Semi-structured Portable Library for Multiprocessor Servers

Georgios Tsilikas and Martin Fleury
University of Essex
Electronic Systems Engineering Dept.
Multimedia Architectures Laboratory
Colchester, Essex, CO4 3SQ, UK
{gtsili,fleum}@essex.ac.uk

Abstract

The MiPPS library supports a hybrid model of parallel programming. The library is targeted at commodity multiprocessors, with support for clusters. The implementation of the concurrency routines reveals discrepancies between popular operating systems. Tests on suitable applications also reveal similar discrepancies in performance across different multiprocessors. The MiPPS library has also been the basis of a parallelization of the Active Chart Parsing algorithm for speech understanding.

1 Introduction

We are developing a portable software library in the C++ programming language, which supports a hybrid programming paradigm, combining shared- and distributed-memory models. Structure is built into the library by imposing phased transitions between the two paradigms. As a result, the application programmer is restricted to the set of library calls appropriate to the phase the application program is in. Given that the library is intended for large-scale, complex applications, the library is built upon object-oriented principles. The MiPPS project[1] library is light-weight and experimental when compared to other libraries, such as ARCH [1], which is dual paradigm, or ACE [14], which is a set of object-oriented wrappers for distributed processing, or indeed by combining shared-memory OpenMP [3] with MPI (Message Passing Interface) [6].

There now exist low-cost multiprocessor servers capable of coping with the greater throughput and latency requirements in electronic commerce, but suitable as multimedia servers. For example, SGI have offered a four-processor Xeon Pentium-III system running Windows NT, the 540 Visual Workstation, and Dell provide a similar machine, the PowerEdge 6400, running Red Hat Linux. Both operating systems have support for scheduling of kernel-level threads, along with primitives for synchronization of shared-memory access. In the first instance, the library is targeted at this commodity class of machine, with the possibility of combining with Beowulf [17] or Mosix clusters for message-passing phases.

Though the Dell machine is intended as a web throughput engine, it can be adapted for other purposes such as a speech interface server. In this scenario, database enquiries in ‘natural speech’ are pre-processed at (say) a mobile ‘phone to form acoustic frames. From these frames, a speech recognition algorithm, such as one based on Hidden Markov Models (HMM’s), extract words and sentences. It has already been established [5] that speech recognition involves both computationally intensive processing to form the Gaussian mixture weightings at each node of the HMM, *and* synchronized access to a complex data structure to update the sentence model as more frames arrive. Subsequently, identified text has to be interpreted to extract its meaning. In this paper, we consider Active Chart parsing, a speech-understanding algorithm, requiring fine-grained access to a shared data structure. Therefore, ideally the two predominant styles of parallel programming are *both* required, message passing between processes running in partitioned memory, and shared-memory access by threads.

Speech processing is by no means isolated in this requirement, for example radar processing involves a preprocessing phase when the pulse Doppler returns are windowed and Fourier transformed, with clutter removal (suitable for message passing); followed by the tracking of moving targets over time, requiring access to a global data structure, (and hence suitable for shared-memory access).

Radar and speech processing are large-scale software systems, with lines of code exceeding 10 k lines of code, and multiple, diverse algorithms. It is possible to embed (say) PVM (Parallel Virtual Machine) or MPI calls into

[1] Mixed Parallel Paradigm Servers (MiPPS) for Multimedia EPSRC Contract: GR/M89225

C++, but this does not enable isolation from the underlying software drivers, transparent modification of those drivers according to the operating system, and ultimately the ability to embed performance and debug monitoring within the class. An alternative higher-level approach [8] is to embed archetypal parallel structures within software components. Increased expressivity results from restricting structuring to the language level.

Given the reality that the underlying hardware cannot normally support both memory models at once (though see [7]), we do seek to impose phase-based structuring to avoid simultaneous use of both types of parallel primitive, and allow portability. Phase-based structuring is imposed by a software barrier between separate phases of computation, in the manner of BSP (Bulk Synchronous Processing) supersteps [12] , though without a linear performance model. An important proviso is that at software development time both models are supported through emulation of distributed-memory processing by memory partitioning on a shared-memory machine.

