Multiple Flows of Control in Migratable Parallel Programs

Gengbin Zheng, Orion Sky Lawlor *, Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
gzheng@uiuc.edu, olawlor@acm.org, kale@uiuc.edu

Abstract

Many important parallel applications require multiple flows of control to run on a single processor. In this paper, we present a study of four flow-of-control mechanisms: processes, kernel threads, user-level threads and event-driven objects. Through experiments, we demonstrate the practical performance and limitations of these techniques on a variety of platforms. We also examine migration of these flows-of-control with focus on thread migration, which is critical for application-independent dynamic load balancing in parallel computing applications. Thread migration, however, is challenging due to the complexity of both user and system state involved. In this paper, we present several techniques to support migratable threads and compare the performance of these techniques.

1 Introduction

When a program consists of many concurrent tasks, it is often simplest to represent these tasks with independent flows of control. Concurrency is required by many applications.

- Large scale scientific and engineering applications such as molecular dynamics simulation code [33], where the computation on each partitioned molecular system cube space can be treated as a separate flow of control.
- Processor virtualization in parallel programming [20, 21], which simplifies development and improves performance by dividing up the work on one physical processor into many "virtual processors", each an independent flow of control.
- Large parallel machine simulation, where separate flows of control can be used to represent each simulated physical processor [40].
- Parallel discrete event simulations, where each simulation object can be treated as a separate flow of control [39].
- Web and other network servers, where communication with each client can be handled by a separate flow of control.

In this paper, we will focus on high-concurrency parallel programs, and use the term "flows" instead of "threads-of-control" to avoid confusion with threads, which are a particular mechanism for encapsulating a control flow.

One processor can only execute one flow of control at a time. A flow-of-control may suspend its execution and give control to another flow either because of external reasons, such as an interrupt indicating the end of a time slice; or because of internal reasons, such as an explicit yield or a wait on an event or resource. A suspended flow-of-control can be resumed at a later time.

There are several methods for supporting multiple flows of control. The oldest example of this sort of concurrent programming is coroutines [27]. Some methods are supported by the OS kernel itself, including separate processes as described in Section 2.1, and kernel threads in Section 2.2. Other methods are like coroutines in that they are defined and supported entirely by user code, such as user-level threads of Section 2.3 and event-driven objects of Section 2.4. We analyze the performance of these methods in Section 4.

On multiprocessor systems, distributing execution of flows-of-control over multiple processors can exploit parallelism and thus achieve improved performance. However, the load in parallel applications may change over time, and to keep overloaded processors from bottlenecking the overall computation, we must perform load balancing [35]. Although migrating flows-of-control between processors is challenging, it presents a useful way to implement application-independent load balancing [41]. Section 3 describes obstacles to and techniques for migrating the flows of control across processors.

*Department of Computer Science, University of Alaska Fairbanks

2 Mechanisms for Supporting Multiple Flows of Control

A flow of control is a single sequential stream of executed instructions, including subroutine calls and returns. Normal machines directly represent a flow of control as the set of machine registers and the machine's execution stack. The scheduling of multiple flows of control on a processor can be supported in either OS kernel or user code.

The basic methods for flows of control examined in this paper include processes, kernel threads, user-level threads and event-driven objects.

2.1 Processes

The simplest and oldest flow of control is the process, which wraps a complete address space around a flow of control. All modern machines support multiple processes, although some parallel machines or schedulers do not allow them or place restrictions on them. For example, Myricom's GM interface does not allow `fork` or `system` calls while Myrinet network ports are open. Other exceptions include the Blue Gene/L [2] and ASCI Red [36] microkernels, which do not have conventional virtual memory systems and hence do not support UNIX system calls such as `fork`, `system` and `exec`.

The advantage of the process model is its complete separation of state. Because a processes' flow of control is completely walled off within its own separate address space and operating system context, processes fully support separate global variables, memory management, blocking I/O calls, and signals.

However, for parallel programming, the total separation of the process model makes it more difficult to communicate between the various flows of control that make up a program. Processes can only interact using relatively limited, cumbersome methods such as pipes and sockets, or with more difficult explicitly shared memory regions such as SYSV Interprocess Communication (IPC). Furthermore, processes are considered "heavy-weight". The substantial amount of per-process kernel state increases the amount of memory used by each process, and increases the overhead of process creation and switching. Worse, some operating systems have a fixed and fairly low limit on the number of processes that can be created. Table 2 in Section 4 summarizes these practical limitations on several stock systems.

2.2 Kernel Threads

Kernel threads consist of a set of registers, a stack, and a few corresponding kernel data structures. When kernel threads are used, the operating system will have a descriptor for each thread belonging to a process and it will schedule all the threads. Unlike processes, all threads within a process share the same address space. Similar to processes, when a kernel thread makes a blocking call, only that thread blocks. All modern machines support kernel threads, most often via the POSIX threads interface "pthreads". Some dedicated parallel machines support kernel threads poorly or not at all. For example, the Blue Gene/L microkernel does not support pthreads.

The purported advantage of kernel threads over processes is faster creation and context switching compared with processes. For shared-memory multiprocessor architectures, the kernel is able to dispatch threads of one process on several processors, which leads to automatic load balancing within the nodes. For parallel programming, threads allow different parts of the parallel program to communicate by directly accessing each others' memory, which allows very efficient, fine-grained communication.

Kernel threads share a single copy of the entire address space, including regions such as global data that may cause conflicts if used by multiple threads simultaneously. Threads can also cause unintentional data sharing, which leads to corruption and race conditions. To avoid this unintentional sharing, programs must often be modified to either lock or access separate copies of common data structures. Several very widely used language features are unsafe when used with threads, such as the use of global and static variables, or the idiom of returning a reference to a static buffer. Especially with large existing codebases with many global variables, this makes kernel threads very difficult to use because in most implementations of kernel threads, it is not possible to assign each thread a private set of global variables.

Kernel threads are considered "lightweight," and one would expect the number of threads to only be limited by address space and processor time. Since every thread needs only a stack and a small data structure describing the thread, in principle this limit should not be a problem. But in practice, we found that many platforms impose hard limits on the maximum number of pthreads that can be created in a process. Table 2 in Section 4 shows the practical limitations on pthreads on several stock systems.

