those inherited from its super-class. If a sub-class redefines any of its inherited virtual operations, the new definitions take precedence. The sub-class satisfies the isa relation on its super-class. That is, if class SynchronizedStack is a sub-class of class Stack,

```cpp
// Sub-class isa Super-class
class SynchronizedStack : Stack {
    //
    // Serialize access to the stack
    //
};
```

then an instance of class SynchronizedStack can be used anywhere an instance of class Stack is expected — SynchronizedStack isa Stack. A synchronized stack object can ensure that concurrent operations on the private data defined in its super-class are serialized, without having to redefine the implementation of those operations. A more complete definition for SynchronizedStack appears later in this paper.

An object's operations may also include constructor and destructor routines. These are procedures that are called automatically when the object is created and deleted, respectively, allowing the object to specify an initialization and termination sequence. New objects can be created on the call-stack by declaring them within a basic block, statically by declaring them outside of any block, or on the heap, by using the new operator.

The definition for an operation can either be included in the class definition itself or given elsewhere. To define the operation st_growstack() separately from the declaration, it is necessary to use the qualifying syntax '::'.

```cpp
//
// Class::Operation qualifies Operation under Class
//
void Stack::st_growstack() // Class::Operation
{
    int j;
    int newsize = st_size * 2; // double size
    int *newstack = new int[newsize];
    for (j = 0; j < st_size; j++) // copy old stack
        newstack[j] = st_stack[j]; // into new
    st_size = newsize;
    delete st_stack;
    st_stack = newstack;
}
```

(These programming segments are meant only to provide enough exposure to C++ so that the reader can understand the remaining examples in this paper. For complete
**EXPLOITING THE OBJECT MODEL IN PARALLEL PROGRAMS**

PRESTO provides the programmer with several classes useful for writing parallel programs. These classes, and the environment in which they execute, help support two of the major goals of PRESTO — efficient execution and comfortable abstractions for expressing concurrency.

In PRESTO, all objects execute in a single address space shared by all processors, allowing for fast inter-object communication and synchronization through shared storage. The object model allows objects to exist in a 'safe' environment, making it difficult (although not impossible) for objects to trounce one another haphazardly. In a sequential object-oriented system, an object hides its data and its implementation. In PRESTO, an object hides not only its data and its implementation, but also its execution. That is, when a caller invokes an operation on an object, the caller is unaware whether that operation executes sequentially or in parallel. The implementor of an object determines the extent of parallelism that is appropriate to the object, much as he/she decides what data structures best suit the needs of the object. Dealing with concurrency in this manner simplifies the task of writing parallel programs.

The following sections describe the major classes used by PRESTO programs, and discuss how their design addresses the goals of the system.

**The thread class**

Thread objects (threads) are the building blocks of PRESTO parallel programs. As the basic unit of execution, threads consist conceptually of a program counter and a stack of invocation records. There are two essential operations that can be performed
on a thread. A thread can be created, allowing the creator to specify the thread’s qualities, such as its name and maximum storage requirements. Once created, a thread can be started executing some operation of some object, wherein it executes in parallel with the starting thread. Start, in fact, is an operation defined for threads; parameters include the object, the operation where the thread is to be started, and any parameters expected by that operation. For example,

```java
Stack *S = new Stack(100);
// Create a new thread named "Pusher" having id TID.
Thread *t = new Thread("Pusher", TID);
// Let t be responsible for pushing 43 onto the stack.
t->start(S, S->push, 43);
```

As noted earlier, PRESTO extends conventional object-oriented programming by allowing an object to hide and control not only its data and its implementation, but also its execution. The user of an object chooses between synchronous and asynchronous invocations, and the implementor of an object chooses between sequential and parallel execution. Table I shows how these choices fit together, and how their combinations affect the overall execution of a program.

