The following paper was originally published in the
Proceedings of the 2nd USENIX Windows NT Symposium
Seattle, Washington, August 3–4, 1998

A System for Structured High-Performance Multithreaded Programming in Windows NT

John Thornley, K. Mani Chandy, and Hiroshi Ishii
California Institute of Technology

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

A System for Structured High-Performance Multithreaded Programming in Windows NT

John Thornley, K. Mani Chandy, and Hiroshi Ishii
Computer Science Department
California Institute of Technology
Pasadena, CA 91125, U.S.A.
{john-t, mani, hishii}@cs.caltech.edu

Abstract

With the advent of inexpensive multiprocessor PCs, multithreading is poised to play an important role in computationally intensive business and personal computing applications, as well as in science and engineering. However, the difficulty of multithreaded programming remains a major obstacle. Windows NT support for threads is well suited to systems programming, but is too unstructured when multithreading is used for the purpose of speeding up program execution. In this paper, we describe a system for structured multithreaded programming. Thread creation operations are multithreaded variants of blocks and loops, and synchronization objects are based on Boolean flags and integer counters. With this system, most multithreaded program development can be performed using traditional sequential methods and tools. The system is integrated with Windows NT and Microsoft Developer Studio Visual C++. We are developing a variety of applications in collaboration with other researchers, to demonstrate the power of structured multithreaded programming on commodity multiprocessors running Windows NT. In one benchmark application (aircraft route optimization), we achieved better performance on a quad-processor Pentium Pro system than the best results reported on expensive supercomputers.

1. Introduction

In the past, high-performance multithreading has been synonymous with either scientific supercomputing, real-time control, or achieving high throughput on multi-user servers. The idea of dividing a computationally intensive program into multiple concurrent threads to speed up execution on multiprocessor computers is well established. However, this kind of high-performance multithreading has made very little impact in mainstream business and personal computing, or even in most areas of science and engineering. Part of the reason has been the rarity and high cost of multiprocessor computer systems. Another part of the reason is the difficulty of developing multithreaded programs, as compared to equivalent sequential programs.

The recent advent of inexpensive multiprocessor PCs and commodity OS support for lightweight threads opens the door to an important role for high-performance multithreading in all areas of computing. Dual-processor and quad-processor PCs are now available for a few thousand dollars. Mass-produced multiprocessors with 8, 16, and more processors will soon follow. Windows NT [2] can already support hundreds of fine-grained threads with low overhead, even on single-processor machines, and future releases will be even more efficient. Examples of applications that could use multithreading to improve performance include spreadsheets, CAD/CAM, three-dimensional rendering, photo/video editing, voice recognition, games, simulation, and resource management.

The biggest obstacle that remains is the difficulty of developing efficient multithreaded programs. General-purpose thread libraries, including the Win32 API [1][8] supported by Windows NT, are well suited to systems programming applications of threads, e.g., control systems, database systems, and distributed systems. However, the interface provided by general-purpose thread libraries is less well suited to applications where threads are used for the purpose of speeding up program execution. General-purpose thread management is unstructured and synchronization operations are complex and error-prone. The unpredictable interactions of multiple threads introduce many problems (e.g., race conditions and deadlock) that do not occur in sequential programming. In many regards, general-purpose thread libraries are the assembly language of high-performance multithreaded programming.

In this paper, we describe our ongoing research to develop a system for structured high-performance multithreaded programming on top of the general-purpose thread support provided by operating systems such as Windows NT. The key attributes of our system are as follows:

- Structured thread creation constructs are based on sequential blocks and `for` loops.
- Structured synchronization constructs are based on Boolean flags and integer counters.
- Subject to a few simple rules, multithreaded execution is deterministic and produces the same results as sequential execution.
- Lock synchronization is provided for nondeterministic algorithms.
- Barrier synchronization is provided for efficiency.

Our system is supported at two levels: (i) Sthreads, a structured thread library, and (ii) Multithreaded C, a set of pragmas transformed into Sthreads calls by a preprocessor. Sthreads is easily implemented as a thin layer on top of general-purpose thread libraries. The Multithreaded C preprocessor is a simple source-to-source transformation tool that involves no complex program analysis. Therefore, the entire system is highly portable.

One of the major strengths of our system is that much of the development of a multithreaded application can be performed using ordinary sequential methods and tools. The Multithreaded C preprocessor is integrated with Microsoft Developer Studio Visual C++ [7]. Applications can either be built either as multithreaded applications (as indicated by the pragmas) or as sequential applications (by ignoring the pragmas). For deterministic applications, most development, testing, and debugging can be performed using the sequential version of the application. Absence of race conditions and deadlock can be verified in the context of sequential execution. Nondeterministic applications can usually be developed with large deterministic components.