In particular, operating system kernels tend to see kernel threads as a special kind of process rather than a unique entity. For example, in the Solaris kernel threads are called "light weight processes" (LWP's). Linux actually creates kernel threads using a special variation of fork called "clone," and until recently gave each thread a separate process ID. Because of this heritage, in practice kernel threads tend to be closer in memory and time cost to processes than user-level threads, although recent work has made some progress in closing the gap, including K42 [5] and the Native POSIX Threading Library (NPTL) and Linux O(1) scheduler.

2.3 User-Level Threads

Like a kernel thread, a user-level thread includes a set of registers and a stack, and shares the entire address space with the other threads in the enclosing process. Unlike a kernel thread, however, a user-level thread is handled entirely in user code, usually by a special library that provides at least start, swap and suspend calls. Because the OS is unaware of a user-level thread's existence, a user-level thread can not separately receive signals or use operating system scheduling calls such as sleep(). Many implementations of user-level threads exist, including: GNU Portable Threads (Pth) [1], FreeBSD's userland threads, QuickThreads [26] and those developed by us for the Charm++ system [25].

The primary advantages of user-level threads are efficiency and flexibility. Because the operating system is not involved, user-level threads can be made to use very little memory, and can be created and scheduled very quickly. User-level threads are also more flexible because the thread scheduler is in user code, which makes it much easier to schedule threads in an intelligent fashion — for example, the application's priority structure can be directly used by the thread scheduler [9].

The primary disadvantage of user-level threads compared to kernel threads is the lack of operating system support. For example, when a user-level thread makes a blocking call, the kernel does not start running another user-level thread. Instead, the kernel suspends the entire calling kernel thread or process, even though another user-level thread might be ready to run. To avoid this blocking problem, some systems such as AIX and Solaris support "N:M" thread scheduling, which maps some number N of application threads onto a (usually smaller) number M of kernel entities. There are two parties, the kernel and the user parts of the thread system, involved in each thread operation for N:M threading, which is complex. The blocking problem can also be avoided by building a smarter runtime layer which intercepts blocking calls, replaces them with a non-blocking call, and starts another user-level thread while the call proceeds [1]. Yet another approach is to provide support in the kernel to notify the user-level scheduler when a thread blocks, often called "scheduler activations" [3, 38].

Since user-level threads are controlled in user code, there is virtually no limit on the maximum number of threads as long as the resource allows. In practice, one can create 50,000 user-level threads on a Linux box very easily (see Table 2).

2.4 Event-Driven Objects

Rather than storing and restoring machine registers to pause and resume execution, the program can be divided into pieces each of which manually stores and restores their state. A single "scheduler" routine then provides the glue to execute these pieces in the appropriate sequence. For example, rather than making a blocking call, the event-driven style would post an asynchronous request, then explicitly return control back to the scheduler. Once the request completes, the scheduler will call the object again to resume execution. We have explored this idea in Charm++ runtime system [24] extensively.

This "event-driven" style avoids operating system and other low-level problems completely. Because suspending and resuming execution is simply a function call, the event-driven style can also be very efficient.

However, the event-driven style can be more difficult for programmers. In particular, if it becomes necessary to block from inside a deeply nested subroutine call, all the calling routines must be changed to pass control back up to the scheduler. Also, a reactive and event-driven style of programming obscures the description of the life-cycle of an object. For example, a program would specify that *when this message A arrives, execute method F; when B arrives, execute G*, but can not specify that the object is going to repeatedly process A and B messages in alternating sequence k times [32].

2.4.1 Return-Switch Functions

A C or C++ subroutine can be written in a return-switch style to mimic thread suspend/resume. When the subroutine is "suspended", it returns instead of blocking with a flag indicating the point it left off. When the subroutine is "resumed", the same subroutine is called with the flag which can then be used in a "goto" or "switch" statement to resume execution at the point it left off. It is possible to wrap a technique similar to Duff's Device inside a set of macros to make this "save, return, and resume from label" process mostly transparent to the programmer [37]. However this technique can still be confusing, error-prone and tough to debug.

2.4.2 Structured Dagger

Subroutines can also be suspended without using threads or macros by applying a simple pre-processor. We developed Structured Dagger (SDAG) [22] as a coordination language for expressing the control flow within a Charm++ parallel object naturally using certain C language-like constructs. In parallel programming, it leads to a style of programming which cleanly separates parallel from sequential functionality.

Figure 1 shows an example of a parallel 5-point stencil program with 1-D decomposition and ghost cell exchange written in SDAG. In the program, the **for** loop implements an outer iteration loop. Each iteration begins by calling *sendStripToLeftAndRight* in an **atomic** construct to send

```
entry void stencilLifeCycle()
{
  for (i=0; i<MAX_ITER; i++)
  {
    atomic {sendStripToLeftAndRight();}
    overlap
    {
      when getStripFromLeft(Msg *leftMsg)
         { atomic { copyStripFromLeft(leftMsg); } }
      when getStripFromRight(Msg *rightMsg)
         { atomic { copyStripFromRight(rightMsg); } }
    }
    atomic{ doWork(); /* loops over interior */ }
  }
}
```

Figure 1. Sample code in Structured Dagger

out messages to the neighbors.[1] The **overlap** immediately following asserts that the two events corresponding to *getStripFromLeft* and *getStripFromRight* can occur and be processed in any order. The **when** construct simply says that when a message (e.g. *getStripFromLeft*) arrives for the construct, it invokes the **atomic** action which calls a plain C++ function *copyStripFromLeft* to process the message. After both **when** constructs are executed, function *doWork* in the last **atomic** construct will be invoked and the program enters the next iteration of the for loop. The Structured Dagger preprocessor transforms all this syntax into code for an efficient finite-state machine, which receives and processes the network messages at runtime.

Overall, the event-driven style can be made quite efficient and reasonably easy to program. However, the event-driven style is difficult to apply to existing codes — in particular, most parallel programming interfaces like the Message Passing Interface (MPI) are written in terms of blocking subroutine calls like MPI_Recv. A traditional C-compatible compiled library can only hope to switch tasks at these blocking calls by saving and restoring the machine's stack and registers in a thread-like fashion. Hence for the remainder of this paper, we focus on supporting threads.

3 Migration

Migration, the process of moving work from one processor to another, is a very flexible tool that can be used to solve a variety of problems in parallel computing. For example, migration can be used to improve load balance, by migrating work away from overloaded processors [11, 4, 41, 6]. Migration can improve communication performance, by moving pieces of work that communicate with each other closer together [33]. Migration can allow all the work to be moved off a processor [17] to allow another job to run there [10], or to vacate a node that is expected to fail or be shut down [34]. Migration techniques can also be used to implement checkpoint/restart for fault tolerance [12, 42] — under this model, checkpointing is simply migration to disk or the local memory of a remote processor. In this section, we discuss issues related to the migration of the flows of control. However, we will mainly focus on the thread migration techniques that we developed in the Charm++/AMPI runtime systems.