An object cannot tell whether it is being invoked synchronously or asynchronously, and the user of an object cannot tell whether an invocation is being performed sequentially or in parallel. For example, the user of a matrix object is probably not concerned with whether the object implements multiplication by using hundreds of parallel threads or a single thread executing over the whole problem. Only the ultimate product is important. It is the responsibility of the matrix object to determine where, when and how much parallelism is dictated by a given invocation of the multiply operation. In cases such as this, synchronous invocation with parallel execution is most appropriate. The following code segment shows how a parallel matrix multiplication operation might be defined in PRESTO:

<table>
<thead>
<tr>
<th>User of object (U) chooses</th>
<th>Implementor of object (I) chooses</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronous</td>
<td>sequential</td>
<td>U invokes I’s operation. U blocks until I finishes, I runs single-threaded with U’s thread.</td>
</tr>
<tr>
<td>synchronous</td>
<td>parallel</td>
<td>As above, only I creates multiple threads and starts them executing its own operations. These threads compute in parallel.</td>
</tr>
<tr>
<td>asynchronous</td>
<td>sequential</td>
<td>U creates a new thread and starts it executing I’s operation. U continues, and I runs with the single thread that U started within it. When I returns, its thread is destroyed.</td>
</tr>
<tr>
<td>asynchronous</td>
<td>parallel</td>
<td>U creates a new thread and starts it executing I’s operation. U continues, while I creates multiple threads and starts them executing I’s operations.</td>
</tr>
</tbody>
</table>
class Matrix {
    int **ma_elems;
    int ma_rows;
    int ma_cols;

    void ma_dotproduct(int *el, Matrix *M, int i, int j) {
        // Compute the dot product of the i'th row
        // of "this" (in ma_elems) and
        // the j'th column of M. Store in *el.
    }

    public:
    Matrix(int rows, int columns); // CONSTRUCT new matrix
    int numcolumns();
    int numrows();
    Matrix *multiply(Matrix *M); // multiple "this" by M
    // other operations...
}

// Create a separate thread to compute each element in the
// product matrix in parallel.
Matrix* Matrix::multiply (Matrix *M)
{
    Matrix *P = new Matrix(ma_rows, M->numcolumns()); // product
    for (int i = 0; i < ma_rows; i++)
        for (int j = 0; j < M->numcolumns();)
            Thread *t = new Thread("multiplier");
            t->start(this, this->ma_dotproduct,
             &P[i][j], M, i, j); // arguments to operation

    // Wait here until all threads terminate
    //
    return P;
}

Alternatively, the insertion of a new entry into a directory object is a non-parallel operation, best handled asynchronously by the object doing the insertion, not by the directory object itself. The insert invocation might appear as:

    // Create the directory
    Directory *dir = new Directory("my_directory");

    // Create a thread to do the insertion
    Thread *t = new Thread("dir_inserter");

    // Start the thread doing the insertion of some file.
    t->start(dir, dir->insert, someFileName, someFileContents);

    // Run in parallel with the dir->insert routine.
    // Its termination is transparent.
A thread may only be started once. It executes either until it terminates itself, or until it returns from the operation in which it was started. A thread may join on another thread, causing the joining thread to block until the joinee finishes. The joinee’s return value from the operation in which it started is returned to the joining thread when it resumes.

Each thread has its own call-stack. Any objects declared on that call-stack are visible only from within the thread to which the call-stack belongs. Objects requiring references from more than one thread must be heap allocated or static. If data is to be shared between threads, then it should be declared as such within the object’s definition.

The synchronization class

Although a thread can be executing in only one object at a time, it is possible to have multiple threads executing within a single object simultaneously. In a multiprocessor system, true concurrency can occur. To provide a controlled environment for multi-threaded objects, PRESTO provides two basic classes of synchronization objects: relinquishing and non-relinquishing locks.

