Communicating Process Architectures 2007
Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch
IOS Press, 2007

C++CSP2: A Many-to-Many Threading Model for Multicore Architectures

Neil BROWN

Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK.
neil@twistedsquare.com/nccb2@kent.ac.uk

Abstract.
The advent of mass-market multicore processors provides exciting new opportunities for parallelism on the desktop. The original C++CSP – a library providing concurrency in C++ – used only user-threads, which would have prevented it taking advantage of this parallelism. This paper details the development of C++CSP2, which has been built around a many-to-many threading model that mixes user-threads and kernel-threads, providing maximum flexibility in taking advantage of multicore and multi-processor machines. New and existing algorithms are described for dealing with the run-queue and implementing channels, barriers and mutexes. The latter two are benchmarked to motivate the choice of algorithm. Most of these algorithms are based on the use of atomic instructions, to gain maximal speed and efficiency. Other issues related to the new design and related to implementing concurrency in a language like C++ that has no direct support for it, are also described. The C++CSP2 library will be publicly released under the LGPL before CPA 2007.

Keywords. C++CSP, C++, Threading, Atomic Instructions, Multicore

Introduction

The most significant recent trend in mass-market processor sales has been the growth of multicore processors. Intel expected that the majority of processors they sell this year will be multicore [1]. Concurrent programming solutions are required to take advantage of this parallel processing capability. There exist various languages that can easily express concurrency (such as occam-π [2]), but the programming mainstream is slow to change languages; even more so to change between programming 'paradigms'. Sequential, imperative, procedural/object-oriented languages remain dominant. It is for this reason that libraries such as JCSP [3], C++CSP [4,5], CTC++ [6] and CSP for the .NET languages [7,8] have been developed; to offer the ideas of occam-π and its ilk to programmers who choose (or are constrained) to use C++, Java and the various .NET languages.

C++CSP has previously been primarily based on user-threads, which simplify the algorithms for the primitives (such as channels and barriers) as well as being faster than kernel-threads, but cannot take advantage of multicore processors. A review of the latter problem has given rise to the development of C++CSP2.

This report details the design decisions and implementation of C++CSP2 [9]. Section 1 explains the new threading model that has been chosen. Section 2 briefly clarifies the terminology used in the remainder of the paper. Section 3 provides more details on the implementation of the run-queue and timeout-queue. Section 4 describes how to run processes in a variety of configurations in C++CSP2's threading model. Section 5 details a new barrier algorithm specifically tailored for the chosen threading model. Section 6 then presents benchmarks and discussion on various mutexes that could be used as the underpinning for many of

C++CSP2's algorithms. Section 7 contains details on the modified channel-ends and section 8 discusses the channel algorithms. Section 9 highlights some issues concerning the addition of concurrency to C++. Finally, section 10 provides a brief discussion of networking support in C++CSP2, and section 11 concludes the paper.

Notes

All the benchmarks in this report are run on the same machine; an Intel Core 2 Duo 6300 (1.9 Ghz) processor with 2GB DDR2 RAM. The 'Windows' benchmark was run under Windows XP 64-bit edition, the 'Linux' benchmark was run under Ubuntu GNU/Linux 64-bit edition.

C++CSP2 is currently targeted at x86 and (x86-64) compatible processors on Windows (2000 SP4 and newer) and Linux (referring to the Linux kernel with GNU's glibc).

1. Threading

1.1. Background

Process-oriented programming, based on CSP (Communicating Sequential Processes) [10] principles, aims to make concurrency easy for developers. In order to provide this concurrency, developers of CSP implementations (such as JCSP [11], CTJ [12], KRoC [13]) must use (or implement) a threading mechanism. When running on top of an Operating System (OS), such threading mechanisms fall into three main categories: user-threads, kernel-threads and hybrid threading models.

User-threads (also known as user-space or user-level threads, or M:1 threading) are implemented in user-space and are invisible to the OS kernel. They are co-operatively scheduled and provide fast context switches. Intelligent scheduling can allow them to be more efficient than kernel-threads by reducing unnecessary context switches and unnecessary spinning. However, they do not usually support preemption, and one blocking call blocks all the user-threads contained in a kernel-thread. Therefore blocking calls have to be avoided, or run in a separate kernel-thread. Only one user-thread in a kernel-thread can be running at any time, even on multi-processor/multicore systems. C++CSP v1.3 and KRoC use user-threads.

Kernel-threads (also known as kernel-space or kernel-level threads, or 1:1 threading) are implemented in the OS kernel. They usually rely on preemption to perform scheduling. Due to the crossing of the user-space/kernel-space divide and other overheads, a kernel-thread context switch is slower than a user-space context switch. However, blocking calls do not cause any problems like they do with user-threads, and different kernel-threads can run on different processors simultaneously. JCSP (on Sun's Java Virtual Machine on most Operating Systems) uses kernel-threads.

Hybrid models (also known as many-to-many threading or M:N threading) mix kernel-threads and user-threads. For example, SunOS contained (adapting their terminology to that used here) multiple kernel-threads, each possibly containing multiple user-threads, which would dynamically choose a process to run from a pool, and run that process until it could no longer be run. Much research on hybrid models has involved the user-thread and kernel-thread schedulers sharing information [14,15].

In recent years hybrid models have faded from use and research. The predominant approach became to increase the speed of kernel-threading (thereby removing its primary drawback) rather than introduce complexity with hybrid models and/or ways to circumvent user-

threading's limitations. This was most obvious in the development of the NGPT (Next Generation POSIX Threads) library for Linux alongside the NPTL (Native POSIX Thread Library). NGPT was a complex hybrid threading library, whereas NPTL was primarily centred on a speed-up of kernel-threading. NPTL is now the default threading library for Linux, while development of NGPT has been quietly abandoned.

1.2. C++CSP2 Threading Options

This section explores the three different threading models that could be used for C++CSP2.

1.2.1. User-Threads

The previous version of C++CSP used only user-threads. With the mass-market arrival of multicore processors, the ability to only run on one processor simultaneously became an obvious limitation that needed to be removed. Therefore, continuing to use only user-threads is not an option.

1.2.2. Kernel-Threads

The most obvious move would have been to change to using only kernel-threads. In future, it is likely that the prevalence of multicore processors will continue to motivate OS developers to improve the speed of kernel-threads, so C++CSP would get faster without the need for any further development.

Channels and barriers between kernel-threads could be implemented based on native OS facilities, such as OS mutexes. The alternative would be to use atomic instructions – but when a process needed to wait, it would either have to spin (repeatedly poll – which is usually very wasteful in a multi-threaded system) or block, which would involve using an OS facility anyway.

1.2.3. Hybrid Model

The other option would be to combine kernel-threads and user-threads. The proposed model would be to have a C++CSP-kernel in each kernel-thread. This C++CSP-kernel would maintain the run queue and timeout queue (just as the C++CSP kernel always has). Each process would use exactly one user-thread, and each user-thread would always live in the same kernel-thread.