In this paper, the construction of the shared-memory class library is considered, discussing portability issues, and validating correct performance on the two machines and operating systems. The paper concludes with a case study on Active Charts.

2. System Overview

Threads are assigned from a thread pool to various processors, depending on the underlying architecture, in a similar manner to the Filaments library [11], which also tries to unify the two forms of memory model. In Fig. 2, tasks are assigned to threads, and thence to processors, which form a virtual parallel machine. An application will generate tasks and place them in the task pool, where the tasks can be sorted based on priority or resource requirements, and also may have priority lanes, a feature of real-time CORBA, [15]. When any thread finishes what it was doing it then it can request a new task from the task pool, with limited start-up overhead.

3. Class library

The class library is built upon the POSIX (IEEE Portable Operating System Interface) Threads Library 1003.1c (applicable to most Unix and Unix-like operating systems (OS) and VMS), and the Win32 Threads API (for WinNT 4.0 and above). Unfortunately, the Windows NT POSIX subsystem lacks operability [10] with other software. Dual development was under Red Hat Linux 7.2, Linux Pthreads and GNU C++ compiler version 2.96, together with WinNT 4.0 and Visual Studio 6.0 VC++. A high-level view of the library class structure is shown in the UML (Unified Modeling Language) diagram of Fig. 2.

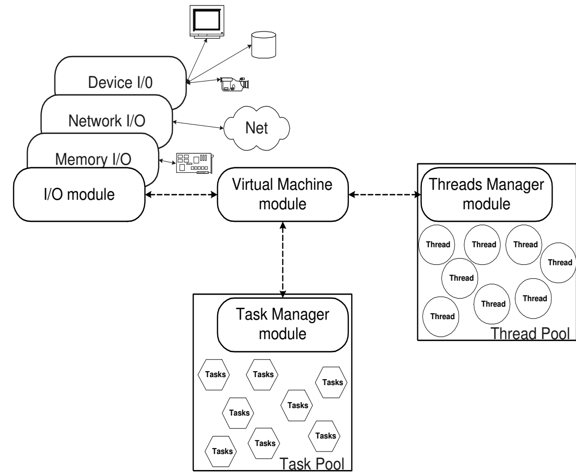


Figure 1. MiPPS system overview

The Thread class can be used to create and manage a thread. The actual thread is not created when the object is initiated but when one of the two `run()` functions is called. The `run()` function is overloaded to either accept a pointer to a function or a task object. In either case, a static private function of the thread object is called that executes the function or the `run()` member function of the task object. The use of a *static* function overcomes a generic C++ problem: member function pointers are not generally passed correctly. The `wait()` member function can be called to wait for the termination of the thread. At present, only one task or function can be assigned to the thread. However, the intention is to extend the functionality of the thread class to accept more that one function or task or even to be able to request a task from a task pool.

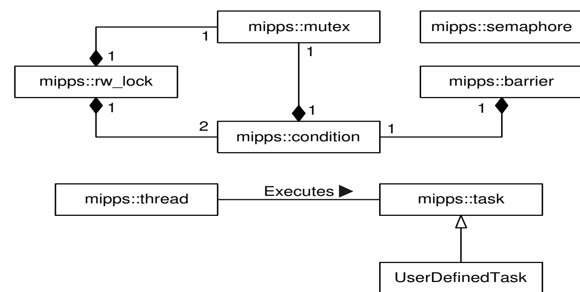


Figure 2. MiPPS class library

The task class contains only a purely virtual function called `run()`. The programmer can then inherit from the task class and create a task by defining the run member function. The thread class uses the `beginthreadex()` function when the OS is Windows or

the `pthread_create()` function when the OS is Linux.

Further weaknesses of any C++ thread handling that we seek to address are: the possibility of premature, external destruction; lack of selective method acceptance depending on state; and the inability to provide a centralized exception handler, as exceptions do not normally cross thread boundaries.