Relinquishing locks

A thread executes until it is pre-empted, terminates or voluntarily relinquishes the processor by performing an operation on a relinquishing object. The simplest relinquishing objects are those defined by the class Lock. The two primary operations on locks are lock() and unlock().

```
// l is a reference to a Lock (Lock* l)
l->lock();
    // critical code
l->unlock();
```

Return from a lock() operation indicates that the caller holds the lock. A lock may be held by only one thread at a time. A thread trying to lock an already held lock relinquishes the processor on which it is running, allowing another ready thread to execute on that processor. The relinquishing thread is made ready for execution when the lock becomes free.

Non-relinquishing locks

Hardware atomic locks serve as the basis for the non-relinquishing synchronization object Spinlock. Spinlocks have a potential performance advantage over simple relinquishing locks. It is less expensive to acquire and release a non-relinquishing lock. Further, if the average waiting time is less than the time to relinquish and reacquire a processor, non-relinquishing locks are more efficient.

As with simple relinquishing locks, there are two operations on spinlocks, lock() and unlock().

```
// s is a reference to a Spinlock (Spinlock* s)
s->lock();
    // spin here if already locked
```
The thread that most recently locked the spinlock is the lock’s owner. A thread relinquishes ownership by unlocking the spinlock. A spinlock can have only one owner, but unlike relinquishing locks, a thread trying to lock an owned spinlock consumes CPU cycles polling the lock until it becomes free. If a thread tries to lock a spinlock that it already owns, the thread will spin forever. The implementation of spinlocks causes a thread to become non-pre-emptible once it acquires one.

More sophisticated synchronization classes

Monitors and condition variables

Although straightforward and easy to understand, simple relinquishing locks can be difficult to use and thus prone to misuse. A more refined relinquishing synchronization mechanism is available through monitors and condition variables.* These work together to provide Mesa-like synchronization semantics. These work together to provide Mesa-like synchronization semantics. As noted in the introduction, we view this as the 'default' PRESTO programming model, but we encourage users to build (and share) support in PRESTO for other parallel programming models when this is dictated by their applications.

In PRESTO, a section of critical code is surrounded by an entry and exit invocation on a monitor object. If several operations must coexist within the same monitor, the programmer is obligated explicitly to name the monitor within each operation. Although there is no direct compile-time support for monitors, it is not necessary for the programmer to make explicit calls to a monitor’s entry and exit routines when writing a monitored object. PRESTO provides a type MONITOR that can be used to guard access to blocks of code:

```c
{ // example of monitored block of code controlled by Monitor *m;
  MONITOR ENTRY(m);   // ENTRY is a nice sounding dummy variable
  // . . . code here
} // m is automatically released when ENTRY goes out of scope here
```

is equivalent to

```c
{
  m->entry();
  // . . . code here
  m->exit();
}
```

but is syntactically cleaner. Furthermore, the first form makes it impossible that the programmer will forget to explicitly release the monitor, even if the code returns from

---

* Condition variables are really condition objects, but the former terminology is well-established, and is therefore retained.
within the block. The constructor for a MONITOR object enters the named monitor, and remembers it. When the MONITOR object goes out of scope (at the bottom of a block), its destructor is automatically called. Within the destructor, the entered monitor is exited.

Only one thread can be active within a monitor at any one time. That thread is called the monitor's owner. When a thread attempts to enter a monitor that is already owned, the thread is blocked, relinquishing the processor on which the thread is executing. Eventually, the owner will release the monitor, causing the least recently executed thread waiting on the monitor to be resumed in an attempt to become the owner.

A thread may wait on a condition variable. When a condition variable is created, it must be bound to some monitor. The condition variable should only be used from within that monitor. It is an error to do otherwise. A thread waiting on a condition variable releases the associated monitor as it blocks. Another thread can signal the condition variable, causing the condition variable's longest waiting thread eventually to resume. The signaler continues to own the monitor until it waits or exits, so a signalled thread, since it must reacquire the monitor, does not execute immediately upon being signalled. A signal must be regarded merely as a hint that an acceptable state had been reached at some prior point. When a waiting thread next runs again, it should check that the condition on which it waited has remained satisfied since being signalled. A thread may also broadcast on a condition variable, causing all threads waiting on that condition variable to be signalled.