It is possible on Windows (and should be on Linux, with some assembly register hacking) to allow user-threads to move between kernel-threads. However, in C++ this could cause confusion. Consider a process as follows:

```
void run()
{
    //section A
    out << data;
    //section B
}
```

With moving user-threads, the code could be in a different thread in section A to section B, without the change being obvious. If this was a language like occam-π, where the code in these sections would also be occam-π code, this may not matter. But C++CSP applications will almost certainly be interacting with other C++ libraries and functions that may not handle concurrency as well as occam-π. These libraries may use thread-local storage or otherwise depend on which thread they are used from. In these cases it becomes important to the programmer to always use the library from the same thread. So allowing user-threads to move would cause confusion, while not solving any particular problem.

Moving user-threads would allow fewer kernel-threads to be created (by having a small pool of kernel-threads to run many user-threads) but the overheads of threading are primarily based on the memory for stacks, so there would not be any saving in terms of memory.

Another reason that hybrid models are considered inferior is that using priority can be difficult. A high priority kernel-thread running a low-priority user-thread would run in preference to a low-priority kernel-thread running a high-priority user-thread. However, C++CSP has never had priority so this is not yet a concern. I believe that the benefits of a hybrid threading model outweigh the drawback of making it difficult to add priority to the library in the future.

1.2.4. Benchmarks

Benchmarking threading models directly is difficult. While context-switches between user-threads can be measured explicitly, this is not so with kernel-threads; we cannot assume that our two switching threads are the only threads running in the system, so there may be any number of other threads scheduled in and out between the two threads we are testing. Therefore the best test is to test a simple producer-consumer program (one writer communicating to one reader), which involves context-switching, rather than trying to test the context switches explicitly.

Four benchmark tests were performed. One test used user-threads on their own (without any kernel-threads or mutexes) and another test tested kernel-threads on their own (without any user-threads or C++CSP2 kernel-interactions). These 'plain' benchmarks reflect a choice of a pure user-threads or pure kernel-threads model. The two hybrid benchmarks show the result of using user-threads or kernel-threads in a hybrid framework (that would allow the use of either or both).

Threading	Windows Time	Linux Time
Plain user-threads	0.19	0.19
Hybrid user-threads	0.27	0.28
Plain kernel-threads	2.7	2.4
Hybrid kernel-threads	3.1	2.8

All figures in microseconds, to 2 s.f.
Each time is per single communication
(including associated context-switches).

1.2.5. Analysis

It is apparent that kernel-threads are at least ten times slower than user-threads. The nature of the benchmark means that most of the time, only one kernel-thread will really be able to run at once. However, these times are taken on a dual-core processor which means that there may be times when less context-switching is needed because the threads can stay on different processors. For comparison, the ratio between user-threads and kernel-threads on single core Linux and Windows machines were both also almost exactly a factor of ten.

For simulation tasks and other high-performance uses of C++CSP2, the speed-up (of using user-threads rather than kernel-threads) would be a worthwhile gain. However, running all processes solely as user-threads on a multi-processor/core system would waste all but one of the available CPUs.

A hybrid approach would allow fast user-threads to be used for tightly-coupled processes, with looser couplings across thread boundaries. Using a hybrid model with kernel-threads would seem to be around 15% slower than 'plain' kernel-threads. If only kernel-threads were used by a user of the C++CSP2 library, the hybrid model would be this much slower than if C++CSP2 had been implemented using only kernel-threads. In fact, this deficit

is because no additional user-threads are used; much of the time in the hybrid model benchmark is spent blocking on an OS primitive, because the user-thread run-queue is empty (as explained in section 3.1). This cost would be reduced (or potentially eliminated) if the kernel-thread contained multiple user-threads, because the run-queue would be empty less often or perhaps never.

A hybrid approach flexibly offers the benefits of user-threads and of kernel-threads. It can be used (albeit with slightly reduced speed) as a pure user-thread system (only using one kernel-thread), or a pure kernel-thread system (where each kernel-thread contains a single user-thread, again with slightly reduced speed), as well as a combination of the two. Therefore I believe that the hybrid-threading model is the best choice for C++CSP2. The remainder of this paper details the design and implementation of algorithms for the hybrid-threading model of C++CSP2.

2. Terminology

Hereafter, the unqualified term 'thread' should be taken to mean kernel-thread. An unqualified use of 'kernel' should be taken to mean a C++CSP2-kernel. An unqualified 'process' is a C++CSP process. A 'reschedule' is a (C++CSP2-)kernel call that makes the kernel schedule a new process, without adding the process to the end of the run-queue; i.e. it is a blocking context-switch, rather than a yield (unless explicitly stated). An alting process refers to a process that is currently choosing between many guards in an Alternative. The verb 'alt' is used as a short-hand for making this choice; a process is said to alt over multiple guards when using an Alternative.

3. Process and Kernel Implementation

3.1. Run Queues

Each thread in C++CSP2 has a kernel, and each kernel has a run-queue of user-threads (each process is represented by exactly one user-thread). Any thread may add to any thread-kernel's run-queue. The run-queue is built around a monitor concept; reads and writes to the run-queue are protected by the mutual exclusion of the monitor. When a kernel has no extra processes to run, it waits on a condition variable associated with the monitor. Correspondingly, when a thread adds to another kernel's empty run-queue (to free a process that resides in a different thread), it signals the condition variable.

In Brinch Hansen [16] and Hoare's [17] original monitor concepts, signalling a condition variable in a monitor effectively passed the ownership of the mutex directly to the signalled process. This had the advantage of preventing any third process obtaining the monitor's mutex in between. In the case of the run-queue, using this style of monitor would be a bad idea. A kernel, X, adding to the run-queue of a kernel, K, would signal the condition variable and then return, without releasing the mutex (which is effectively granted to K). Any other threads that are scheduled after X but before K would not be able to add to K's run-queue because the mutex would still be locked. These processes would have to spin or yield, which would be very inefficient and unnecessary.

Instead, the monitor style used is that described by Lampson and Redell [18]. In their system, the signalling kernel (X) does not grant the mutex directly to the signalled kernel K. Instead, it merely releases the mutex. K will be scheduled to run at some future time, at which point it will contend for the mutex as would any other kernel. This allows other kernels to add to K's run-queue before K has been scheduled – but only X (which changes the run-queue from empty to non-empty) will signal the condition variable.

There is one further modification from Lampson and Redell's model. They replace Hoare's "IF NOT (OK to proceed) THEN WAIT C" with "WHILE NOT (OK to proceed) DO WAIT C"; because the monitor can be obtained by another process in between the signaller and the signallee, the other process could have changed the condition ("OK to proceed") to false again. In the case of C++CSP2, kernels may only remove processes from their *own* run-queue. Therefore the condition (the run-queue being non-empty) can never be invalidated by another kernel (because they cannot remove processes from the queue).