[1]The atomic construct encapsulates sequential C/C++ language code.

3.1 Migration State

To migrate a piece of work to a new processor, we must ensure that all needed state information will be available on the new processor. In a fully shared-memory parallel system, this is trivial—because all processors implicitly share all state, migration simply means transferring control to the new processor, which will be able to access the work's state directly.

In a distributed-memory parallel system, we must either proactively ship all needed state along with the piece of work; or else lazily ship state as it is needed. Lazy memory state shipping is commonly done by software distributed shared memory systems. State that is shared between several threads clearly can not simply be shipped to a new location, and so must be either shuffled back and forth at runtime (costing performance) or tightly regulated or banned (costing flexibility). We currently allow shared state only via well-defined interfaces in our own work related to Charm++ and AMPI runtime.

The state required by a migrating control-of-flow invariably falls into one of three categories:

- Memory state, which includes dynamically allocated data stored in the heap, local variables stored in the stack, and global variables.
- Communication state, which includes pending outgoing and incoming network messages.
- Kernel state, which includes open files, mapped and shared regions of memory, and pending signals.

Most migration systems only support shipping a subset of the possible state, forcing users to manually reconstruct other state. In the case of migratable threads, when several threads in a process can share memory, kernel, and communication state, it can be very difficult to extract the set of resources needed by a single thread. In our Charm++/AMPI runtime system, we only allow different threads to share state in a small number of well-defined ways, which makes it possible to efficiently migrate a single thread to a different address space.

3.1.1 Memory State Migration

Migrating memory state is conceptually quite straightforward — all needed memory is collected and shipped to the new processor. This is complicated in practice by various low-level difficulties.

For heap data, we must collect all the allocated data for a particular piece of work. For object-oriented applications, we have built a convenient and general-purpose framework for describing and shipping complicated user-defined objects called the PUP (Pack/UnPack) Framework [19]. For more general applications, it is possible to provide hooks to the memory allocation routines (malloc) to capture all the memory allocated by each thread.

For global variables, process migration is straightforward because each process image contains a separate copy of the global variables. For thread migration, however, if several threads in a process share global variables, it may be impossible to extract the value needed by each thread. Our current solution is to privatize global variables so that each user-level thread has its own copy of the global variables. Thus migration of global variables is conceptually simplified. In order to support the transparent privatization of global variables for existing codebases, we implemented a *swap-global* scheme by analyzing the Executable and Linking Format (ELF) object files in a way similar to the Weaves runtime framework [31]. A dynamically linked ELF file format executable always accesses global variables via the Global Offset Table (GOT), which contains one pointer to each global variable. To make separate copies of the global variables, we then simply make separate copies of the GOT — one for each user-level thread. The thread scheduler then swaps the GOT when switching threads.

3.1.2 Communication State Migration

In our runtime system, migratable entities only communicate via the communication sub-system, which provides a location independent communication that supports migration at any time. We have constructed an efficient communication system that allows object or thread migration with ongoing point-to-point and collective communication. As this system is described in our earlier work [28], we will not describe it in detail here.

3.1.3 Kernel State Migration

Kernel state includes data managed by the OS for the application — for example, the state of open files, signals. Kernel state is mostly invisible to users and is platform-dependent, so very few systems support kernel state migration. Instead, kernel state is often treated as a non-migratable portion of a migrating process. For example, Mosix [6] divides migrating process into migratable user context, and non-migratable system context information. Our runtime system currently does not support migration of kernel state.

3.2 Event-driven Objects

The simplest kind of migration is for event-driven objects [41, 8, 7, 10, 11]. In Charm++ runtime system, event-driven objects are normally location-independent, requiring little persistent runtime or system state. Because the entire execution state normally consists of a few application data structures and the name of the next event to run, to migrate to a new processor we need only copy these data structures to a new processor and begin executing the next event.

3.3 Process Migration

Because processes provide a well-defined memory, kernel, and communication interface, process migration is an old and widely implemented technique. Since the entire address space is migrated, all the pointers in the user application are still valid on the new processor. Systems that support process migration include Sprite [13], Condor [29], MOSIX [6], and many others, including the recent VMADUMP interface for Linux [14]. An extensive survey is available from the ACM [30]; so we will not go into further detail here.

3.4 Thread Migration

Thread migration is very challenging due to the fact that the system and user state of a thread is only one part of an enclosing process. It is often very difficult to extract the state of one thread from the others in a process. For example, heap data allocated by a thread may be shared by multiple threads. In our work, we assume that the user data allocated by one thread is used only by that thread. Data sharing among threads can still be achieved via read-only global varaibles or fast local message passing via the thread scheduler.

Another difficult aspect of thread migration is the fact that a thread stack contains a large number of pointers (including function return addresses, frame pointers, and pointer variables), and many of these point into the stack itself. Thus, if we copy the thread stack to another processor, all these pointers must be updated to point to the new copy of the stack instead of the old copy. However, because the stack layout is determined by the vagaries of the machine architecture and compiler, there is no simple and portable method by which all these pointers can even be identified, much less changed.

A feasible approach for thread migration, then, is to guarantee that the stack will have exactly the same address on the new processor as it did on the old processor. Because the stack addresses haven't changed, no pointers need to be updated because all references to the original stack's data remain valid on the new processor. This idea was originally developed for thread migration in the PM2 runtime system [4]. In the following subsections, we present three mechanisms to ensure that the stack's address remains the same after migration. These mechanisms for migration apply to both kernel and user-level threads. However, in this paper, we will mainly focus on the migration of the user-level threads supported by the Charm++ and AMPI runtime systems.

3.4.1 Stack Copying Threads

A naive implementation of this approach called "stack-copying threads" is simply to pick *one* address for the stack system-wide. The thread scheduler then copies each thread's stack data into this address before executing the thread, and copies it out again when the thread is suspended. Because there is only one address used by all active thread stacks even on different processors, migrating a thread is simple.

One complicating factor is that modern machines may not always allocate the system stack at the same virtual memory address on each processor. A fixed stack address makes it easier to mount a buffer overflow ("stack smashing") attack, so some systems change the stack's starting address from run to run as a security measure. On such systems, it is thus impossible to use the system stack for stack copying threads because the stack address is different on each machine.