The focus of our work is on commodity multiprocessors and operating systems, in particular multiprocessor PCs and Windows NT. However, our programming system is portable across platforms ranging from low-end PCs to high-end workstations and supercomputers. For this reason, the value of our work is not restricted to developers of applications for commodity systems. Our portable system for high-performance multithreaded programming also allows for high-end applications to be developed, tested, and debugged on accessible low-end platforms.

The remainder of this paper is organized as follows: in Section 2, we discuss the interface and performance of Windows NT support for multithreading; in Section 3, we describe our structured multithreaded programming system; in Section 4, we report in some detail on one particular application (aircraft route optimization) that we have developed using our system; in Section 5, we give a brief outline of several other applications that we are developing; in Section 6, we compare our system with related work; and in Section 7, we summarize and conclude.

2. Windows NT Multithreading

In this section, we describe the interface and performance of standard Windows NT thread support. Since our emphasis is on commodity systems and applications, Windows NT is the ideal platform on top of which to build our system for structured multithreaded programming. The following are particularly important to us: (i) the Windows NT thread interface provides the functionality that we require for our system, and (ii) the Windows NT thread implementation efficiently supports large numbers of lightweight threads.

2.1. Windows NT Thread Interface

Windows NT implements the Win32 thread API. This interface provides all the functionality that is needed for high-performance multithreaded programming. However, because the scope of the Win32 thread API is general-purpose, the interface is more complicated and less structured than is desirable when threads are used for the purpose of speeding up program execution. For this reason, we have built a less general, less complicated, and more structured layer on top of the functionality provided by the Win32 thread API.

The Win32 thread API is typical of other general-purpose thread libraries, e.g., Pthreads [6] and Solaris threads [4]. It provides a set of function calls for creating and terminating threads, suspending and resuming threads, synchronizing threads, and controlling the scheduling of threads using priorities. The interface contains a large number of constants and types, a large number of functions, and many optional arguments. Although all these operations have important uses in general-purpose multithreading, only a structured subset of this functionality is required for our purpose.

As with other general-purpose thread libraries, Win32 thread creation is unstructured. A thread is created by passing a function pointer and an argument pointer to a `CreateThread` call. The new thread executes the given function with the given argument. The thread can be created either runnable or suspended. After creation, there is no special relationship or synchronization between the created thread and the creating thread. For example, the created thread may outlive the creating thread, causing problems if the created thread references variables in the creating thread. Many unstructured operations are permitted on threads. For example, one thread can arbitrarily suspend, resume, or terminate the execution of another thread.

The Win32 thread API provides a large range of synchronization operations. A thread can synchronize on the termination of another thread, or on the termination of one or all of a group of other threads. Critical

section, mutex, semaphore, and event objects, and interlocked operations allow many other forms of synchronization between threads. There are many options associated with these synchronization operations, particularly with operations on event objects. All of these synchronization operations almost inevitably introduce nondeterminacy to a program. Nondeterminacy is an implicit part of most systems programming applications of threads, but is best avoided if possible in other applications, because of the difficulty it adds to testing, debugging, and performance prediction.

To summarize, Windows NT supports a rich but complex set of general-purpose thread management and synchronization operations. We implement a simpler layer for structured multithreaded programming on top of the Windows NT thread interface.

2.2. Windows NT Thread Performance

Lightweight multithreading is an integral part of our programming model. If multithreaded applications are to make an impact in commodity software, they must be able to execute efficiently on systems with differing numbers of processors, and dynamically adapt to varying background load conditions. The best way to achieve this is to build applications with large numbers of dynamically created lightweight threads that can take advantage of whatever processing resources become available during execution. The traditional scientific supercomputing model of one static thread per processor on an otherwise unloaded system is not sufficient in the commodity software domain.

We have performed experiments that demonstrate that Windows NT supports large numbers of lightweight threads efficiently. Specifically, on single-processor, dual-processor, and quad-processor Pentium Pro systems running Windows NT, we have demonstrated the following:

- Thread creation and termination overheads are low.
- Synchronization overheads are low.
- Hundreds of threads can be supported without significant performance degradation.
- On single-processor systems, multithreaded programs with many threads can run as fast as equivalent sequential programs.
- On multiprocessor systems, multithreaded programs with many threads can achieve good speedups over equivalent sequential programs.

We have developed many small and large programs which demonstrate that multithreaded Windows NT applications can execute efficiently on both single-processor and multiprocessor systems, without any kind of reconfiguration. We have found that good speedups are maintained with much larger numbers of threads than processors.