Hoare has shown that monitors can be implemented using semaphores [17] (in this case only two would be needed – one to act as a mutex, one for the condition variable). Therefore one implementation option would be to use an OS semaphore/mutex with an OS semaphore. In section 6 we will see that our own mutexes are considerably faster than OS mutexes, therefore a faster implementation is to use our own mutex with an OS semaphore. This is how it is implemented on Linux – on Windows, there are 'events' that are more naturally suited to the purpose fulfilled by the semaphore on the Linux.

There were other implementation options that were discounted. The semaphore could have been implemented using atomic instructions in the same way most of the mutexes in section 6 are. This would inevitably have involved spinning and yielding. The process will be blocked for an indefinite amount of time, which makes spinning and yielding inefficient. The advantage of an OS semaphore/event is that it blocks rather than spinning, which will usually be more efficient for our purposes. The other discarded option is that the POSIX threads standard supports monitors directly (in the form of a combination of a mutex and condition variable). Benchmarks revealed this option to be at least twice as slow as the mutex/semaphore combination that C++CSP2 actually uses. The newly-released Windows Vista also provides such support for monitors, but I have not yet been able to benchmark this.

3.2. Timeouts

Similar to previous versions of C++CSP, a timeout queue is maintained by the kernel. It is actually stored as two queues (in ascending order of timeout expiry) – one for non-alting processes and one for alting processes. Before the kernel tries to take the next process from the run queue (which may involve waiting, as described in the previous section), it first checks the timeouts to see if any have expired. If any non-alting timeouts have expired, the processes are unconditionally added back to the run queue. If any alting timeouts have expired, an attempt is made to add the processes back to the run-queue using the `freeAltingProcess` algorithm described in the next section.

The previous section described how a kernel with an empty run-queue will wait on a condition variable. If there any timeouts (alting or non-alting) that have not expired, the wait is given a timeout equal to the earliest expiring timeout. If no timeouts exist, the wait on the condition variable is indefinite (i.e. no timeout value is supplied).

3.3. Alting

Processes that are alting pose a challenge. Alting is implemented in C++CSP2 in a similar way to JCSP and KRoC. First, the guards are enabled in order of priority (highest priority first). Enabling can be seen as a 'registration of interest' in an event, such as indicating that we may (conditionally) want to communicate on a particular channel. If no guards are ready (none of the events are yet pending) then the process must wait until at least one event is ready to take place. As soon as a ready guard is found, either during the enable sequence or after a wait, all guards that were enabled are disabled in reverse order. Disabling is simply the reverse of enabling – revoking our interest in an event. At the end of this process, the highest priority ready guard is chosen as the result of the alt.

Consider a process that alts over two channels and a timeout. It may be that a process in another thread writes to one of the channels at around the same time that the process's kernel finds that its timeout has expired. If the process is waiting, *exactly one* of these two threads should add the process back to the run queue. If the process is still enabling, the process should not be added back to the run queue.

This problem has already been solved (and proven [19]) in JCSP, so C++CSP2's algorithm is an adaptation of JCSP's algorithm, that changes the monitor-protected state variable into a variable operated on by atomic instructions. The skeleton of JCSP's algorithm is as follows:

```
class Alternative
{
  private int state; //can be inactive, waiting, enabling, ready

  public final int priSelect ()
  {
    state = enabling;
    enableGuards ();
    synchronized (altMonitor) {
      if (state == enabling) {
        state = waiting;
        altMonitor.wait (delay);
        state = ready;
      }
    }
    disableGuards ();
    state = inactive;
    return selected;
  }

  //Any guard that becomes ready calls schedule:
  void schedule () {
    synchronized (altMonitor) {
      switch (state) {
        case enabling:
          state = ready;
          break;
        case waiting:
          state = ready;
          altMonitor.notify ();
          break;
        // case ready: case inactive:
        // break
      }
    }
  }
}
```

C++CSP2's algorithm is as follows. Note that unlike JCSP, it is possible that the `freeAltingProcess` function might be called on a process that is not alting – hence the case for dealing with `ALTING_INACTIVE`.

```
unsigned int csp::Alternative::priSelect()
{
    int selected, i;
    AtomicPut(&(thisProcess->altingState),ALTING_ENABLE);

    //Check all the guards to see if any are ready already:
    for (i = 0;i < guards.size();i++)
    {
        if (guards[i]->enable(thisProcess))
            goto FoundAReadyGuard;
    }
    i -= 1;

    if (ALTING_ENABLE == AtomicCompareAndSwap(&(thisProcess->altingState),
        /*compare:*/ ALTING_ENABLE, /*swap:*/ ALTING_WAITING))
    {
        reschedule(); //wait
    }

FoundAReadyGuard: //A guard N (0 <= N <= i) is now ready:
    for (;i >= 0;i--)
    {
        if (guards[i]->disable(thisProcess))
            selected = i;
    }

    AtomicPut(&(thisProcess->altingState),ALTING_INACTIVE);
    return selected;
}

void freeAltingProcess(Process* proc)
{
    usign32 state = AtomicCompareAndSwap(&(proc->altingState),
        /*compare:*/ ALTING_ENABLE, /*swap:*/ ALTING_READY);

    //if (ALTING_ENABLE == state)
        //They were enabling, we changed the state. No need to wake them.
    //if (ALTING_READY == state)
        //They have already been alerted that one or more guards are ready.
        //No need to wake them.

    if (ALTING_INACTIVE == state)
    {
        freeProcess(proc); //Not alting; free as normal
    }
    else if (ALTING_WAITING == state)
    {
        //They were waiting.  Try to atomically cmp-swap the state to ready.
        if (ALTING_WAITING == AtomicCompareAndSwap(&(proc->altingState),
            /*compare:*/ ALTING_WAITING, /*swap:*/ ALTING_READY))
        {
            freeProcess(proc); //We made the change, so we should wake them.
        }
        //Otherwise, someone else must have changed the state from
        //waiting to ready. Therefore we don't need to wake them.
    }
}
```

Thus, the above algorithm does not involve claiming any mutexes, except the mutexes protecting the process's run-queue – and this mutex is only claimed by a maximum of one process during each alt. This makes the algorithm faster, and avoids many of the problems caused by an 'unlucky' preemption (the preemption of a thread that holds a lock, which will cause other processes to spin while waiting for the lock).

JCSP's algorithm has a "`state = ready`" assignment after its wait, without a corresponding line in C++CSP2. This is because the wait in JCSP may finish because the specified timeout has expired – in which case the assignment would be needed. In C++CSP2 timeouts are handled differently (see section 3.2), so the process is always woken up by a call to `freeAltingProcess`, and therefore the state will always have been changed before the reschedule function returns. With the addition of atomic variables in Java 1.5, it is possible that in future ideas from this new algorithm could be used by JCSP itself.

4. Running Processes

The vast majority of processes are derived from the `CSProcess` class. The choice of where to run them (either in the current kernel-thread or in a new kernel-thread) is made when they are run; the process itself does not need to take any account of this choice. The one exception to this rule is described in section 4.1.