The following example demonstrates monitored access to stacks. The class SynchronizedStack is defined as a sub-class of the Stack class presented earlier. Synchronized stacks have all the characteristics of their super-class, but guarantee that access to the stack is atomic by redefining pop() and push() to require the possession of an exclusive private monitor. Further, a thread trying to pop() from an empty stack blocks, and does not resume until the stack becomes non-empty.

```java
// Sub-class
class SynchronizedStack : Stack {

    // Our own private data for synchronizing.
    Monitor *s_monitor;     // — for atomic access
    Condition *s_condition; // — reads are blocking

public:
    // Constructor to create a new stack
    Stack(int sz) {
        // Call CONSTRUCTOR of Super-class
        s_monitor = new Monitor("StackMonitor");
        s_condition = new Condition(s_monitor, "StackCondition");
    }

    void push(int newitem) {
        MONITOR ENTRY (s_monitor);
        // Qualify to the super-class operation
```
PRESTO

Stack::push (newitem);
if (depth() == 1) {
    // Signal if any could be waiting
    s_condition->signal();
}

int pop() {
    MONITOR ENTRY(s_monitor);
    int topitem;
    while (depth() == 0) {
        s_condition->wait();
        // Consider the signal only as a hint.
        // Must check depth again.
    }
    topitem = Stack::pop();
    return topitem;
}

Using instances of this class, multiple threads can safely share the same stack. Furthermore, operations written to operate on the super-class Stack can just as easily operate on a SynchronizedStack.

Atomic integers

To address the common situation where one would like simply to update a counter or some other integral value within an otherwise unsynchronized region of code, PRESTO provides an atomic integer class. The class AtomicInt guarantees multiple-reader, single-writer semantics for integers by automatically enclosing their reference within a spinlock's lock() and unlock(). AtomicInt supports the full complement of integer operations (assignment, increment, decrement, etc.). An AtomicInt can be used anywhere an integer is expected:

```cpp
{  
    AtomicInt a;
    AtomicInt b = 10;
    int c = a;
    int d;
    // w_lock -> write lock; w_unlock -> write unlock
    // r_lock -> read lock; r_unlock -> read unlock
    d = a++; // w_lock a, a++, d = a, w_unlock a
    b += c;  // w_lock b, increment b by c, w_unlock b
    c = a + b; // r_lock a; a' = a; r_unlock a;
                // r_lock b; b' = b; r_unlock b;
```
Atomic integers are an example of how the object-oriented programming model meshes well with the needs of parallel programs. Instances of class AtomicInt are responsible for ensuring their own synchronization and providing their own access semantics in a parallel environment. Users of the class are insulated from the details of the class's implementation, and are guaranteed of its correct operation.

SYSTEM ARCHITECTURE

PRESTO exists as a run-time library on Sequent Balance and Symmetry shared-memory multiprocessors. (PRESTO will soon also be operational on the DEC SRC Firefly, an experimental prototype multiprocessor workstation).

The Sequent's operating system is Dynix, a 4.2BSD UNIX† look-alike with support for shared memory. Dynix provides support for writing parallel programs, but this support is limited. The Dynix unit of execution is the UNIX process, an expensive ('heavyweight') and inflexible entity. Because the basic synchronization mechanisms are cumbersome to use, a 'parallel programming library' is provided. This library restricts the 'threadedness' of a parallel program to the number of physical processors in the system, prohibiting the design of algorithms that have hundreds (or even thousands) of independent threads of execution. Even if the parallel programming library were redesigned to remove this restriction, the performance of the basic system primitives would seriously limit the ways in which parallelism could be used. (We must emphasize that these limitations are not unique to Dynix and the Sequent, and that on balance we are delighted with the Sequent system.)