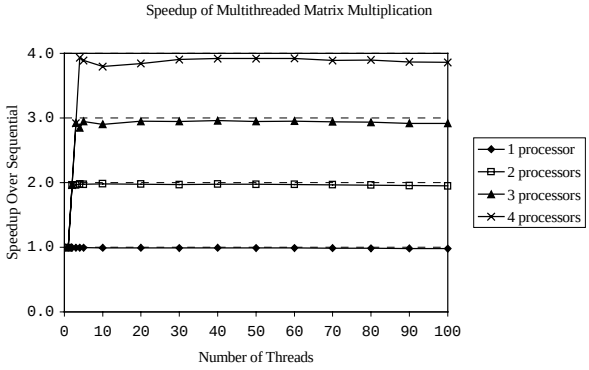

Figure 1: Speedup of multithreaded matrix multiplication over sequential matrix multiplication on one to four processors of a quad-processor Pentium Pro system.

As a simple example, Figure 1 shows the speedup of multithreaded matrix multiplication over sequential matrix multiplication for 1000-by-1000 element matrices on a 200 MHz quad-processor Pentium Pro system running Windows NT Server 4.0. A straightforward three nested-loops algorithm is used. Sequential matrix multiplication takes 50 seconds. In the multithreaded version, the iterations in the outer loop are evenly partitioned among the threads. (The intention of this example is to demonstrate Windows NT multithreaded performance characteristics, not optimal matrix multiplication algorithms.) Near-perfect speedups are achieved and are maintained with large numbers of threads.

To summarize, we have found that Windows NT efficiently supports large numbers of lightweight threads. Our structured multithreaded programming system is implemented as a very thin layer on top of Windows NT support for threads.

3. A System for Structured Multithreaded Programming

In this section, we describe the features of our structured multithreaded programming system. Since it is beyond the scope of this short paper to describe all the details of our system, we aim to give an overview of the fundamental constructs and development methods.

3.1. Overview

We are developing a two-level approach to structured multithreaded programming in ANSI C [3]. The higher level is a pragma-based notation (Multithreaded C), and the lower level is a structured thread library (Sthreads). In both Multithreaded C and Sthreads, thread creation

constructs are multithreaded variants of sequential block and `for` loop constructs. With Multithreaded C, the constructs are supported as pragma annotations to a sequential program. With Sthreads, exactly the same constructs are supported as library calls. At both levels, synchronization objects and operations are supported as Sthreads library calls.

The Sthreads library is implemented as a very thin layer on top of the Windows NT thread interface. Multithreaded C is implemented as a portable source-to-source preprocessor that directly transforms annotated blocks and `for` loops into equivalent calls to the Sthreads library. The programmer has the option of either using the pragmas and preprocessor or making Sthreads calls directly. The Sthreads library and Multithreaded C preprocessor are integrated with Microsoft Developer Studio Visual C++. Building a project automatically invokes the preprocessor where necessary and links with the Sthreads library.

3.2. Multithreadable Blocks and `for` Loops

Multithreadable blocks and `for` loops are ordinary blocks and `for` loops for which multithreaded execution is equivalent to sequential execution. In Multithreaded C, multithreadable blocks and `for` loops are expressed using the `multithreadable` pragma. It is obvious that the pragma can be applied blocks and `for` loops in which the statements or iterations are independent of each other. As a simple example, consider the following program to sum the elements of a two-dimensional array:

```
void SumElements(
    float A[N][N], float *sum, int numThreads)
{
    int i;
    float rowSum[N];

    #pragma multithreadable \
            mapping(blocked(numThreads))
    for (i = 0; i < N; i++) {
        int j;
        rowSum[i] = 0.0;
        for (j = 0; j < N; j++)
           rowSum[i] = rowSum[i] + A[i][j];
    }
    *sum = 0.0;
    for (i = 0; i < N; i++)
        *sum = *sum + rowSum[i];
}
```

Multithreaded execution of the `for` loop is equivalent to sequential execution because the iterations all modify different `rowSum[i]` and `j` variables. The arguments following the pragma indicate that multithreaded execution should assign iterations to `numThreads` different threads using a blocked mapping. There is a rich set of options that control the mapping of iterations to threads.

3.3. Synchronization Using Flags and Counters

Flags and counters are provided to express deterministic synchronization within multithreadable blocks and `for` loops. Flags support `Set` and `Check` operations. Initially, a flag is not set. A `Set` operation on a flag atomically sets the flag. A `Check` operation on a flag suspends until the flag is set. Once a flag is set, it remains set. Counters support `Increment` and `Check` operations. A counter has an initial value of zero. An `Increment` operation on a counter atomically increments the value of the counter. A `Check` operation on a counter suspends until the value of the counter reaches a given value. The value of a counter cannot be decremented. The only way to test the value of a flag or counter is with a `Check` operation. As a simple example, consider the following program to sum the elements of a two-dimensional array:

```
void SumElements(
    float A[N][N], float *sum, int numThreads)
{
    int i;
    SthreadCounter counter;

    SthreadCounterInitialize(&counter);
    #pragma multithreadable \
            mapping(blocked(numThreads))
    for (i = 0; i < N; i++) {
        int j;
        float rowSum;
        rowSum = 0.0;
        for (j = 0; j < N; j++)
           rowSum = rowSum + A[i][j];
        SthreadCounterCheck(&counter, i);
        *sum = *sum + rowSum;
        SthreadCounterIncrement(&counter, 1);
    }
    SthreadCounterFinalize(&counter);
}
```