Stack copying threads suffer from the high cost incurred by stack copying. Because the entire stack must by copied for each thread switch, changing threads is quite slow, especially when the stack contains a large amount of data. Further, because there is only one stack location, there can only be one thread active in each address space, which means a machine with two physical processors can not run two stack-copying theads from the same address space simultaneously.

3.4.2 Isomalloc Thread Migration

The PM2 implementaton of "isomalloc" [4] overcomes these disadvantages by allocating a globally unique address for each thread stack. To avoid conflicts between threads, stack addresses must be unique across the entire parallel machine, which makes allocating stacks somewhat complicated. As illustrated in Figure 2, the PM2 approach is to divide up the entire unused virtual memory address space, or "isomalloc region", into per-processor slots which can subsequently be allocated in parallel. This isomalloc region must be agreed upon by all processors at startup — normally the largest space available lies between the process stack and the heap. A processor can then grant any local thread a new globally reserved range of virtual addresses from within that processor's region of the shared address space. The threads can then be confident that they can migrate to any other processor, and their addresses will be free for use on that processor.

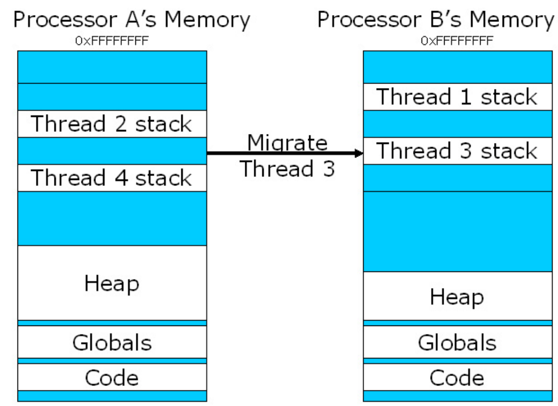

Figure 2. Migrating a thread stack allocated with isomalloc.

This approach can be seen as a sort of distributed shared memory system, in that each thread is using globally unique virtual memory addresses. However, because threads never directly share data, we can eliminate the possibility of DSM page faults by sending all a thread's data along with the thread at migration time.

Clearly, with n threads per processor, s bytes per thread, and p processors, the isomalloc approach uses at least nsp bytes of address space. For 10 threads per processor, 1MB of data per thread, and 1000 processors, this amounts to 10 gigabytes of address space! But luckily we can use the virtual memory hardware to avoid using such a large amount of physical memory. The system call `mmap` allows individual pages of program virtual address space to be assigned to physical memory, so on each processor we assign physical memory only to the addresses in use by local threads. Addresses used by all remote threads are claimed only in principle, but never actually allocated physical memory unless that remote thread migrates in.

The original PM2 required applications to be modified to call the special memory allocation routines isomalloc and isofree, which was a burden on developers and was not feasible when linking with a third-party library. In our runtime system, we extended this approach by overriding the system malloc/free routines to use the new isomalloc/free when it is called within a thread. Of course, malloc/free called from outside the threading context (e.g. by the communication layer of the runtime system) is still directed to the normal system version of malloc/free. This approach thus allows unmodified applications to use migratable thread memory for their heap data.

Since one thread's data is always allocated at the same addresses inside the isomalloc region, a thread can be migrated simply by copying all its data to the new processor — pointers within and between the thread's stack and heap need not be modified. Because we only allocate physical memory to local threads, the physical memory usage on each processor is modest. Unlike stack-copy threads, no data needs to be moved when switching threads, and multiple threads can run simultaneously, which allows the straightforward exploitation of SMP machines.

However, isomalloc stacks have the disadvantage that they consume virtual address space on *each* processor proportional to the total number of threads on *all* processors. 32-bit machines only have 4 GB of virtual address space, of which a substantial fraction is already occupied by the operating system, shared libraries, and other machinery. Even if the entir 32-bit address space were available for thread stacks, if each thread uses 1 megabytes, there would only be room for 4,096 threads. 64-bit machines, by contrast, normally have terabytes of virtual memory space available, and so never suffer from this problem.

Unfortunately, machines today continue to be constructed with high processor counts using the simplest, cheapest parts available, which today are 32-bit microprocessors. Large 32-bit x86 Linux clusters are commonplace; and the Blue Gene/L machine built by IBM for LLNL consists of 128K processors, each of which is a 32-bit PowerPC derivative. On this kind of machine, virtual address space usage quickly becomes a significant impediment to the use of isomalloc-based migratable threads. This motivated us to design a new approach to reduce the usage of virtual address space which is described next.

3.4.3 Memory Aliasing Stacks

As a complement to the isomalloc approach, we have designed and implemented an alternative implementation that uses much less virtual address space, and so can scale on large 32-bit machines. The idea is simple: accelerate stack-copying threads by simulating the copy using the virtual memory hardware. Like stack-copying threads, in this approach all stacks are executed from the same address. However, to switch in a new thread, we simply map the new thread's stack onto the pages at the stack address by calling `mmap`, rather than actually copying the stack data. That is, each thread's data is stored in separate pages of physical memory. To run a thread, we first map the thread's data into the common virtual address space with a memory mapping call as shown in Figure 3, then execute the thread.

The advantage of this approach is that only one stack-size of virtual address space is used, which allows this technique to be used even on machines with very limited address

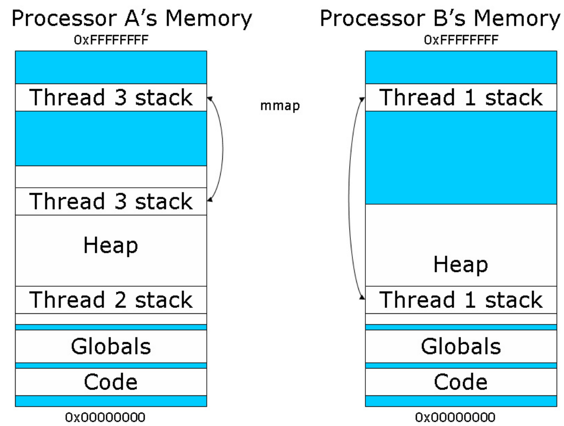

Figure 3. Memory-aliasing stacks

space. However, since each context switch involves an extra `mmap` system call, the performance of memory aliasing stacks is not as good as that of the isomalloc-based threads. However, because no data is actually copied, the performance of memory aliasing stacks is much better than the stack-copying threads. Like the stack copying threads, this technique has the same disadvantage that only one thread is allowed to be active in each address space. Section 4.2 compares the performance of these three thread migration techniques.