The basic role of the PRESTO run-time system is to map user's threads onto physical processors and to provide access to a global shares memory in which all objects reside. In the case of a system like Dynix, PRESTO maps threads onto Dynix processes, relying on the Dynix kernel to complete the mapping onto a physical processor. Although there are two levels of indirection required, a Dynix process can be permanently bound to a physical processor, so the second level of indirection is done only once. All details of the mappings are invisible to the PRESTO programmer.

The architecture of PRESTO adheres to the threaded object model described earlier. The system maintains a single scheduler object. The scheduler object keeps track of all threads that are runnable but not yet running. A thread becomes runnable when first started within an object, or when resumed by a synchronization object after blocking. Each processor in the system is represented by a processor object. There may be more processor objects than processors, but this is not the intention. One scheduler thread runs within each processor object, and that thread's only activity is to request runnable threads from the scheduler object. When a scheduler thread obtains a runnable thread from the scheduler object, the scheduler thread stops, and the processor on which the scheduler thread was running begins running the now-runnable

† UNIX is a trademark of AT&T, Bell Laboratories.
thread. When the newly-running thread blocks or terminates, the scheduler thread is resumed and continues to check for more runnable threads.

Simultaneous requests to the scheduler object from multiple scheduler threads are synchronized so that no thread can be scheduled on more than one processor at any instant. However, a thread may execute on different processors at different times. Migration occurs only if a thread is blocked and then resumed at some later time when some other processor is idle. Scheduler threads are an exception to this — they never migrate. A scheduler thread runs only on the processor for which it is scheduling. Figure 1 illustrates how the scheduler object, processor objects and physical CPUs are related. Because these objects interact with one another only through their operations, each can be easily replaced or modified without affecting the others. For example, the scheduler object could be changed to maintain multiple priority queues for threads rather than a single runnable queue. Since scheduler threads interact with the scheduler only through a GetAThread() operation, they would remain unaffected by the change.

The PRESTO scheduler eventually halts when there are no longer any runnable or running threads. At this point, all existing synchronization objects are destroyed. If any one of them indicates a waiting thread, the system declares deadlock and displays the state of all interminably blocked threads. Because a thread waiting on a spinlock is still technically executing, this very simple criterion for detecting deadlock fails if one or more threads are waiting for a spinlock to become free. More sophisticated halting semantics have been implemented through a redefinition of the scheduler. For example, a message-based discrete event simulation scheduler has been built that resolves deadlock arising from circular message dependencies when no threads are runnable.

PROGRAM STRUCTURE

In PRESTO, a user's parallel program consists of a set of class definitions for objects used in the program, and a set of implementation routines that define the operations for each class. In addition, the programmer must provide one operation for a system defined class called Main:

![Scheduler Object](image)

**Scheduler Object**

*ready thread queue*

---

**Processor Objects**

*Figure 1. PRESTO components*
The programmer links his code with the PRESTO library and obtains an executable program. The function main() required by the UNIX loader is already provided by the library. This routine creates an object of class Main and starts at least one thread in the operation Main::main() for that object.

The programmer may also provide two other operations, Main::init() and Main::done(). Main::init(), if provided, is called before the system begins executing; it can be used to override certain system default parameters such as the number of processors to use. Main::done(), if provided, is called when there are no more runnable threads, allowing PRESTO programs to clean up after themselves. Once running in Main::main(), the system is under the control of the programmer.

SOME EARLY PERFORMANCE FIGURES

This section presents early performance measurements for PRESTO. All figures represent measurements taken from a Sequent Balance 21000 with ten 32032 processors. As a baseline, a processor can do a null procedure call and return in approximately 15 μs, and can execute a single iteration of a for-loop in 7 μs.