For example, the following code runs each process in a separate kernel-thread[1]:

```
Run( InParallel
    (processA)
    (processB)
    (InSequence
        (processC)
        (processD)
    )
);
```

To run processes C and D in the same kernel-thread, the call `InSequenceOneThread` would be used in place of `InSequence` in the previous code. To instead run processes A and B in one kernel-thread, and C and D in another kernel-thread, the code would look as follows:

```
Run( InParallel
    ( InParallelOneThread
        (processA) (processB)
    )
    ( InSequenceOneThread
        (processC) (processD)
    )
);
```

To run them all in the current kernel-thread:

```
RunInThisThread( InParallelOneThread
    (processA)
    (processB)
    (InSequenceOneThread
        (processC)
        (processD)
    )
);
```

[1]The syntax, which may seem unusual for a C++ program, is inspired by techniques used in the Boost 'Assignment' library [20] and is valid C++ code

In occam-π terminology, we effectively have `PAR` and `SEQ` calls (that run the processes in new kernel-threads) as well as `PAR.ONE.THREAD` and `SEQ.ONE.THREAD` calls. Notice that the shorter, more obvious method (`InParallel` and `InSequence`) uses kernel-threads. Novice users of the library usually assume that, being a concurrent library, each process is in its own kernel-thread. They make blocking calls to the OS in separate processes, and do not understand why (in previous versions, that used only user-threads) this blocked the other processes/user-threads. Therefore it is wise to make the more obvious functions start everything in a separate kernel-thread, unless the programmer explicitly states not to (usually for performance reasons, done by advanced users of the library) by using the `InParallelOneThread`/`InSequenceOneThread` calls.

The reader may notice that there is very little difference from the user's point of view between `InSequence` and `InSequenceOneThread`. The two are primarily included for completeness; they are used much less than the corresponding parallel calls, because sequence is already present in the C++ language. A call to "`Run(InSequence(A)(B));`" is equivalent to "`Run(A);Run(B);`".

4.1. Blocking Processes

As stated in the previous section, most processes can be run as a user-thread in the current kernel-thread, or in a new kernel-thread – decided by the programmer using the process, not the programmer that wrote the process. Some processes, for example a file-reading process, will make many blocking calls to the OS. If they are placed in a kernel-thread with other user-threads, this would block the other user-threads repeatedly. Therefore the programmer writing the file-reading process would want to make sure that the process being run will always be started in a new kernel-thread. Only the sub-processes of the file-reading process can occupy the same kernel-thread, otherwise it will be the only process in the kernel-thread.

This is done in C++CSP2 by inheriting from `ThreadCSProcess` instead of `CSProcess`. The type system ensures that the process can only be run in a new kernel-thread. This will not be necessary for most processes, but will be applicable for those processes repeatedly interacting with OS or similar libraries, especially if the call will block indefinitely (such as waiting for a GUI event, or similar).

5. Barrier Algorithm

Like JCSP, C++CSP2 offers a barrier synchronisation primitive. Unlike most implementations of barriers, dynamic enrollment and resignation is allowed. That is, the number of processes enrolled on a barrier is not constant. The implementation of barriers in JCSP (from which the original C++CSP barrier algorithm was taken) has a '`leftToSync`' count (protected by a mutex) that is decremented by each process that synchronises. The process that decrements the count to zero then signals all the other waiting threads and sets the `leftToSync` count back up to the number of processes enrolled (ready for the next sync). This section details a new replacement barrier algorithm for use in C++CSP2.

The idea of using software-combining trees to implement a barrier on a multi-processor system is described by Mellor-Crummey and Scott in [21]. The processors are divided into hierarchical groups. Each processor-group synchronises on its own shared counter, to reduce hot-spot contention (due to shared-cache issues, reducing the number of processors spinning on each shared 'hot-spot' is desirable). The last ('winning') processor to synchronise in the group goes forward into the higher-level group (which has a further shared counter) and so on until the top group synchronises. At this point the method is reversed and the processors go back down the tree, signalling all the shared counters to free all the blocked processes in the lower groups that the processor had previously 'won'.

This idea can easily be transferred to multi-threaded systems, with each thread blocking rather than spinning. A spinning thread is usually wasteful in a system with few processors but many threads. In order for it to finish spinning it will likely need to be scheduled out, and the other thread scheduled in to finish the synchronisation. Therefore, yielding or blocking is usually more efficient than spinning in this situation.

C++CSP2 uses a many-to-many threading approach. The software-combining tree approach can be adapted into this threading model by making all the user-threads in a given kernel-thread into one group, and then having another (higher-tier) group for all the kernel-threads. This forms a two-tier tree. This tree allows for optimisations to be made as follows.

Consider a group for all the user-threads in a kernel-thread. In C++CSP2 each user-thread is bound to a specific kernel-thread for the life-time of the user-thread. The user-threads of a particular kernel-thread can never be simultaneously executing. This means that a group shared among user-threads does not need to be protected by a mutex during the initial stages of the synchronisation, nor do the operations on it have to be atomic. This allows speed-up over the traditional barrier implementation where all the user-threads (in every kernel-thread) would always need to claim the mutex individually.

The code for this optimised version would look roughly as follows:

```
struct UserThreadGroup
{
    int leftToSync;
    int enrolled;
    ProcessQueue queue;
};

//Returns true if it was the last process to sync
bool syncUserThreadGroup(UserThreadGroup* group)
{
    addToQueue(group->queue,currentProcess);
    return (--(group->leftToSync) == 0);
}

void sync(UserThreadGroup* group)
{
    if (syncUserThreadGroup(group))
        syncKernelThread();
    else
        reschedule();
}
```

The `reschedule()` method makes the C++CSP2-kernel pick the next user-thread from the run-queue and run it. It does not automatically add the current user-thread back to the run-queue – it effectively blocks the current process.

Only the higher-tier group (that is shared among kernel-threads) needs to consider synchronisation. This group *could* be mutex-protected as follows:

```
int threadsLeftToSync;
map<KernelThreadId, UserThreadGroup > userThreadGroups;
Mutex mutex;

void syncKernelThread()
{
    mutex.claim();
    if (--(threadsLeftToSync) == 0)
    {
        int groupsLeft = userThreadGroups.size();
```

```
        for each group in userThreadGroups
        {
            group->leftToSync = group->enrolled;
            if (group->enrolled == 0)
            {
                remove group from userThreadGroups;
                groupsLeft -= 1;
            }
            freeAllProcesses(group->queue);
        }
        threadsLeftToSync = groupsLeft;
    }
    mutex.release();
}
```

The code only finishes the synchronisation if all the user-thread groups have now synchronised (that is, `threadsLeftToSync` is zero). The user-thread groups are iterated through. Each one has its `leftToSync` count reset. If no processes in that group remain enrolled, the group is removed. Finally, the `threadsLeftToSync` count is reset to be the number of kernel-threads (user-thread groups) that remain enrolled.