3.4.4 Thread Migration Portability

Table 1 illustrates the portability of our implementation of each thread migration technique on various platforms. Stack copying threads are portable to many platforms including even Windows. The only restriction on stack copy threads is the requirement that the system stack frames on all processors begin at the same base address. In practice, however, our implementation of stack copying threads is based on QuickThreads which has not been ported to the IA64. By contrast, isomalloc and memory aliasing stack thread migration mechanisms work on all machines except for those where the *mmap* system call is unavailable, for example Blue Gene/L and Windows. Windows supports virtual memory with the *MapViewOfFileEx* call, and so could be supported with only a small amount of effort. Blue Gene/L does not have mmap, but we have shown our scheme for memory aliasing can be supported by adding a small extension to the BlueGene/L microkernel to allow user processes to remap their heap data over the stack location.

Thread	X86	IA64	Opteron	Mac OS X	IBM SP	SUN	Alpha	BG/L	Windows
Stack Copy	Yes	Maybe	Yes	Maybe	Yes	Yes	Yes	Maybe	Yes
Isomalloc	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Maybe
Memory Alias	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Maybe	Maybe

Table 1. Portability of current implementations of migratable threads. "Yes" means we have implemented this technique. "Maybe" indicates there are no theoretical problems, but no existing implementation. "No" indicates the technique is impossible on this machine.

4 Performance

We have examined the practical performance of these methods for implementing migratable flows of control on a variety of real machines. The tests were carried out with both benchmark tests such as NAS Parallel Benchmark Multi-Zone version [18] and real-world application like a BigSim parallel simulator [43].

4.1 Number of Flows

We measured the context-switching performance of four different implementations of flows of control.

- Processes, created using fork() and yielding using sched_yield(). This is an imperfect benchmark, because some operating systems seem to ignore sched_yield() when called repeatedly, resulting in an artificially low measurement of the context switching time.

- Pthreads, created using pthread_create() and yielding using sched_yield().

- Cth (Converse Threads) [23], our implementations of user-level threads created using CthCreate() and scheduled using CthYield(). We used the non-migratable version of these threads.

- AMPI (Adaptive MPI) [16, 15] [2] user-level threads created by the AMPI runtime and scheduled using the AMPI routine MPI_Yield(). These are migratable threads, implemented using the isomalloc stack allocation approach based on the Cth threads, although no migrations actually occur.

We ran our experiments on a variety of machines. We report context switch times as the time per flow of control per context switch.

[2] An implementation of MPI that runs each MPI process in an AMPI thread.

- Linux, on a typical x86 laptop, with a 1.6 GHz Pentium M running Linux 2.4.25/glibc 2.3.3 (Red Hat 9). Context switch time is shown in Figure 4.

- Mac OS X, on Turing cluster at University of Illinois, each node has 2GHz G5 processors and 4 GB of RAM. Context switch time is shown in Figure 5.

- Sun Solaris, with a 700MHz SunBlade 1000 workstation running Solaris 9. Context switch time is shown in Figure 6.

- IBM SP, on the production machine cu.ncsa.uiuc.edu, with one 1.3GHz Power4 "Regatta" node running A/IX 5.1. Context switch time is shown in Figure 7.

- HP/Compaq Alpha, on the production machine lemieux.psc.edu, with one 1 GHz ES45 AlphaServer node running Tru64 Unix. Context switch time is shown in Figure 8.

Our experiments have shown there is a wide variation in the limitations and performance of these methods on different machines. In general, the user-level threads (Cth) on most of these machines have the fastest context switch time except on IBM SP and Alpha machines. On these machines except IBM SP, the context switch time of the user-level threads tends to increase slowly as the number of flows increases.

Table 2 illustrates approximate practical limitations on. stock systems. It shows the approximate maximum number of processes a user can create on a processor and the maximum number of threads a user can create in a process. As we can see, an unmodified Linux Red Hat 9 machine can spawn less than 256 pthreads in one process; while the per-user process limit on our IBM SP was only 100 processes. Both of these limitations can be extended with only a small amount of system administrator effort, but this effort is likely beyond the reach of a typical parallel user. In general, processes and kernel threads were limited to a few thousand, with only the Alpha allowing more than 5000 threads at a time and IA-64 without such limitation. By contrast, we could create tens of thousands of user-level threads easily on all platforms.

Flow of control	Limiting Factor	Linux	Sun	IBM SP	Alpha	Mac OS	IA-64
Process	ulimit/kernel	8000	25000	100	1000	500	50000+
Kernel Threads	kernel	250	3000	2000	90000+	7000	30000+
User-level Threads	memory	90000+	90000+	15000	90000+	90000+	50000+

Table 2. Approximate practical limitations (on stock systems) for various methods to implement flow of control.

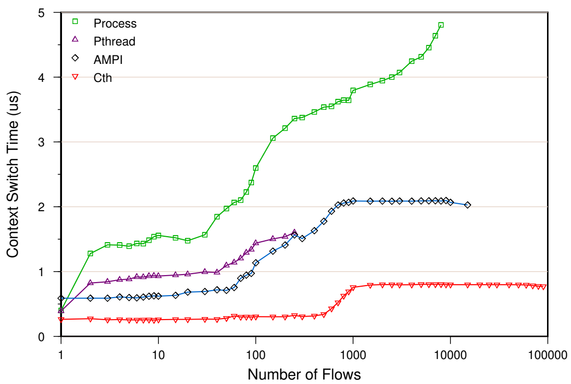

Figure 4. Context switching time vs. number of flows on a x86 Linux machine.

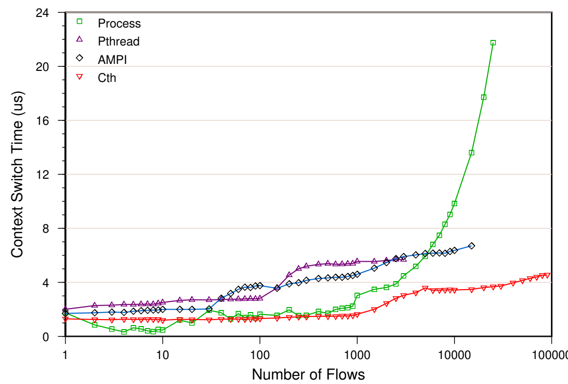

Figure 6. Context switching time vs. number of flows on a Sun Solaris machine.