Program performance

Figure 2 illustrates the performance of PRESTO when running a matrix multiplication algorithm over varying numbers of processors. The problem was decomposed equally among as many threads as there were processors, and each thread ran independently of the others. The two optimal curves are based on the performance of the algorithm when run with \( n \) threads on \( n \) processors. These are clearly best-case examples, since the scheduling and synchronization costs imposed by the algorithm are essential zero. Nevertheless, it shows that an optimal breakdown of an optimal problem can yield nearly optimal results under PRESTO. An implementation of the same algorithm directly on top of Dynix performs identically. This is not surprising since the processor speed is the same, and neither PRESTO nor Dynix are doing anything to assist (or hinder) the computation.

A difference does exist between the optimal and measured curves, and this same difference exists in the Dynix implementation of the same algorithm. It is primarily attributable to the start-up costs of the program and of initializing the processors. The matrices to be multiplied must first be initialized. Although data initialization could
be done in parallel, the program represented in Figure 2 does not do so. The data initialization costs are reflected only in the measured curves, not in the optimal ones. In addition to program initialization, PRESTO itself must be initialized. The time to initialize and begin executing on nine processors is much greater than for a single processor (the cost is roughly 55 ms per processor), but the total lifetime of the computation is much shorter. Consequently, the percentage of time doing work unrelated to the multiplication algorithm increases with the number of processors, and this appears as a break from the optimal curve when the total computation time gets very small. For longer-running computations, the initialization effects disappear.

The cost of threads

In Figure 2, the matrix multiplication algorithm was designed so that the number of threads was equal to the number of available processors. This is in some sense 'optimal' from a performance point of view.

In the case of matrix multiplication, designing an algorithm that is 'parametrized' by the number of processors is straightforward. In other problem settings, though, there may be a 'natural' decomposition of the problem into threads of control, and 'warping' this decomposition onto a specific number of processors may impose a
significant hardship. As noted, a key goal of PRESTO is to decrease the cost of parallelism so that the problem structure, rather than the underlying system, can be allowed to determine the way in which parallelism is used.

To demonstrate that this key goal was achieved, Figure 3 illustrates the performance of the PRESTO matrix multiplication algorithm as the number of threads is allowed to become very large (as many as 200 threads working on a 200×200 matrix). The Figure demonstrates that fine granularity using many threads can be inexpensive with PRESTO – an important advantage over many existing parallel programming systems. In PRESTO, the cost of several hundred threads is not much more than the cost of a few threads (except for the small cost of first scheduling each thread). Thus, in PRESTO, threads can be used as a ‘program structuring’ tool.

An interesting characteristic of Figure 3 is the small hump at around 25 threads for several of the curves. The hump is due to a decomposition of the problem inappropriate to the number of processors used. Let $E$ be the execution time of one thread on one processor, $n$ be the number of processors available and $T$ be the number of threads over which a computation is divided. Since $E$ is the same for all threads, execution essentially proceeds in lock-step. At each step, $T/n$ threads complete, except that when $n$ is not a factor of $T$, it is not possible to evenly distribute the final $T \mod n$ threads over the $n$ available processors. Consequently, the total computation time, $t_c$, is
When $T$ is small and $E$ is large (as it is with only 25 threads working on a 200×200 matrix), this 'tail effect' can be quite pronounced and manifests itself as the hump seen in Figure 3. The hump is largest when $T \mod n$ is small, but non-zero. For example, when $n$ is eight, seven processors are wasted during the final phase of the computation as only one thread remains to execute. With nine processors though, seven threads execute during the final phase, leaving only two processors idle. As would be expected, 25 threads on five processors produces no hump.

This phenomenon of some processors idling while others work is called starvation. When processors execute in lock-step, the amount of wasted processing time due to starvation effects is $E \times n_r$, where $n_r$ is the number of processors suffering from starvation. Even when processors execute asynchronously, unless all threads terminate concurrently, there must come a time near the end of a computation when there are fewer runnable threads than processors. Starvation can even become a factor when the 'optimal' number of threads is used but some delay exists between the starting time of the first thread and that of the last. Clearly, the negative impact of starvation diminishes with decreasing $E$. For fixed size problems, $E$ decreases with increasing $T$. If the overhead due to a large $T$ can be offset by preventing the degrading effects of starvation, appreciable performance benefits can be realized through very fine grained decomposition. PRESTO makes this possible.