During this synchronisation, we modify the `UserTheadGroupS` of other kernel-threads, even though they are not mutex-protected. This is possible because for us to be performing this operation, all currently enrolled processes must have already synchronised (and hence blocked) on the barrier, so they cannot be running at the same time *until after the freeAllProcesses call* (which is why that call is made last in the for-loop). If a process tries to enroll on the barrier, it must claim the mutex first. Since we hold the mutex for the duration of the function, this is not a potential race-hazard. The resign code would look as follows:

```
void resign(UserThreadGroup* group)
{
    group->enrolled -= 1;
    if (--(group->leftToSync) == 0)
        syncKernelThread();
}
```

The enrolled count is decremented, as is the `leftToSync` count. If this means that all the user-threads in the group have now synchronised (or resigned), we must perform the higher-tier synchronisation. The mutex does not need to be claimed unless as part of the `syncKernelThread()` function. The enroll code is longer:

```
UserThreadGroup* enroll()
{
    UserThreadGroup* group;
    mutex.claim();
    group = find(userThreadGroups,currentThreadId);
    if (group == NULL)
    {   //Group did not already exist, create it:
        group = create(userThreadGroups,currentThreadId);
        group->enrolled = group->leftToSync = 1;
        threadsLeftToSync += 1; //Increment the count of threads left to sync
    } else
    {   //Group already existed:
        group->enrolled += 1;
        group->leftToSync += 1;
    }
    mutex.release();  return group;
}
```

There is one further (major) optimisation of the algorithm possible. All but the final thread to call `syncKernelThread()` will merely claim the mutex, decrement a counter and release the mutex. This can be simplified into an atomic decrement, with an attempt only being made to claim the mutex if the count is decremented to zero:

```
int threadsLeftToSync;
map<KernelThreadId, UserThreadGroup > userThreadGroups;
Mutex mutex;

void syncKernelThread()
{
    if (AtomicDecrement(&threadsLeftToSync) == 0)
    {
        mutex.claim();
        //Must check again:
        if (AtomicGet(&threadsLeftToSync) == 0)
        {
            int groupsLeft = 0;

            for each group in userThreadGroups
            {
                if (group->enrolled != 0)
                    groupsLeft += 1;
            }

            AtomicPut(&threadsLeftToSync,groupsLeft);

            for each group in userThreadGroups
            {
                group->leftToSync = group->enrolled;
                if (group->enrolled == 0)
                    remove group from userThreadGroups;

                freeAllProcesses(group->queue);
            }
        }
        mutex.release();
    }
}
```

There are some subtle but important features in the above code. The `threadsLeftToSync` count is first reset. This is important because as soon as any processes are released, they may alter this count (from another kernel-thread) without having claimed the mutex. Therefore the groups must be counted and the `threadsLeftToSync` variable set before freeing any processes. This could be rearranged to set the `threadsLeftToSync` count to the size of the `userThreadGroups` map at the start, and performing an atomic decrement on the `threadsLeftToSync` variable each time we find a new empty group. However, it is considered that the above method, with a single atomic write and two iterations through the map, is preferable to repeated (potentially-contested) atomic decrements and a single iteration through the map.

The other feature is that the `threadsLeftToSync` count is checked before and after the mutex claim. Even if our atomic decrement sets the variable to zero, it is possible for an enrolling process to then claim the mutex and enroll before we can claim the mutex. Therefore, once we have claimed the mutex, we must check again that the count is zero. If it is not zero (because another process has enrolled) we cannot finish the synchronisation.

5.1. Benchmarks

The proposed new algorithm is more complicated than a 'standard' barrier algorithm. This complexity impacts maintenance of the code and reduces confidence in its correctness; it has not been formally verified. In order to determine if the new algorithm is worthwhile, its speed must be examined. Barrier synchronisations were timed, the results of which are given below.

OS	Barrier	1x100	1x1000	1x10000	2x1	2x5000	100x1	100x100
Windows	New	20	370	7,500	3.5	5,900	170	6,400
	Standard	24	490	8,600	3.4	7,700	300	9,500
Linux	New	19	200	5,700	2.4	4,400	180	5,100
	Standard	21	400	6,400	2.9	5,600	240	7,100

All figures in microseconds, to 2 s.f. Each column heading is
*(Number of kernel-threads)*x*(Number of processes in **each** kernel-thread)*.
Each time is per single barrier-sync of all the processes.

The new algorithm is at least as fast as the standard algorithm in all cases bar one. As would be expected, the performance difference is most noticeable with many user-threads in each of many kernel-threads. The new algorithm eliminates use of the mutex among sibling user-threads, where the standard algorithm must claim the mutex each time – with competition for claiming from many other threads. The expectation is that with more cores (and hence more of these contesting threads running in parallel), the new algorithm would continue to scale better than the standard algorithm.

6. Mutexes

Most C++CSP2 algorithms (such as channels and barriers) use mutexes. Therefore fast mutexes are important to a fast implementation. As well as mutexes provided by the operating system (referred to here as OS mutexes) there are a number of mutexes based on atomic instructions that could be used. This section describes various mutex algorithms and goes on to provide benchmarks and analysis of their performance.

6.1. Spin Mutex

The simplest mutex is the spin mutex. A designated location in shared memory holds the value 0 when unclaimed, and 1 when claimed. An attempt at claiming is made by doing an atomic compare-and-swap on the value. If it was previously 0, it will be set to 1 (and therefore the mutex was claimed successfully). If it is 1, nothing is changed – the process must re-attempt the claim (known as spinning). Spinning endlessly on a system that has fewer processors/cores than threads is often counter-productive; the current thread may need to be scheduled out for the thread holding the mutex before a claim will be successful. Therefore C++CSP2 spins an arbitrary number of times before either scheduling in another process in the same thread or telling the OS to schedule another thread in place of the spinning thread (i.e. yielding its time-slice). For the purposes of this benchmark, the latter option was implemented.

6.2. Spin Mutex Test-and-Test-and-Set (TTS)

The TTS mutex was developed for multi-processor machines where an attempted atomic compare-and-swap would cause a global cache refresh. Multiple attempted claims on a much-contested location would cause what is known as the 'thundering herd' problem, where multiple caches in the system have to be updated with each claim. The TTS mutex spins on a

read-only operation, only attempting a claim if the read indicates it would succeed. Although the thundering herd problem should not occur on the benchmark system, the TTS mutex is included for completeness.

6.3. Queued Mutex

The Mellor-Crummey Scott (MCS) algorithm is an atomic-based mutex with strict FIFO (first-in first-out) queueing. It is explained in greater detail in [21], but briefly: it maintains a queue of processes, where the head is deemed to own the mutex. New claimers add themselves to the tail of the current list and spin (in the original MCS algorithm). When the mutex is released, the next process in the queue notices, implicitly passing it the mutex.

The MCS algorithm has been adapted to C++CSP2 by removing the spinning. Instead of spinning, the process immediately blocks after inserting itself into the queue. Instead of a process noticing the mutex is free by spinning, the releasing process adds the next process in the queue back to the appropriate run-queue. When it runs again, it implicitly knows that it must have been granted the mutex.

This mutex has the benefit of being strictly-FIFO (and hence avoids starvation) as well as having no spinning (except in a corner-case with unfortunate timing). The memory allocation for the queue is done entirely on the stack, which will be quicker than using the heap.