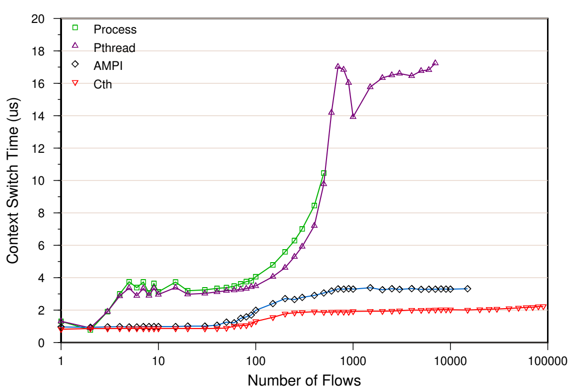

Figure 5. Context switching time vs. number of flows on a Mac Apple G5 machine.

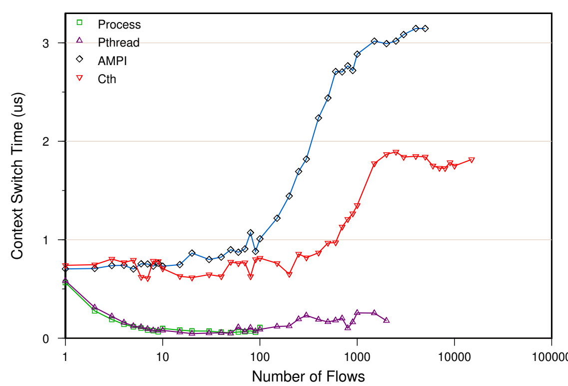

Figure 7. Context switching time vs. number of flows on an IBM SP machine. This is a 16-way SMP node. We believe the low times for processes and threads are due to the OS ignoring our repeated sched_yield() calls.

4.2 Stack Size

We also examined the effect the stack size has on the migratable thread context switching time. We examined three migratable thread methods, as described in detail in Section 3.4: stack copying threads, isomalloc threads, and our new memory aliasing threads. As our results in Figure 9 show, stack copying becomes unusably slow if more than 20KB of stack data is used. Isomalloc threads are the fastest overall, and show no dependence on the amount of stack

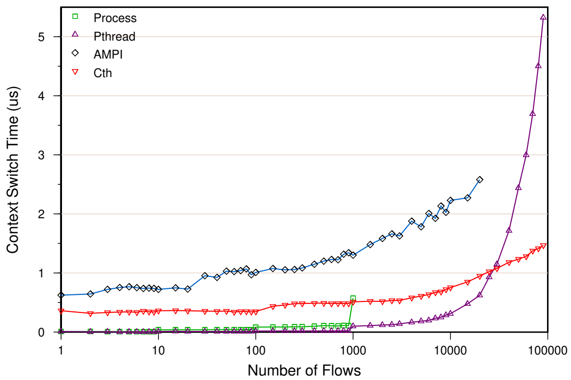

Figure 8. Context switching time vs. number of flows on Alpha machine. This is a 4-way SMP node. Again, process and thread switching numbers may be unrealistically low.

data in use. Memory aliasing stacks took about 4us per context switch, which is substantially slower than isomalloc threads, and the time grows with larger stacks although very slowly. Nevertheless, memory aliasing stacks can provide the small virtual memory footprint of stack copying, but (especially for larger stacks) with much faster context switching. It is suitable for applications with very large number of threads running on parallel machines with only 32-bit address space (e.g. BlueGene/L), where isomalloc threads may quickly run out of virtual address space.

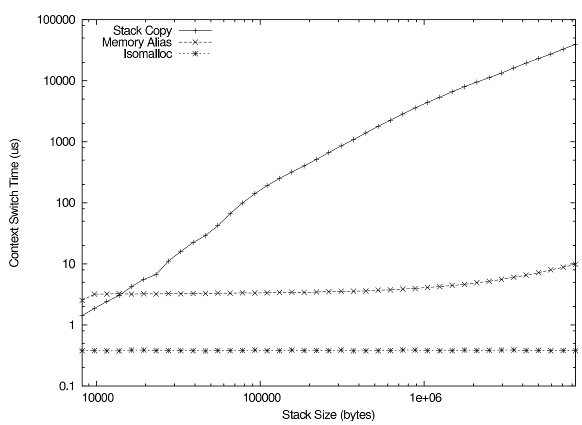

Figure 9. Context switching time vs. stack size on a x86 Linux machine. Various amounts of stack space from 8KB to 8MB were consumed using alloca().

4.3 Minimal Context Switching

We have determined a lower bound on the number of instructions required to explicitly context switch two user-level threads, as initiated by a subroutine call to a library's switch routine. Because the switch routine is a subroutine, this means only the saved registers defined in the architecture's subroutine calling convention need actually be saved and restored — scratch registers will automatically be saved and restored by the compiler, just like at any subroutine call.

This observation makes it possible to write extremely efficient user-level thread switch routines, particularly for today's popular x86 and x86-64 CPUs.[3] Figure 10 shows the minimum correct thread swap routines for 32 and 64-bit x86 CPUs. Note that on x86, the floating point registers must be empty before making a subroutine call, so the compiler will already save and restore floating point registers when needed.

```
swap32:#(old,new)
mov 4(%esp),%eax
mov 8(%esp),%ecx
# Save registers
push %ebp
push %ebx
push %esi
push %edi
# Save old stack
mov %esp,(%eax)
# Load new stack
mov (%ecx),%esp
# Restore regs
pop %edi
pop %esi
pop %ebx
pop %ebp
ret
```

(a)

```
swap64:#(old,new)
# Save registers
push %rdi
push %rbp
push %rbx
push %r12
push %r13
push %r14
push %r15
#Save old stack
mov %rsp,(%rdi)
#Load new stack
mov (%rsi),%rsp
# Restore regs
pop %r15
pop %r14
pop %r13
pop %r12
pop %rbx
pop %rbp
pop %rdi
ret
```

(b)

Figure 10. Minimal user-level thread context switching routines for 32-bit (a) and 64-bit (b) x86 CPUs. AT&T/GNU assembly syntax shown.

The subroutines in Figure 10 can swap user-level threads in 16ns (32-bit mode) and 18ns (64-bit mode) on a 2.2GHz Athlon64. Of course, a real thread library also requires a scheduling component to decide which thread to swap in when another thread suspends, but for many applications

[3] Together 32 and 64-bit x86 architectures provided 68.2% of the FLOPS in the 2005 Top500 list.

thread scheduling can be very simple — for example, a circular linked list of runnable threads.