The Mutex class can be used to protect a resource that requires mutual exclusion. Under Windows the class creates a critical section, while under Linux it creates a 'fast' mutex (one with limited safety, *e.g.* without ownership checks). Other kinds of mutexes exist in both architectures. However, the ones selected are the most primitive and, therefore, have the fastest implementations. One difference between the Windows and Linux variation is that the Linux one will deadlock if the owner of the lock tries to lock the mutex again (*i.e.* recursive locking) while the Windows' one will not deadlock, but it will require the lock to be unlocked twice. The intention is to extend the mutex class to include other kinds off locks such as spin-locks, recursive locks, priority or in general ordered locks, and finally locks that will prevent deadlocks or starvation, and that function in the same manner under each OS. Some OS's already provide some of the mutexes mentioned earlier in native form, so libraries can extend the native support to provide a uniform mutex class.

Counting semaphores are variables that can be incremented arbitrarily high, but decremented only to zero. The semantics for both the Linux and Windows variation are the same as both OS provide natively the same functionality.

Condition variables exist in native form on Linux systems. Condition variables under Windows are emulated in the library using events though their implementation can be challenging and error prone. For example some implementation methods suffer from starvation, unfairness, or race conditions [16]. The current implementation is based on the `SignalObjectAndWait()` function that is not available before Windows NT. Therefore, the class cannot be used under Windows 9X.

R/W (Read/Write) locks can be used to allow simultaneous access to many readers but only one writer. When no writers are present, readers can access the resource immediately; the only overhead is the mutual increment of an integer that counts the number of readers. Writers have priority over the resource in the sense that when a writer tries to access the resource it will wait for the existing readers to finish but new readers will queue and be allowed to access the resource only when all the writers have finished. This lock can be used on shared data structures that get read often but written to only seldom. Neither POSIX nor Win32 libraries define R/W locks except from some specific non-portable extensions. R/W locks are implemented in the library using two condition variables and a mutex that is shared between the condition variables.

Barriers are not one of the low-level synchronization constructs of either POSIX or Win32 thread API. In the POSIX extension, it is proposed that when a thread enters the barrier it will automatically sleep and wait for the last thread to arrive. When the last thread enters the barrier all threads are released and continue execution. The current barrier class is implemented using the conditional variable class, and hence provides a local barrier. BSP has already investigated various forms of efficient global barrier synchronization [9].

4. Results

Two preliminary tests of relative performance between the OS were made, which also served to verify correct behavior of the library calls. In the first 'toy' example, matrix multiplication was intended to test the memory hierarchy, while, in the second, multi-thread access to linked lists was intended to examine synchronization performance.

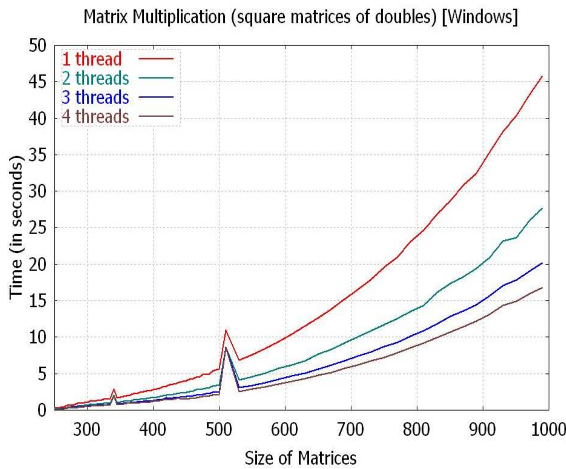
4.1. Matrix multiplication parallelization

The surprise result, considering the relative clock speed of the SGI and Dell machines, respectively 450 MHz with 512 MB L1 cache and 750 MHz with 1 GB L1 cache, was the superior performance of the SGI machine, Fig. 3, despite a careful configuration of Linux. One reason for this behavior might be attributable to the specialist I/O bus on the SGI machine, which is intended for graphics. Another reason (apparently) might be a relatively poor Linux implementation of threading, as it was observed that a single-threaded version of matrix multiplication was slower than a version without threading, while the same effect did not occur on the Windows machine. A naïve, row-column multiplication algorithm, $O(n^3)$ operations, was found to be slower on the Linux machine but faster on the Windows machine. (Winograd's algorithm [4], $O(n^{\log 7})$, was found to be significantly faster under Linux but slower under Windows than the naïve method.) Another unexpected result was that no significant difference in performance was found as the number of threads was increased beyond the number of processors to 20 threads on both machines.