Towards cheaper threads

The construction of a thread's call-stack is a significant contributor to thread creation cost. To mitigate this in PRESTO, threads are reclaimed upon termination for possible reuse. When the programmer requests a new Thread, the system checks if any reclaimed threads are available. If so, a new thread is not created; the reclaimed one is reinitialized and returned to the programmer. If not, a thread template is created and marked as incomplete. The thread template can be manipulated in the same ways as a complete thread. Eventually, the thread template will attempt to execute for the first time. When this happens, the reclaim pool is checked again. Only if it is empty the second time is an entirely new thread created and initialized with the values stored in the template.

This aggressive design, which is totally transparent to the programmer, significantly

<table>
<thead>
<tr>
<th>Processors</th>
<th>Threads created and destroyed</th>
<th>Elapsed time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100,000</td>
<td>44.0</td>
</tr>
<tr>
<td>2</td>
<td>100,000</td>
<td>28.2</td>
</tr>
<tr>
<td>3</td>
<td>100,000</td>
<td>21.6</td>
</tr>
<tr>
<td>4</td>
<td>100,000</td>
<td>18.1</td>
</tr>
<tr>
<td>5</td>
<td>100,000</td>
<td>15.8</td>
</tr>
<tr>
<td>6</td>
<td>100,000</td>
<td>14.0</td>
</tr>
<tr>
<td>7</td>
<td>100,000</td>
<td>12.8</td>
</tr>
<tr>
<td>8</td>
<td>100,000</td>
<td>11.8</td>
</tr>
</tbody>
</table>
reduces the cost of threads in situations where a number of threads are started simultaneously and run to termination without blocking; only as many call-stacks will be allocated as there are processors. A peculiar side-effect is that it is difficult to talk in a meaningful way about the 'cost of thread creation' in PRESTO, since this cost depends upon the style of use.

Table II shows the time to create and destroy PRESTO threads when every thread can be reclaimed. The table demonstrates two points. First, thread creation is relatively inexpensive. For the single processor case, the average time to create and destroy a thread is about 440 μs. Secondly, the rate at which threads can be created, while not linear with the number of processors, is much better than constant. The non-linearity arises because all threads originate from and return to a common pool, and access to this pool can be a bottleneck as locking must occur. Practically though, bursty periods of thread creation are usually the result of a single thread co-ordinating the activities of the new threads, so high contention is unlikely.

The cost of synchronization

There are two types of synchronization costs in a parallel program: non-competitive and competitive. A non-competitive cost is incurred whenever a thread accesses a synchronization object and is able immediately to become its owner. The programmer pays the competitive cost whenever a thread must wait for some other thread to relinquish ownership, or when the thread blocks on a condition variable. The competitive cost always includes, and is therefore greater than, the non-competitive cost. Table III shows these costs for four different synchronization operations. Only the overhead involved in actually blocking and unblocking a thread is reflected in these figures.

The top half of the table represents non-competitive synchronization overhead, and the bottom, competitive. Lock_test, monitor_test, spin_test and atom_test show the times required for threads to acquire and release a simple relinquishing lock, monitor, spinlock and atomic integer, respectively. The 21 μs required to use a spinlock is due to the slowness of the hardware atomic locks available on the Sequent. Since spinlocks serve as the basis for all other synchronization primitives, their lack-lustre performance negatively influences the other timings. When atom_test is run with two processors,