Without the counter operations, multithreaded execution of the `for` loop would not be equivalent to sequential execution, because the iterations all modify the same `*sum` variable. However, the counter operations ensure that multithreaded execution is equivalent to sequential execution. In sequential execution, the iterations are executed in increasing order and the `SthreadCounterCheck` operations succeed without suspending. In multithreaded execution, the counter operations ensure that the operations on `*sum` occur atomically and in the same order as in sequential execution. Iteration `i` suspends at the `SthreadCounterCheck` operation until iteration `i – 1` has executed the `SthreadCounterIncrement` operation.

3.4. Sequential Development Methods

The equivalence of multithreaded and sequential execution of multithreadable blocks and `for` loops allows multithreaded programs to be developed using ordinary sequential methods and tools. The `multithreadable` pragma is an assertion by the programmer that the block or `for` loop *can* be executed in a multithreaded manner without changing the results of the program. It is not a directive that the block or `for` loop *must* be executed in a multithreaded manner. The Multithreaded C preprocessor has two modes: sequential mode in which the `multithreadable` pragma is ignored and multithreaded mode in which the `multithreadable` pragma is transformed into Sthreads calls. Programs can be developed, tested, and debugged in sequential mode, then executed in multithreaded mode for performance. In addition, performance analysis and tuning can often be performed in sequential mode.

The advantages of this approach to multithreaded programming are clear. However, the programmer is responsible for correct use of the `multithreadable` pragma. Fortunately, the rules for correct use of the pragma are straightforward and can be stated entirely in terms of sequential execution. In sequential execution, accesses to shared variables must be separated by `Set` and `Check` operations or `Increment` and `Check` operations, and the `Check` operations must not suspend. In multithreaded execution, the synchronization operations will ensure that accesses to shared variables occur atomically and in the same order as in sequential execution. Therefore, multithreaded execution will be equivalent to sequential execution.

3.5. Determinacy

Determinacy of results is an important consequence of the equivalence of multithreaded and sequential execution. If sequential execution is deterministic (which is usually the case), multithreaded execution will also be deterministic. Determinacy is usually desirable, since program development and debugging can be difficult when different runs produce different results. In many other multithreaded programming systems, determinacy is difficult to ensure. For example, locks, semaphores, and many-to-one message passing almost always introduce race conditions and hence nondeterminacy. However, nondeterminacy is important for efficiency in some algorithms, e.g., branch-and-bound algorithms.

3.6. Multithreaded Blocks and `for` Loops

Multithreaded blocks and `for` loops are blocks and `for` loops that *must* be executed in a multithreaded manner. Multithreaded execution is not necessarily equivalent to sequential execution. In Multithreaded C, multithreaded blocks and `for` loops are expressed using the `multithreaded` pragma. Unlike the `multithreadable` pragma, the `multithreaded` pragma is transformed into Sthreads calls by the Multithreaded C preprocessor in both sequential and multithreaded mode.

3.7. Synchronization Using Locks

Locks are provided to express nondeterministic synchronization, usually mutual exclusion, within multithreaded blocks and `for` loops. Our locks support the usual `Acquire` and `Release` operations. The order in which concurrent `Acquire` operations succeed is nondeterministic. Therefore, there is very little use for locks within multithreadable blocks and `for` loops. As a simple example, consider the following program to sum the elements of a two-dimensional array:

```
void SumElements(
    float A[N][N], float *sum, int numThreads)
{
    int i;
    SthreadLock lock;

    SthreadLockInitialize(&lock);
    #pragma multithreaded \
            mapping(blocked(numThreads))
    for (i = 0; i < N; i++) {
        int j;
        float rowSum;
        rowSum = 0.0;
        for (j = 0; j < N; j++)
           rowSum = rowSum + A[i][j];
        SthreadLockAcquire(&lock);
        *sum = *sum + rowSum;
        SthreadLockRelease(&lock);
    }
    SthreadLockFinalize(&lock);
}
```

Like the flag operations in the program in Section 3.3, the lock operations in this program ensure that the operations on `*sum` occur atomically. However, unlike the flag operations, the lock operations do not ensure that the operations on `*sum` occur in the same order as in sequential execution, or even in the same order each time the program is executed. Therefore, since floating-point addition is not associative, the program may produce different results each time it is executed. However, because execution order is less restricted, this program allows more concurrency than the program in Sec-

tion 3.3. This is an example of the commonly occurring tradeoff between determinacy and efficiency.