4.2. Linked-list parallelization

A small application was constructed whereby a linked list of tasks, protected by a mutex, was accessed by threads. Each task sends the thread to sleep for a time determined statistically (constant or deterministic, Gaussian (normal) and Uniform pdf's (probability density function's)). The total theoretical time that the execution should take with no overhead was compared to the actual time using a system

class. The theoretical time was subtracted from the practical time to find the overhead, caused by either the contention on the mutex or by the scheduler of the OS. The test was run for 1024 tasks ranging from 1 ms up to 20 ms and from 1 thread up to 128 threads. The size of the tasks did not seem to affect significantly the overall delay. Some of the delay is caused because of an unbalanced number of tasks (*e.g.* 5 tasks being completed by 4 threads). It appears as if the single mutex is a 'hot spot' that adds some overhead to the application. However, some of the overhead may very well have been because of the system having to schedule a large number of threads, especially for small tasks. Fig. 4.2 shows the results for a constant task size on the Windows machine.



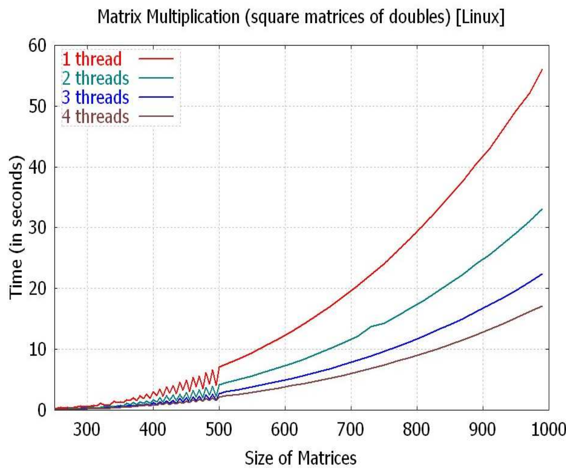


Figure 3. Relative performance between OS for matrix multiplication

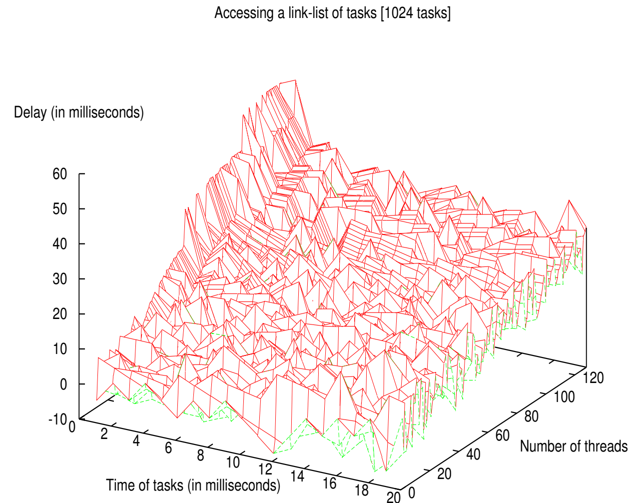


Figure 4. Multithreading overhead

4.3. Active Chart Parsing parallelization

The Cowichan problems [19] are a set of medium-sized, realistic applications suitable for assessing the performance of parallel programming systems. Active Chart Parsing (ACP) is a technique to generate all possible parsings of a sentence, given an ambiguous grammar. Therefore, ACP would be one part of a natural language interface to a database server.

Previously reported results on a Network of Workstations (NOW) [18] show no speedup, a not surprising conclusion as ACP is inherently fine-grained and synchronous. Partial results, tentative or definite identification of phrases, are stored for later use in the serial algorithm, and exchanged in any parallel algorithm. One difficulty in the way of any parallelization is that there is an ordering constraint, as, at least in the English language, aggregated phrases are formed from adjacent words or phrases.

The implementation of [18] was through the Orca language [2], which adopts a shared-object model with implicit message passing. Replication of shared object state is reported to have resulted in the slow-down, as local changes of state required a broadcast. Therefore, the ACP algorithm is appropriate for a physically shared-memory address space, where the MiPPS or similar library synchronizes global-access to shared data structures.