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Processors</th>
<th>Threads</th>
<th>Iterations</th>
<th>Elapsed time (s)</th>
<th>Average time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-competitive</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock_test</td>
<td>1</td>
<td>1</td>
<td>1,000,000</td>
<td>94.9</td>
<td>94.9</td>
</tr>
<tr>
<td>monitor_test</td>
<td>1</td>
<td>1</td>
<td>1,000,000</td>
<td>153</td>
<td>153</td>
</tr>
<tr>
<td>spin_test</td>
<td>1</td>
<td>1</td>
<td>1,000,000</td>
<td>21.2</td>
<td>21.2</td>
</tr>
<tr>
<td>atom_test</td>
<td>1</td>
<td>1</td>
<td>1,000,000</td>
<td>25.7</td>
<td>25.7</td>
</tr>
<tr>
<td>Competitive</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock_test</td>
<td>2</td>
<td>2</td>
<td>1,000,000</td>
<td>158.1</td>
<td>158.1</td>
</tr>
<tr>
<td>monitor_test</td>
<td>2</td>
<td>2</td>
<td>1,000,000</td>
<td>238.3</td>
<td>238.3</td>
</tr>
<tr>
<td>switch_test</td>
<td>1</td>
<td>2</td>
<td>100,000</td>
<td>123.9</td>
<td>1239</td>
</tr>
<tr>
<td>switch_test</td>
<td>2</td>
<td>2</td>
<td>100,000</td>
<td>73.1</td>
<td>731</td>
</tr>
<tr>
<td>atom_test</td>
<td>2</td>
<td>2</td>
<td>1,000,000</td>
<td>27.7</td>
<td>27.7</td>
</tr>
</tbody>
</table>
the elapsed time increases slightly over the single-processor case. This is due to an optimization for spinlocks (on which atomic integers are based) biased towards non-competitive acquisition. When the optimization is removed, one processor performs no better than two.

The time required for two threads to switch back and forth on a condition variable is shown for both one and two processors in switch_test. Each thread enters the monitor, signals a condition variable, and then waits on that condition variable, relinquishing the monitor (and the processor on which it is running). Although only one thread can be active in the monitor at any instant, two processors perform substantially better than one. The reason is that the context switch times for the waiting and signalled threads can be overlapped so that one thread can be switched in while the other is being switched out. With a single processor, this is not possible.

CONCLUSIONS

PRESTO is both a production system for use in writing everyday parallel programs and a flexible research tool with which various scheduling, synchronization and granularity issues can be explored.

The former goal is met by joining the classical notions of concurrent programming with the powerful concepts of object-oriented design. Objects can be made completely responsible for their own execution, as well as modification and presentation. This relieves the user of an object from concern about potential misuse in a parallel environment. By exploring the class mechanism of the language, programmers can derive parallel, inherently safe objects (such as a synchronized stack) from simpler, well-understood, sequential versions. A key point, emphasized in this paper, is that the performance of PRESTO’s primitives is sufficiently good that the ‘natural’ decompositions of problems, rather than artificial constraints imposed by the system, can be the determining factor in the structure of parallel algorithms.

The utility of PRESTO as a research vehicle arises from its underlying structure. A system component (the scheduler, a processor, even a thread) can be redefined through inheritance without affecting the other components.

PRESTO is not a toy. It is the current system of choice for parallel programming at the University of Washington. Certain applications have been built on top of the ‘default’ Mesa-like environment; for example, a parallel solution package for queueing network performance models. Other applications have customized certain aspects of PRESTO, taking advantage of its ‘open’ design; a parallel Othello program involves a new PRESTO scheduler; an instrumentation package for parallel programs involves an extension of threads to include monitoring capabilities.

ACKNOWLEDGEMENTS

We would like to thank Kenneth Almquist, Tom Anderson, Jeff Chase, and David Wagner for their user-view feed-back on the design and implementation of PRESTO. They, along with Ellen Ratajak, Doug Comer, and the referees, provided many helpful comments concerning this paper. Our work is supported by the U.S. National Science Foundation (Grants No. CCR-8619663, CCR-8700106, and CCR-8703049), the Naval Ocean Systems Center, US WEST Advanced Technologies, the Washington Technology Center, the USENIX Association, and Digital Equipment Corporation (the Systems Research Center and the External Research Program).
REFERENCES