3.8. Synchronization Using Barriers

Barriers are provided to express collective synchronization of a group of threads in cases when thread termination and recreation is too expensive. Our barriers support the usual `Pass` operation. All the threads in a group must enter the `Pass` operation before all the threads in the group are allowed to leave the `Pass` operation. In current systems, the cost of N threads executing a `Pass` operation is less than the cost of creating and terminating N threads. Therefore, a typical use of barriers is to replace a sequence of multithreadable loops with a single multithreaded loop containing a sequence of barrier `Pass` operations. However, with modern lightweight thread systems such as Windows NT, we are discovering that barriers are required for efficiency in very few circumstances.

3.9. The Sthreads Interface

The examples that we have given so far are expressed using the Multithreaded C pragma notation. As we described previously, there is a direct correspondence between the pragma notation for thread creation and the Sthreads library functions that support thread creation. As a simple example, the following is program from Section 3.3 implemented using Sthreads:

```
typedef struct {
    float (*A)[N];
    float *sum;
    SthreadCounter *counter;
} LoopArgs;

void LoopBody(
    int i, int notused1, int notused2,
    LoopArgs *args)
{
    int j;
    float rowSum;
    rowSum = 0.0;
    for (j = 0; j < N; j++)
       rowSum = rowSum + (args->A)[i][j];
    SthreadCounterCheck(args->counter, i);
    *(args->sum) = *(args->sum) + rowSum;
    SthreadCounterIncrement(args->counter, 1);
}

void SumElements(
    float A[N][N], float *sum, int numThreads)
{
    int i;
    SthreadCounter counter;
    LoopArgs args;

    SthreadCounterInitialize(&counter);
    args.A = A;
    args.sum = sum;
    args.counter = &counter;
    SthreadRegularForLoop(
        (void (*)(int, int, int, void *))
        LoopBody,
        (void *) &LoopArgs,
        0, STHREAD_CONDITION_LT, N, 1,
        1, STHREAD_MAPPING_BLOCKED,
        numThreads,
        STHREAD_PRIORITY_PARENT,
        STHREAD_STACK_SIZE_DEFAULT);
    SthreadCounterFinalize(&counter);
}
```

Although this program is syntactically more complicated than the Multithreaded C version, it is considerably less complicated than the same program expressed using Windows NT threads. The mechanics of creating threads, assigning iterations to threads, and waiting for thread termination is handled within the Sthreads library call.

3.10. Performance Issues

In our multithreaded programming system, obtaining good performance is under the control of the programmer. The following issues must be taken into account:

- Multithreading overheads: Threading and synchronization operations are time consuming.
- Load balancing: Enough concurrency must be expressed to keep the processors busy.
- Memory contention: Locality in memory access patterns prevents memory contention.

The key tradeoff is granularity. If multithreading is too fine-grained, the overheads will swamp the useful computation. If multithreading is too coarse grained, the processors will spend too much time idle. Fortunately, Windows NT supports lightweight threads with low overheads.

3.11. Implementation and Performance on Windows NT

Sthreads for Windows NT is implemented as a very thin layer on top of the Win32 thread API. As a consequence, there is essentially no performance overhead associated with using Sthreads or Multithreadable C, as compared to using Win32 threads directly.

Multithreadable blocks and `for` loops are implemented as a sequence of `CreateThread` calls followed by a `WaitForSingleObject` call on an event. Terminating threads perform an `InterlockedDecrement` call on a counter, and the last thread to terminate performs a `SetEvent` call on the event. Flags are implemented directly as Win32 events. Counters are implemented as linked lists of Win32 events, with one event for every value on which some thread is waiting. Locks are implemented directly as

Win32 critical sections. Barriers are implemented as a pair of Win32 events and a Win32 critical section.

On a single-processor 200 MHz Intel Pentium Pro system running Windows NT Server 4.0, the time to create and terminate each thread in a multithreadable block or `for` loop is approximately 500 microseconds. The time to initialize and finalize a flag, counter, lock, or barrier is between 5 and 20 microseconds. The time to perform a synchronization operation on a flag, counter, lock, or barrier is less than 10 microseconds.

3.12. Current Status

The Sthreads implementation for Windows NT is complete and robust. Over the last year, we have developed many different multithreaded applications using Sthreads on quad-processor Pentium Pro systems running Windows NT. This academic year, we taught an undergraduate class at Caltech on multithreaded programming, with Sthreads used for all the homework assignments. We are in the process of implementing Sthreads on top of Pthreads for a variety of Unix platforms, including Sun UltraSPARC, SGI Origin, and HP Exemplar. We have developed a comprehensive, platform-independent test suite for Sthreads and a timing suite to compare the performance of different Sthreads implementations.