6.4. OS Mutex

Both Windows and Linux provide native OS mutexes. In fact, Windows provides two (a 'mutex' and a 'critical section'). They can be used as blocking or non-blocking, as described in the following sections.

6.4.1. Blocking Mutexes

Blocking mutexes cannot be used with C++CSP2. One user-thread cannot block *with the OS* on a mutex, because this would block the entire kernel-thread. Instead, processes (user-threads) must block with the C++CSP2-kernel, or not block at all (spinning or yielding). Therefore blocking OS mutexes are not a candidate for use with C++CSP2. The performance figures are given only for comparison, had C++CSP2 been purely kernel-threaded – in which case it could have used such mutexes.

6.4.2. Non-Blocking Mutexes

In contrast to the blocking mutexes, non-blocking OS mutexes are real candidates for use in C++CSP2.

6.5. Benchmarks

Benchmarks for each of the four mutexes are given below (five in the case of Windows). 'Uncontested' means that the mutex is claimed repeatedly in sequence by a single process – i.e. there is no parallel contention. '2x1' is two concurrent kernel-threads (each with one user-thread) repeatedly claiming the mutex in sequence. '10x10' is ten concurrent kernel-threads (each with ten concurrent user-threads) repeatedly claiming the mutex in sequence – a total of one hundred concurrent claimers.

OS	Mutex	Uncontested	2x1	10x10
Windows	Spin	30	86	6,100
	Spin TTS	33	140	4,100
	Queued	53	6,000	180,000
	OS (Mutex), B	1,000	5,500	280,000
	OS (Mutex), NB	1,100	2,800	230,000
	OS (Crit), B	53	360	19,000
	OS (Crit), NB	56	310	17,000
Linux	Spin	35	85	6,700
	Spin TTS	35	84	6,400
	Queued	53	3,500	180,000
	OS, B	62	150	13,000
	OS, NB	58	120	7,200

All figures in nanoseconds, to 2 s.f. Each column heading is *(Number of kernel-threads)*x*(Number of processes in **each** kernel-thread)*.
B = Blocking, NB = Non-Blocking.

6.6. Analysis

It is clear that the Windows 'mutex' is much slower than the alternatives, especially when uncontested.

Performance of the queued mutex is of the same order of magnitude as the other mutexes when uncontested, but scales badly. This is because of the continued interaction with the C++CSP run-queues. Consider what will happen if a process is preempted while holding a mutex in the 10x10 case. The next thread will be run, and each of the ten user-threads will probably queue up on the mutex. Then each of the further eight threads will run, and each of the ten user-threads in each will probably queue up on the mutex. So 90 user-threads in total may be scheduled. Compare this to the spin mutexes, where only 10 user-threads would be scheduled (each performing a thread-yield).

The reason for the queued mutex's appalling performance in the 2x1 case is not as immediately clear. A clue can be found in the performance on a single-core system, which is only a factor of two behind the fastest mutexes, rather than a factor of over 40. Consider the two threads running simultaneously (one on each core), repeatedly claiming and releasing. Each time a claim is attempted, it is reasonably likely that the other thread will hold the mutex. The second process will queue up, and if the release does not happen soon enough, the run-queue mutex will be claimed, and the condition variable waited upon. Thus, a wait on a condition variable is reasonably likely to happen *on each and every claim*. Therefore the performance is particularly bad for repeated claims and releases by kernel-threads with no other processes to run.

The Linux OS (now 'futex'-based [22]) mutex and Windows critical section work in a similar manner to each other. They first attempt to claim the mutex using atomic instructions. If that does not immediately succeed (potentially after spinning for a short time), a call is made to the OS kernel that resolves the contention, blocking the thread if necessary. Therefore when there is no or little contention the performance is very close to the spin mutexes, and only becomes slower when there is more competition and hence more calls need to be made to the OS kernel to resolve the contention.

The benchmarks were carried out with no action taking place while the mutex was held. For the channel mutexes, this is fairly accurate. Only a couple of assignments are performed while the mutex is held, and a maximum of two processes compete for the mutex. Therefore the best mutex for channels is clearly the spin mutex, which has the best performance with little or no contention.

The mutex for a barrier (under the new algorithm) is only claimed by an enrolling process or by the last process to sync (that is, it is only claimed once per barrier-sync, barring any enrollments). It is not contended if no processes are enrolling. Therefore the spin mutex is also the best choice for the barrier algorithm. The best mutex for the run-queues (explained in section 3.1) is similarly the spin mutex.

The other major use of a mutex is for shared channel-ends. Unlike all the other uses of a mutex, in this case the mutex will be held indefinitely (until the channel communication has completed). Therefore spinning is not advisable. The queued mutex is ideally suited for this case. While it does not perform as well as the other mutexes for quick claim-release cycles, it offers no spinning and strict-FIFO ordering, which suits shared channel-ends (to prevent starvation).

7. Channel Class Design

Like all the other CSP systems mentioned in this paper, C++CSP has the important concept of channels. Channels are typed, unidirectional communication mechanisms that are fully synchronised. In C++CSP, channels are templated objects that are used via their channel-ends (a reading end and a writing end).

C++CSP v1 had two channel-end types: `Chanin` and `Chanout` [4]. The former supplied methods for both alting and extended rendezvous, and threw an exception if an operation was attempted on a channel that did not support it (for example, channels with a shared reading-end do not support alting). This was bad design, and has now been rectified. There are now two channel reading ends (`Chanout` remains the only writing-end); `Chanin` and `AltChanin`. The former does not provide methods to support alting, whereas the latter does. In line with the latest JCSP developments [23], they both support extended rendezvous on all channels (including buffered channels).

In JCSP the `AltingChannelInput` channel-end is a sub-class of `ChannelInput`. However, in C++CSP2 `AltChanin` is not a sub-class of `Chanin`. This is because channel-ends in C++CSP2 are rarely held by pointer or reference, so sub-classing would be of no advantage (and indeed would suffer additional virtual function call overheads) – except when passing parameters to constructors; specifically, an `AltChanin` could be passed in place of a parameter of type `Chanin`. To facilitate this latter use, implicit conversions are supplied from `AltChanin` to `Chanin` – but not, of course, in the opposite direction.

8. Channel Algorithms

In [24] Vella describes algorithms for implementing CSP channels based on atomic instructions, for use in multi-processor systems. C++CSP2 even has an advantage over the constraints that Vella had to work with. Vella is careful to not re-add a process to the run-queue before it has blocked, in case another thread takes it off the run-queue and starts running it simultaneously. In C++CSP2, this is not possible, because processes cannot move between threads (so it will only be re-added to the run-queue for its own thread).

C++CSP2 does not use Vella's algorithms however, because the complications that are added by supporting poisoning have not yet been resolved with the difficult atomic algorithms. Instead, a mutex is used to wrap around the channel algorithms (one mutex per channel). There are two other changes from the original C++CSP channel algorithms (described in [4]), which are motivated in the following two sub-sections on poison and destruction.