Most user-level thread packages provide far worse performance than this for two reasons. First, real systems often include multiple layers of scheduling and prioritization which costs function-call overhead. Secondly, many user-level threads implementations save and restore far more state than is necessary, either through fear or ignorance. In particular, popular implementations of swapcontext and setjmp/longjmp (often used to implement user-level threading) save and restore all registers, including scratch registers. Worse, they often include system calls to save and restore the signal mask, even though very few scientific applications manipulate signals at runtime. If a user-level thread context switch involves even one system call, most of the speed advantage of user-level threads is lost. This is because a system call involves saving application registers when entering kernel space and restoring application registers when leaving kernel space, so the kernel could just as quickly perform a process switch by simply saving the registers of one process and restoring the registers of a different process!

4.4 Application — Parallel Simulator

BigSim [43, 44] is a parallel simulator developed on top of our Charm++ runtime system. It is capable of predicting performance of parallel applications on a massively parallel machine with petascale performance using an existing parallel machine with only hundreds of processors even before the target machine is built. Such simulation requires that one physical processor to simulate hundreds or even thousands of processors of the simulated machine, hence creating the scenarios of running multiple flows-of-control, one for simulating each target processor, on a simulating processor.

In a typical simulation, we simulated a Blue Gene like machine with 200,000 processors running a molecular dynamics (MD) simulation code. Running the test on 4 processors requires that each processor simulate 50,000 separate target processors, which clearly is not feasible on most machines using either processes or kernel threads. But by using user-level threads, we were able to simulate 50,000 target processors using 50,000 user-level threads on just one real processor.

Figure 11 illustrates the performance of BigSim using Cth, Converse user-level threads. The test was run on LeMieux which is 750 Quad AlphaServer ES45 node machine at Pittsburgh Supercomputing Center (PSC). Each node is a 4 processor SMP, with 4 Gbytes of memory. We measured the time taken to simulate one timestep of the MD simulation using 4 to 64 LeMieux processors. The figure demonstrates excellent scalability of the simulator in this test.

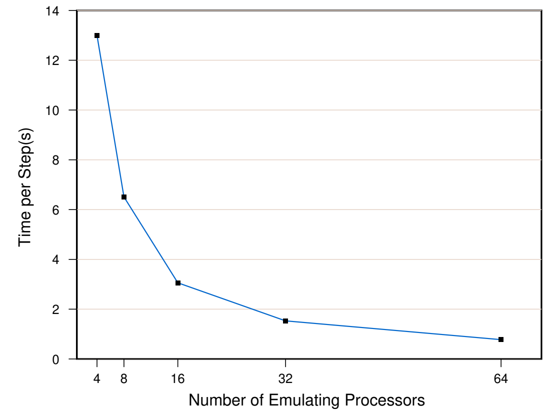

Figure 11. Simulation time per step using a total of 200,000 user-level threads

4.5 Thread Migration and Load Balancing

The "Multi-Zone" NAS Parallel Benchmark [18] is an extension to the well-known NPB suite. It involves solving the application benchmarks LU, BT and SP on various collections of loosely coupled discrete meshes. It is characterized by partitioning the problems on a coarse-grain level to expose more parallelism and to stress the need for load balancing. Among these tests, BT-MZ creates the most dramatic load imbalance, which is used in our test runs.

We ran the BT-MZ benchmark with Adaptive MPI and used thread migration for the load balancing. The migratable threads use the isomalloc and swap-global mechanisms (Section 3.4.2) to allow transparent thread migration without having to change any of the benchmark code. In order for load balancing to be effective, AMPI requires the number of AMPI migratable threads to be much larger than the actual number of processors, so that AMPI threads can migrate from overloaded processors to underloaded ones to improve load balance.

These tests were run on the Tungsten Xeon Linux cluster at NCSA. This cluster is based on Dell PowerEdge 1750 servers, each with two Intel Xeon 3.2 GHz processors, running Red Hat Linux and Myrinet interconnect network. Figure 12 shows the total execution time with various configurations of BT-MZ with vs. without load balancing. The x-axis represents each test case. For example, "A.8,4PE" indicates that the BT-MZ is compiled with `CLASS=A` and `NPROCS=8` running with 8 AMPI threads but on 4 actual processors. Note that same class (A, B, etc) problems have same problem size. Indeed, for all three class B tests on 8 processors (B.16,8BE, B.32,8PE and B.64,8PE), the execution times after load balancing are about the same, while there is a dramatic variation in execution times before load

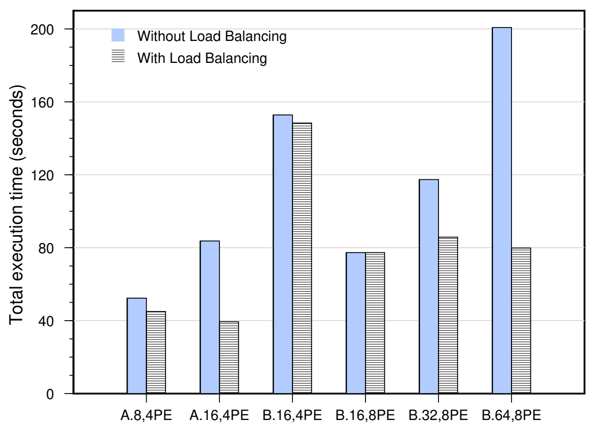

Figure 12. The NAS BT-MZ benchmark with and without thread migration for automatic parallel load balancing.

balancing. This benchmark demonstrates the effect of load balancing that is made feasible via thread migration.

5 Conclusion

We presented a study of four flow-of-control mechanisms that are widely used in parallel programming. These mechanisms are processes, kernel threads, user-level threads, and event-driven objects.

Through experiments, we illustrated the practical limitations of using these techniques on a variety of platforms. We further analyzed the performance of these techniques while varying parameters such as the number of threads and the amount of memory used by each thread. We demonstrated a wide variation in the performance of these flow-of-control mechanisms in context switching overhead. However, in general, user-level threads provide both flexible implementations and scalable performance. This makes user-level threads an attractive approach for programming parallel applications with a large number of flows of control.