At the time of writing, the Multithreaded C preprocessor is still under development. We hope to make a prototype preprocessor available by the fourth quarter of 1998. In the meanwhile, we are developing multithreaded applications by making Sthreads calls directly. Because of the direct correspondence between the pragma annotations and Sthreads calls, the design and development of algorithms and programs is the same in Sthreads and Multithreaded C. The only difference is in the syntax.

4. An Example Application: Aircraft Route Optimization

In this section, we report on one example application that we have developed. The Aircraft Route Optimization Problem is part of the U.S. Air Force Rome Laboratory C3I Parallel Benchmark Suite [5]. For this application, we achieved better performance using Sthreads on a quad-processor Pentium Pro system running Windows NT than the best reported results for message-passing programs running on expensive Cray and SGI supercomputers with up to 64 processors. The flexibility of shared-memory, lightweight multithreading, and sequential development methods allowed us to develop a much more sophisticated and efficient algorithm than would be possible on a message-passing supercomputer.

4.1. The C3I Parallel Benchmark Suite

The U.S. Air Force Rome Laboratory C3I Parallel Benchmark Suite consists of eight problems chosen to represent the essential elements of real C3I (Command, Control, Communication, and Intelligence) applications. Each problem consists of the following:

- A problem description giving the inputs and required outputs.
- An efficient sequential program (written in C) to solve the problem.
- The benchmark input data.
- A correctness test for the benchmark output data.

For some of the problems, a parallel message-passing program is also provided. Rome Laboratory maintains a publicly accessible database of reported performance results.

The C3I Parallel Benchmark Suite provides a good framework for evaluating our structured multithreaded programming system. The problems are computationally intensive and involve a variety of complex algorithms and data structures. The sequential program provides us with a good starting point and a fair basis for performance comparison. The performance database allows us to compare our results with those of other researchers. For these reasons, we are developing multithreaded solutions to several of the C3I Parallel Benchmark Suite problems.

4.2. The Aircraft Route Optimization Problem

The task in the Aircraft Route Optimization Problem is to find the lowest-risk path for an aircraft from an origin point to a set of destination points in the airspace over an uneven terrain. The risk associated with each transition in the airspace is determined by its proximity to a set of threats. The problem involves realistic constraints on aircraft speed and maneuverability. The aircraft is also constrained to fly above the underlying terrain and beneath a given ceiling altitude.

The problem is essentially the single-source, multiple-destination shortest path problem with a large, sparsely connected graph. The airspace for the benchmark is 100 km by 100 km in area and 10 km in altitude, discretized at 1 km intervals. The 100,000 positions in space correspond to 2,600,000 nodes in the graph, since each position can be reached from 26 different directions. Because of aircraft speed and maneuverability constraints, each node is connected to only nine or ten geographically adjacent nodes. Therefore, the graph consists of approximately 2.6 million nodes and 26 million edges.

4.3. Sequential Algorithm

The sequential algorithm to solve the Aircraft Route Optimization Problem is based on a queue of nodes. Initially the queue is empty except for the origin node. At each step, one node is removed from the queue. Valid transitions from this source node to all adjacent destination nodes are considered. For each destination node, if the path to the node via the source node is shorter than the current shortest path to the node, the path to the node is updated and the node added to the queue. The algorithm continues until the queue is empty, at which stage the shortest paths to all reachable nodes have been computed.

The queue is ordered on path length so that shorter paths are expanded before longer paths. This has a significant effect on performance. Without ordering, longer paths are expanded, then discarded when shorter paths to the same points are expanded later in the computation. However, whether the queue is ordered, partially ordered, or unordered does not affect the results of the algorithm.

4.4. Multithreaded Algorithm

The most straightforward approach to obtaining parallelism in the Aircraft Route Optimization Problem is to geographically partition the airspace into blocks, with one thread (or process) responsible for each block. Each thread runs the sequential algorithm on its own block using its own local queue and periodically exchanges boundary values with neighboring blocks. This approach is particularly appealing on distributed-memory, message-passing platforms, because memory can be permanently distributed according to the blocking pattern. If the threads execute a reasonably large number of iterations between boundary exchanges, good load balance can be achieved.

The problem with this algorithm is that, as the number of blocks/threads is increased the total amount of computation also increases. Therefore, any speedup is based on an increasingly inefficient underlying algorithm. At any time, the local queues in most blocks contain paths that are too long and are irrelevant to the actual shortest paths. The processors are kept busy performing computation that is later discarded. At any given time, it is only productive to work on an irregular and unpredictable subset of the graph. However, irregular and adaptive blocking schemes do not solve the problem, since there is usually equal work available in all blocks. The issue is the distinction between productive and unproductive work.