8.1. Poison

C++CSP has always offered poisonable channels. Poisoning a channel is used to signal to other processes using that channel that they should terminate. Either end of a channel can be used to poison it, and both ends will 'see' the poison (a poison exception will be thrown) when they subsequently try to use the channel.

The channel algorithms in C++CSP v1 had a curious behaviour with regards to poison. Imagine, for example, that a reader was waiting for input on a channel. A writer arrives, provides the data and completes the communication successfully. As its next action the writer poisons the channel. When the reader wakes up, it sees the poison straight away and throws a poison exception. The data that the writer thought had 'successfully' been written is lost. This could be further obscured if on a shared channel, one writer completed the communication and another writer did the poisoning.

Sputh treats this as a fault in his JCSP algorithm, and corrects it [25]. I think that his decision is correct, and the consequent implication that C++CSP's original semantics (with regards to poison) were flawed is also correct. This problem is solved by introducing an additional state flag into the channel, which indicates whether the last communication completed successfully (before the poison) or not (it was aborted due to poison).

Another area in which poison semantics have been corrected are buffered channels. Previously, when a writer poisoned a buffered channel, the reader would see the poison immediately, even if there was unread data in the buffer. This caused a similar problem to the one above – data that the writer viewed as successfully sent would be lost. The new effects of poisoning buffered channels are summarised below:

Writer poisons the channel: The channel is flagged as poisoned, the buffer is not modified.
Reader poisons the channel: The channel is flagged as poisoned, and the buffer is emptied.
Writer attempts to use the channel: Poison is always noticed immediately.
Reader attempts to use the channel: Poison is noticed only when the buffer is empty.

The semantics are asymmetric. The simplest rationale behind their choice is that poisoning a channel that uses a first-in first-out buffer of size N now has a similar effect to poisoning a chain of N identity processes.

8.2. Destruction

There are often situations in which the user of C++CSP2 will want to have a single server process serving many client processes. Assuming the communication between the two is a simple request-reply, the server needs some way of replying specifically to the client who made the request. One of the easiest ways of doing this is for the client to send a reply channel-end with its request (channel-ends being inherently mobile in C++CSP2):

```
//requestOut is of type Chanout< pair< int,Chanout<int> > >
//reply is of type int
{
    One2OneChannel<int> replyChannel;
    requestOut << make_pair(7,replyChannel.writer());
    replyChannel.reader() >> reply;
}
```

The corresponding server code could be as follows:

```
//requestIn is of type Chanin< pair< int,Chanout<int> > >
pair< int, Chanout<int> > request;
requestIn >> request;
request.second << (request.first * 2);
```

For this trivial example, requests and replies are integers, and the server's answer is simply double the value of the request.

If the old algorithms were used, this code would be potentially unsafe. The following trace would have been possible:

1. Client sends request, server receives it.
2. Server attempts to send reply, must block (waiting for the client).
3. Client reads reply, adds server back to the run-queue.
4. Client continues executing, destroying `replyChannel`.
5. Server wakes up and needs to determine whether it woke because it was poisoned or because the communication completed successfully. The server checks the poison flag; a member variable of `replyChannel`.

This situation thus leads to the server checking a flag in a destroyed channel. To help avoid this problem, the first party to the channel (the one who must wait) creates a local stack variable that will be used to indicate whether the communication completed successfully, and puts a pointer to it in a channel variable. The second party uses the pointer to modify the variable. When the first party wakes up, it can then check its local stack variable successfully, even if the channel has been destroyed.

9. Scoped Forking

Scope is a useful part of structured programming. In most languages, variable storage is allocated when variables come into scope and de-allocated when variables go out of scope. In C++ classes this concept is built on to execute a constructor when an object variable comes into scope, and a destructor to be called when an object variable goes out of scope. This feature, which is not present in Java, can be both useful and dangerous in the context of C++CSP2. Both aspects are examined in this section.

C++CSP2 takes advantage of the scope of objects to offer a `ScopedForking` object that behaves in a similar manner to the `FORKING` mechanism of occam-π [26]. In occam-π, one might write:

```
FORKING
  FORK some.widget()
```

In C++CSP2 the equivalent is:

```
{
    ScopedForking forking;
    forking.fork(new SomeWidget);
} //end of block
```

The name of the `ScopedForking` object is arbitrary (`forking` is as good a name as any). At the end of the scope of the `ScopedForking` object (the end of the block in the above code), the destructor waits for the forked processes to terminate – the same behaviour as at the end of the `FORKING` block in occam-π.

The destructor of a stack object in C++ is called when the variable goes out of scope – this could be because the end of the block has been reached normally, or because the function was returned from, or an exception was thrown. In these latter two cases the destructor will still be executed.

For example:

```
{
    ScopedForking forking;
    forking.fork(new SomeWidget);

    if (something == 6)
        return 5;

    if (somethingElse == 7)
        throw AnException();
}
```

Regardless of whether the block is left because of the return, the throw, or normally, the code will only proceed once the `SomeWidget` process has terminated. Using such behaviour in the destructor allows us to emulate some language features of occam-π in C++, and even take account of C++ features (such as exceptions) that are not present in occam-π. However, there is one crucial difference – the occam-π compiler understands the deeper meaning behind the concepts, and can perform appropriate safety checks. In contrast, the C++ compiler knows nothing of what we are doing. In section 8.2, one potential problem of using objects concurrently was demonstrated. There are two further mistakes that can be made using the new `ScopedForking` concept, which are explained in the following two sub-sections.

9.1. Exception Deadlock

Consider the following code:

```
One2OneChannel<int> c,d;
try
{
    ScopedForking forking;
    forking.fork(new Widget(c.reader()));
    forking.fork(new Widget(d.reader()));
    c.writer() << 8;
    d.writer() << 9;
}
catch (PoisonException)
{
    c.writer().poison();
    d.writer().poison();
}
```

At first glance this code may seem sensible. The try/catch block deals with the poison properly, and the useful `ScopedForking` process makes sure that the sub-processes are waited for whether poison is encountered or not. Consider what will happen if the first `Widget` process poisons its channel before the example code tries to write to that channel. As part of the exception being thrown, the program will destroy the `ScopedForking` object *before* the catch block is executed. This means that the program will wait for both `Widget`s to terminate before poisoning the channels. If the second `Widget` is waiting to communicate on its channel, then deadlock will ensue.

This problem can be avoided by moving the declaration of the `ScopedForking` object to outside the try block. The general point, however, is that the C++ compiler can offer no protection against this mistake. In a language such as Rain [27], which offers both concurrency and poison exceptions, the compiler could avoid such problems by detecting them at compile-time in the first place, or by ensuring that all catch blocks for poison are executed before the wait for sub-processes.