Fig. 4.3 shows the serial version of the ACP algorithm adopted from [13], with the portion that is now distributed on a per-thread basis picked out. Input to the algorithm is a putative sentence (or phrase) to be parsed, and a set of grammatical rules.

In the serial algorithm, each word of the sentence is initially stored in a stack data-structure called an `Agenda`. At this stage, a word is also an `Edge`, or grammatical unit. An `Active Edges` represents a hypothesis that an aggregate edge can be formed by combination with an `Inactive Edge`. Likewise, `Inactive Edges` are matched to `Active Edges`, but there is an added possibility of making a new edge identification by re-applying the grammatical rules. Any edges formed are stored on either an `Active Edge` or `Inactive Edge` array. Edges formed are also pushed onto the `Agenda`, which represents a set of tasks executed in last-in first-out (LIFO) order. Edges taken from the `Agenda` are matched against edges held in either the `Active` or `Inactive Edge` arrays. Termination is when there are no further edges in the `Agenda`.

In the parallel algorithm, the number of threads created is equal to the number of words in a global `Agenda`. However, once a thread is created it then puts the edge formed from its initial word on a local `Agenda` and continues to push any edges identified onto the local `Agenda`. However, it also copies those edges into the appropriate global edge array. Therefore, access to the two global edge arrays is synchronized. In the parallel algorithm, it is important to ensure that if an identified edge is held by a thread that it is placed in a global array to avoid another thread terminating prematurely through 'edge starvation'.

The following C++ code extract shows the main processing loop with MiPPS calls to create a thread, and subsequently to assign a task to that thread. The chart, which is a singleton object, holds the edge arrays. The dynamic cast is required to assign a derived task type to the task base class.

```
thread *tharr[MAX_THREADS];
mytask *tkarr[MAX_THREADS];

while ( !agenda.isEmpty() ){

   // TAKE AN EDGE FROM THE AGENDA.
   Edge edge = agenda.getNext();

   // ADD AN EDGE TO CHART.
   // TRY TO COMBINE EDGE W/
   // OTHERS IN CHART.

   tharr[curth] = new thread;
   tkarr[curth] = new mytask(chart,
                  edge,agenda,rules);

   tharr[curth]->run(dynamic_cast<task &>
                    (*tkarr[curth]));
   curth++;
  }

// TEST CHART FOR SUCCESS(ES)
return chart.success( goal, string );
```

Performance based on partial results is encouraging. Thus, a sentence with fourteen words, and hence fourteen threads, was tested on the four-processor machine running Windows NT. The serial version took 1.6s to complete, whereas the parallel version took 0.58s, a speed-up of 2.76.

5. Conclusion

Design of programming libraries for multimedia applications, considered as a whole, must take into account that these applications are not, like some numerical analysis algorithms, amenable solely to data parallelism. Multimedia applications are also likely to run on commodity processors, which now include shared-memory multiprocessors with limited scalability. The library that has been designed includes both of the major parallel programming paradigms, but, given the reported difficulty of parallel programming, seeks to organize the application programmer by introducing structural phases.

The paper reports implementation considerations when going between the major commodity operating systems. It also raises the difficulty of predicting performance in this environment. A medium-sized parallel application, indicative of algorithms suitable for shared-memory processing, Active Chart Parsing, has been developed. First results indicate an improvement on previous reported results. Future work will add message-passing primitives to the MiPPS library, before a further phase of performance testing.

References

[1] J.-M. Adamo. ARCH, an object oriented MPI-based library for asynchronous and loosely synchronous parallel system programming. In *4th European PVM/MPI Users' Meeting*, pages 67–74, 1997.

[2] H. E. Bal, A. S. Tanenbaum, and M. F. Kaashoek. Orca: A language for distributed programming. *SIGPLAN Notices*, (5), 1990.

[3] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming with OpenMP*. Morgan Kaufmann, San Francisco, CA, 2001.

[4] S. Chatterjee, A. R. Leback, P. K. Patnala, and M. Thottethodi. Recursive array layouts and fast parallel matrix multiplication. In *ACM Symposium on Parallel Algorithms and Architectures*, 1999.