Our solution is to statically partition the airspace into a large number of blocks and to use a much smaller number of threads. A measure of the average path length is maintained with each local queue. At each step, the blocks with local queues containing the shortest paths are assigned to the threads. Therefore, the subset of blocks that are active and the assignment of blocks to threads change dynamically throughout program execution. This algorithm takes advantage of the symmetric multiprocessing model, in which all threads can access the entire memory space with uniform cost. It also takes advantage of the lightweight multithreading model to achieve good load balance, since the workload within each thread at each step is highly variable.

The ability to develop, test, and debug using sequential methods was crucial in the development of this sophisticated multithreaded algorithm. The entire program was tested and debugged in sequential mode before multithreaded execution was attempted. In particular, development of the complex boundary exchange and queue update algorithms would have been considerably more difficult in multithreaded mode.

The ability to analyze and tune performance using sequential methods was also very important. Good performance depended on exposing enough parallelism without significantly increasing the total amount of computation. We determined efficient values for the number of blocks, the number of threads, and the number of iterations between boundary exchanges by measuring computation times and operation counts of the multithreaded program in running in sequential mode. This detailed analysis would have been very difficult to perform in multithreaded mode. We avoided memory contention in multithreaded mode by avoiding cache misses in sequential mode. The analysis of memory access patterns in sequential mode is much simpler than in multithreaded mode.

4.5. Performance

Other researchers have developed and reported results for message-passing solutions to the Aircraft Route Optimization Problem running on Cray T3D and SGI Power Challenge supercomputers with 16 and 64 processors. We developed our program using Sthreads on a quad-processor 200 MHz Pentium Pro system running Windows NT Server 4.0. (One Pentium Pro processor is approximately the same speed as one SGI Power Challenge processor and twice the speed of one Cray T3D processor.) As shown in Figure 2, our results are better than the results reported on the expensive supercomputers. The reason is the combination of the shared-memory model supported by the Pentium Pro, the lightweight multithreading model supported by Windows NT, and the structured multithreaded pro-

gramming system with sequential development methods supported by Sthreads.

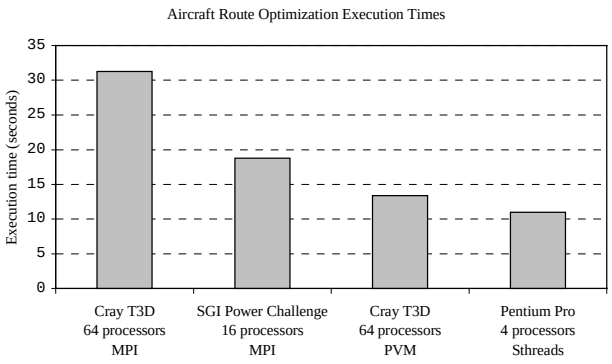

Figure 2: Sthreads on a quad-processor Pentium Pro outperforms message-passing on supercomputers for the Aircraft Route Optimization Problem.

These results should not be misinterpreted as meaning that commodity multiprocessors are now as powerful as supercomputers, or that structured multithreading can obtain super-linear speedups from a small number of processors. For the Aircraft Route Optimization Problem, we obtain approximately three-fold speedup over sequential execution on four processors. However, unlike the message-passing approach, the speedups are obtained from a multithreaded algorithm that adds no significant overheads to the sequential algorithm.

This example is intended as an interesting case study that highlights the strengths of shared-memory, lightweight threads, and structured multithreaded programming. Clearly, there are many other applications for which message-passing supercomputers would outperform a quad-processor Pentium Pro system. Nonetheless, it is significant that inexpensive commodity multiprocessors are now a practical consideration in many areas that were previously the sole domain of supercomputers.

5. Additional Applications under Development

We are in the process of developing a number of other multithreaded applications to demonstrate the power of structured multithreaded programming on commodity multiprocessors running Windows NT. These applications include the following:

- **Volume Rendering:** Volume rendering computes an image of a three-dimensional data volume. The image may or may not be realistic, depending on the nature of the data. We obtain better than 3.5-fold speedups on quad-processor Pentium Pro systems.
- **Terrain Masking:** Terrain masking computes the altitudes at which an aircraft will be visible to ground-based radar systems in an uneven terrain. This problem has obvious applications in military and civil aviation. Terrain masking is part of the C3I Parallel Benchmark Suite. We obtain 3-fold speedups on quad-processor Pentium Pro systems.
- **Threat Analysis:** Threat analysis is a military application that performs a time-stepped simulation of the trajectories of incoming threats and analyses options for intercepting the threats. Threat analysis is part of the C3I Parallel Benchmark Suite. We obtain almost 4-fold speedups on quad-processor Pentium Pro systems.
- **Protein Folding:** Protein folding takes a known protein chain and computes the most likely three-dimensional structures for the molecule. This problem is of vital interest to biochemists involved in drug design. We obtain 3-fold speedups on quad-processor Pentium Pro systems.
- **Molecular Dynamics Simulation:** We are developing a multithreaded implementation of an existing molecular dynamics simulation program (MPSim) that uses the cell multipole method. MPSim consists of more than 100 source files and over 100,000 lines of code. We obtain better than 3.5-fold speedups on quad-processor Pentium Pro systems for simulations involving up to half a million atoms.