9.2. Order of Destruction

Consider the following code:

```
{
    ScopedForking forking;
    One2OneChannel<int> c,d;
    forking.fork(new WidgetA(c.reader(),d.writer()));
    forking.fork(new WidgetB(d.reader(),c.writer()));
}
```

This code creates two processes, connected together by channels, and then waits for them to complete[2]. This code is very unsafe. In C++, objects are constructed in order of their declaration. At the end of the block, the objects are destroyed in reverse order of their declaration. This means that at the end of the block in the above code, the channels will be destroyed, and then the `ScopedForking` object will be destroyed. So the processes will be started, the channels they are using will be destroyed, and then the parent code will wait for the processes to finish, while they try to communiate using destroyed channels.

Again, this problem can be avoided by re-ordering the declarations. This code is dangerous (in the context of our example):

```
ScopedForking forking;
One2OneChannel<int> c,d;
```

This code is perfectly safe:

```
One2OneChannel<int> c,d;
ScopedForking forking;
```

The subtle difference between the two orderings, the non-obvious relation between the two lines, and the ramifications of the mistake (in all likelihood, a program crash) make for a subtle error that again cannot be detected by the C++ compiler. In languages such as occam-π or Rain, this mistake can be easily detected at compile-time (variables must remain in scope until the processes that use them have definitely terminated) and thus avoided.

The documentation for the C++CSP2 library explains these pitfalls, and offers design rules for avoiding the problems in the first place (for example, always declare all channels and barriers outside the block containing the `ScopedForking` object). The wider issue here is that adding concurrency to existing languages that have no real concept of it can be a dangerous business. Compile-time checks are the only real defence against such problems as those described here.

10. Networking

C++CSP v1 had in-built support for sockets and networked channels, as detailed in [5]. The network support was integrated into the C++CSP kernel; every time the kernel was invoked for a context switch, it checked the network for new data, and attempted to send out pending transmissions. Happe has benchmarked this model against other models (with different threading arrangements) [28]. His results showed that using separate threads (one for waiting on the network and one for processing requests) produced the best performance. C++CSP2's network support (which has not yet been implemented) will be considered in light of these results, and with consideration for facilities available only on Linux (such as `epoll`) or Windows (such as I/O completion ports).

[2]The same effect could have easily been achieved *safely* using the `Run` and `InParallel` constructs demonstrated in section 4.

11. Conclusions

C++CSP2 now supports true parallelism on multicore processors and multi-processor systems. This makes it well-positioned as a way for C++ programmers to take advantage of this parallelism, either by wrapping the process-oriented methodology that C++CSP2 offers around existing code, or by developing their programs on top of C++CSP2 from the outset.

This paper has presented benchmarks of various mutexes and selected the fastest for C++CSP2's algorithms. Where possible these algorithms have actually used atomically-updated variables, avoiding the use of mutexes, in order to reduce contention for mutexes and minimise the chance of processes being scheduled out while holding a mutex. The effect of this work is to make C++CSP2 as fast as possible on multicore and multi-processor machines by reducing spinning and blocking to a minimum.

This work should prove relevant to the efforts to take advantage of multicore processors in other CSP implementations. The atomic alting algorithm described in section 3.3 could prove useful in JCSP, while the barrier algorithm and mutex benchmarks may be applicable to the implementation of occam-π.

The major opportunities for future work are implementing the network support (mentioned briefly in section 10) and formally proving some of the new algorithms presented in this paper. There are also new features being added to JCSP, such as alting barriers, output guards and broadcast channels [23] that would be advantageous to add to C++CSP2.

The C++CSP2 library will have been released before this paper is published, and can be found at [9]. In this new version, particular effort has been put into improving and expanding the documentation to make the library accessible to both novice and advanced users.

11.1. Final Thought

"I don't know what the next major conceptual shift will be, but I bet that it will somehow be related to the management of concurrency." – Bjarne Stroustrup, designer of C++.

References

[1] Anton Shilov. Single-core and multi-core processor shipments to cross-over in 2006 – Intel. `http://www.xbitlabs.com/news/cpu/display/20051201235525.html`, 10 February 2007.

[2] Fred Barnes. occam-pi: blending the best of CSP and the pi-calculus. `http://www.occam-pi.org/`, 10 February 2007.

[3] Peter H. Welch. Java Threads in Light of occam/CSP (Tutorial). In Andrè W. P. Bakkers, editor, *Proceedings of WoTUG-20: Parallel Programming and Java*, pages 282–282, 1997.

[4] Neil C. C. Brown and Peter H. Welch. An Introduction to the Kent C++CSP Library. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 139–156, 2003.

[5] Neil C. C. Brown. C++CSP Networked. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 185–200, 2004.

[6] B. Orlic. and J.F. Broenink. Redesign of the C++ Communicating Threads Library for Embedded Control Systems. In F. Karelse STW, editor, *5th Progress Symposium on Embedded Systems*, pages 141–156, 2004.

[7] Alex Lehmberg and Martin N. Olsen. An Introduction to CSP.NET. In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, pages 13–30, 2006.

[8] Kevin Chalmers and Sarah Clayton. CSP for .NET Based on JCSP. In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, pages 59–76, 2006.

[9] Neil Brown. C++CSP2. `http://www.cppcsp.net/`, 10 February 2007.

[10] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[11] Peter Welch. Communicating Sequential Processes for Java. `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`, 10 February 2007.

[12] Gerald Hilderink. Communicating Threads for Java. `http://www.ce.utwente.nl/JavaPP/`, 10 February 2007.

[13] Fred Barnes. Kent Retargetable occam Compiler. `http://www.cs.kent.ac.uk/projects/ofa/kroc`, 10 February 2007.

[14] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, 1992.

[15] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 110–121, New York, NY, USA, 1991. ACM Press.

[16] Per Brinch Hansen. *Operating System Principles*. Prentice-Hall, 1973.

[17] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[18] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Commun. ACM*, 23(2):105–117, 1980.

[19] Peter H. Welch and Jeremy M. R. Martin. Formal Analysis of Concurrent Java Systems. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 275–301, 2000.

[20] Thorsten Ottosen. Boost.Assignment Documentation. `http://www.boost.org/libs/assign/doc/`, 10 February 2007.

[21] John M Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[22] Ulrich Drepper. Futexes are tricky. Technical Report 1.3, Red Hat, December 2005.

[23] Peter Welch, Neil Brown, Bernhard Sputh, Kevin Chalmers, and James Moores. Integrating and Extending JCSP. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch, editors, *Communicating Process Architectures 2007*, pages –, 2007.

[24] Kevin Vella. *Seamless Parallel Computing On Heterogeneous Networks Of Multiprocessor Workstations*. PhD thesis, University of Kent, 1998.

[25] Bernhard Herbert Carl Sputh. *Software Defined Process Networks*. PhD thesis, University of Aberdeen, August 2006. Initial submission.

[26] Fred Barnes and Peter Welch. Prioritised Dynamic Communicating Processes - Part I. In James Pascoe, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, pages 321–352, 2002.

[27] Neil C. C. Brown. Rain: A New Concurrent Process-Oriented Programming Language. In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, pages 237–252, 2006.

[28] Hans Henrik Happe. TCP Input Threading in High Performance Distributed Systems. In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, pages 203–214, 2006.