[5] M. Fleury, A. C. Downton, and A. F. Clark. Parallel structure in an integrated speech-recognition network. In *Euro-Par'99*, pages 995–1004, 1999. LNCS 1685.

[6] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2*. MIT, Cambridge, MA, 1999.

[7] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of message passing and shared memory in the Stanford FLASH multiprocessor. In *6th International Confer-*

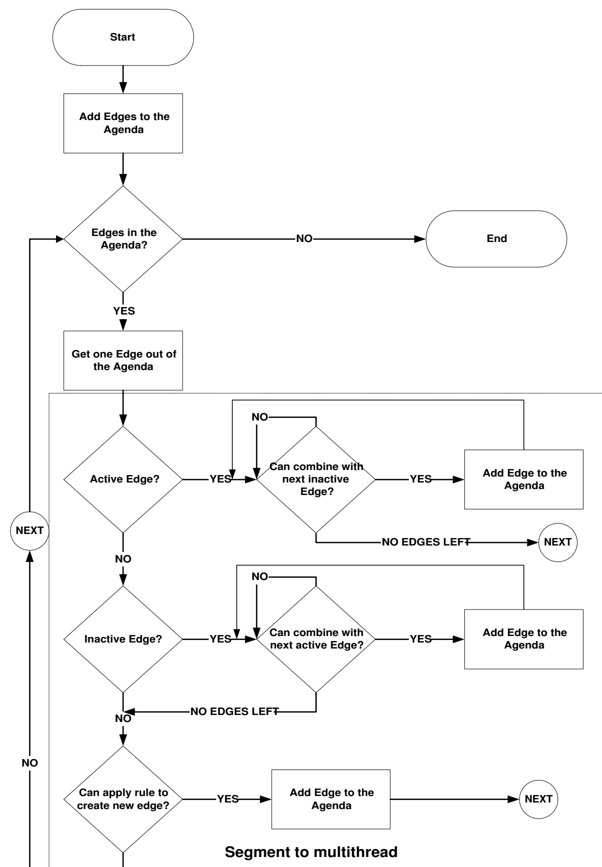


Figure 5. Flowchart of Chart Parsing Algorithm indicating the basis of multithreading

ence on Architectural Support for Programming Languages and Operating Systems, pages 38–50, 1994.

[8] D. J. Johnston, M. Fleury, and A. C. Downton. Multi-paradigm frameworks for parallel image processing. In *IPDPS/PDIVM'03*, 2003. In this volume.

[9] J.-S. Kim, S. Ha, and C. S. Jhon. Efficient barrier synchronization mechanism for the BSP model on message-passing architectures. In *IPDPS/SPDS*, 1998.

[10] D. G. Korn. Porting UNIX to Windows NT. In *USENIX Technical Conference*, pages 43–57, 1997.

[11] D. K. Lowenthal and V. C. Freeh. Architecture-independent parallelism for both shared- and distributed-memory machines using the Filaments package. *Parallel Computing*, 26:1297–1323, 2000.

[12] W. G. McColl. Scalable computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pages 46–51, 1995. LNCS 1000.

[13] D. Perelman-Hall. *Byte*, pages 237–241, February 1992.

[14] D. C. Schmidt. The ADAPTIVE communication environment: An object-oriented network programming toolkit for developing communication software. *Concurrency: Practice and Experience*, 5(4):269–286, 1993.

[15] D. C. Schmidt and F. Kuhns. An overview of real-time CORBA. *IEEE Computer*, 33(6):56–63, 2000.

[16] D. C. Schmidt and I. Pyarali. Strategies for implementing POSIX condition variables on Win32. *C++ Report SGS*, 10(5), 1998.

[17] T. Sterling, editor. *Beowlf Cluster Computing with Linux*. MIT, Cambridge, MA, 2002.

[18] A. R. Sukul. Parallel implementation of an Active Chart parser in Orca. Technical report, Vrije Universiteit, Amsterdam, 1996.

[19] G. V. Wilson and H. E. Bal. An empirical assessment of the usability of Orca using the Cowichan problems. *IEEE Parallel and Distributed Technology*, 4(3):36–44, 1996.