All of these applications are extremely computationally intensive. Depending on the input data, the problems may require many hours or days to solve on fast single-processor machines. For this reason, much of the previous work on these problems has been with expensive supercomputers. The computational challenge associated with these problems is not about to disappear with the next generation of fast processors.

6. Comparison with Related Work

Over the years, hundreds of different systems for multithreaded programming have been proposed and implemented. These systems include thread libraries, concurrent languages, sequential languages with pragmas, data parallel languages, automatic parallelizing compilers, and parallel dataflow languages. We have attempted to combine the best attributes of these other approaches into one simple, timely, and highly accessible multithreaded programming system.

The contribution of our system is not in the individual constructs, but in their combination and context. Multithreaded block and `for` loop constructs date back

to the 1960s and have been incorporated in many forms in concurrent languages, pragma-based systems, and data parallel languages. Synchronization flags and counters are derived from concepts originally explored in the context of parallel dataflow languages. Locks and barriers are standard synchronization constructs in many thread libraries and concurrent languages.

An important difference between our system and others is the combination of the following: (i) the emphasis on practical development of sophisticated multithreaded programs using sequential methods, (ii) the ability to ensure deterministic execution, but express nondeterminacy where necessary, and (iii) the minimalist integration of these ideas with popular languages, operating systems, and tools. We believe that this is the right approach to making multithreading practical for a wide range of applications running on modern multiprocessor platforms.

7. Conclusion

We are developing a system for structured high-performance multithreaded programming on top of the support that Windows NT provides for lightweight multithreading. Our system consists of a structured thread library and a preprocessor integrated with Microsoft Developer Studio Visual C++. An important attribute of our system is the ability to develop multithreaded programs using traditional sequential methods and tools. We have developed multithreaded applications in a wide range of areas such as optimization, graphics, simulation, and applied science. The performance of these applications on multiprocessor Pentium Pro systems has been excellent. Our experience developing these applications has convinced us that high-performance multithreading is ready to enter the mainstream of computing.

Availability

Information on obtaining the multithreaded programming system described in this paper can be found at `http://threads.cs.caltech.edu/` or can be requested from `threads@cs.caltech.edu`.

Acknowledgments

The applications described in this paper are being developed by the following Caltech students: Eric Bogs, Marrq Ellenbecker, Yuanshan Guo, Maria Hui, Lei Jin, Autumn Looijen, Scott Mandelsohn, Bradley Marker, Jason Roth, Sean Suchter, and Louis Thomas. We are developing the applications in collaboration with the following Caltech faculty, staff, and students: Prof. Peter Schröder in the Computer Science Department, Dr. Jerry Solomon and David Liney in the Computational Biology Department, and Prof. William Goddard III, Dr. Tahir Cagin, and Hao Li in the Materials and Process Simulation Center.

Thanks to Intel Corporation for donating multiprocessor Pentium Pro computer systems. Thanks also to Microsoft Corporation for donating software and documentation. Special thanks to David Ladd of Microsoft University Research Programs for being so responsive to our requests. The design of our multithreaded programming system greatly benefited from discussions with Tim Mattson of Intel Corporation. This research was funded in part by the NSF Center for Research on Parallel Computation under Cooperative Agreement No. CCR-9120008 and by the Army Research Laboratory under Contract No. DAHC94-96-C-0010.

References

[1] Jim Beveridge and Robert Wiener. *Multithreading Applications in Win 32.* Addison-Wesley, Reading, Massachusetts, 1997.

[2] Helen Custer. *Inside Windows NT.* Microsoft Press, Redmond, Washington, 1993.

[3] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[4] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming with Threads.* Prentice Hall, Upper Saddle River, New Jersey, 1996.

[5] Richard C. Metzger, Brian VanVoorst, Luiz S. Pires, Rakesh Jha, Wing Au, Minesh Amin, David A. Castanon, and Vipin Kumar. *The C3I Parallel Benchmark Suite - Introduction and Preliminary Results.* Supercomputing '96, Pittsburgh, Pennsylvania, November 17-22 1996.

[6] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrel. *Pthreads Programming.* O'Reilly, Sebastopol, California, 1996.

[7] David J. Kruglinski. *Inside Visual C++.* Microsoft Press, Redmond, Washington, fourth edition, 1997.

[8] Thuan Q. Pham and Pankaj K. Garg. *Multithreaded Programming with Windows NT.* Prentice Hall, Upper Saddle River, New Jersey, 1996.