We also examined approaches to support thread migration, which is particularly useful for load balancing. We described several methods for implementing migratable threads that can be used for load balancing large scale parallel applications. We have implemented these techniques — stack copying, isomalloc, and memory aliasing stacks — in the Charm++/Adaptive MPI runtime system [16]. We have also shown that these techniques can be used on a wide variety of platforms by a variety of real parallel applications.

References

[1] The GNU Portable Threads Library. http://www.gnu.org/software/pth.

[2] N. Adiga, G. Almasi, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. Bright, and J. An overview of the bluegene/l supercomputer, 2002.

[3] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *Transactions on Computer Systems*, 10(1):53–79, Feburary 1992.

[4] G. Antoniu, L. Bouge, and R. Namyst. An efficient and transparent thread migration scheme in the PM^2 runtime system. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) San Juan, Puerto Rico. Lecture Notes in Computer Science 1586*, pages 496–510. Springer-Verlag, April 1999.

[5] J. Appavoo, M. Auslander, M. Burtico, D. D. Silva, O. Krieger, M. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. K42: an open-source linux-compatible scalable operating system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

[6] A. Barak, S. Guday, and R. G. Wheeler. The mosix distributed operating system. In *LNCS 672*. Springer, 1993.

[7] K. Barker, A. Chernikov, N. Chrisochoides, and K. Pingali. A Load Balancing Framework for Adaptive and Asynchronous Applications. In *IEEE Transactions on Parallel and Distributed Systems*, volume 15, pages 183–192, 2003.

[8] K. J. Barker and N. P. Chrisochoides. An Evaluation of a Framework for the Dynamic Load Balancing of Highly Adaptive and Irregular Parallel Applications. In *Proceedings of SC 2003*, Phoenix, AZ, 2003.

[9] J. A. Booth. Balancing priorities and load for state space search on large parallel machines. Master's thesis, University of Illinois at Urbana-Champaign, 2003.

[10] R. K. Brunner and L. V. Kalé. Adapting to load on workstation clusters. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 106–112. IEEE Computer Society Press, February 1999.

[11] R. K. Brunner and L. V. Kalé. Handling application-induced load imbalance using parallel objects. In *Parallel and Distributed Computing for Symbolic and Irregular Applications*, pages 167–181. World Scientific Publishing, 2000.

[12] S. Chakravorty and L. V. Kale. A fault tolerant protocol for massively parallel machines. In *FTPDS Workshop for IPDPS 2004*. IEEE Press, 2004.

[13] F. Douglis and J. Ousterhout. Process migration in the sprite operating system. In *Proceedings of the 7th International Conference on Distributed Computer Systems*, pages 18–25, 1987.

[14] E. A. Hendriks. Bproc: The beowulf distributed process space. In *16th Annual ACM International Conference on Supercomputing*. ACM Press, 2002.

[15] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, pages 306–322, College Station, Texas, October 2003.

[16] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé. Performance evaluation of adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[17] J.Frey, T.Tannenbaum, I.Foster, M.Livny, and S.Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*, 2001.

[18] H. Jin and R. F. V. der Wijngaart. Performance characteristics of the multi-zone nas parallel benchmarks. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[19] R. Jyothi, O. S. Lawlor, and L. V. Kale. Debugging support for Charm++. In *PADTAD Workshop for IPDPS 2004*, page 294. IEEE Press, 2004.

[20] L. V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.

[21] L. V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[22] L. V. Kale and M. Bhandarkar. Structured Dagger: A Coordination Language for Message-Driven Programming. In *Proceedings of Second International Euro-Par Conference*, volume 1123-1124 of *Lecture Notes in Computer Science*, pages 646–653, September 1996.

[23] L. V. Kale, M. Bhandarkar, R. Brunner, and J. Yelon. Multiparadigm, Multilingual Interoperability: Experience with Converse. In *Proceedings of 2nd Workshop on Runtime Systems for Parallel Programming (RTSPP) Orlando, Florida - USA*, Lecture Notes in Computer Science, March 1998.

[24] L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[25] L. V. Kale and J. Yelon. Threads for Interoperable Parallel Programming. In *Proc. 9th Conference on Languages and Compilers for Parallel Computers*, San Jose, California, August 1996.

[26] D. Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington Department of Computer Science and Engineering, May 1993.

[27] D. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1997.

[28] O. S. Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. *Concurrency and Computation: Practice and Experience*, 15:371–393, 2003.

[29] M. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the unix kernel. In *Usenix Conference Proceedings*, pages 283–290, January 1992.

[30] D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

[31] J. Mukherjee and S. Varadarajan. Weaves: A framework for reconfigurable programming. *International Journal of Parallel Programming*, 33(2-3):279–305, 2005.

[32] J. C. Phillips, R. Brunner, A. Shinozaki, M. Bhandarkar, N. Krawetz, L. Kalé, R. D. Skeel, and K. Schulten. Avoiding algorithmic obfuscation in a message-driven parallel MD code. In *Computational Molecular Dynamics: Challenges, Methods, Ideas*, volume 4 of *Lecture Notes in Computational Science and Engineering*, pages 472–482. Springer-Verlag, November 1998.

[33] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.

[34] Sayantan Chakravorty, Celso Mendes and L. V. Kale. Proactive fault tolerance in large systems. In *HPCRI Workshop in conjunction with HPCA 2005*, 2005.

[35] W. W. Shu and L. V. Kalé. A dynamic load balancing strategy for the Chare Kernel system. In *Proceedings of Supercomputing '89*, pages 389–398, November 1989.

[36] D. S. T. G. Mattson and S. T. Wheat. A teraflop supercomputer in 1996: the asci tflops system. In *Proceedings of the International Parallel Processing Symposium*, 1996.

[37] S. Tatham. Coroutines in c, 2005. http://www.chiark.greenend.org.uk/s̃gtatham/coroutines.html.

[38] N. Williams. An implementation of scheduler activations on the netbsd operating system, 2002.

[39] T. Wilmarth and L. V. Kalé. Pose: Getting over grainsize in parallel discrete event simulation. In *2004 International Conference on Parallel Processing*, pages 12–19, August 2004.

[40] T. L. Wilmarth, G. Zheng, E. J. Bohm, Y. Mehta, N. Choudhury, P. Jagadishprasad, and L. V. Kale. Performance prediction using simulation of large-scale interconnection networks in pose. In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation*, pages 109–118, 2005.

[41] G. Zheng. *Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[42] G. Zheng, C. Huang, and L. V. Kalé. Performance evaluation of automatic checkpoint-based fault tolerance for ampi and charm++. *ACM SIGOPS Operating Systems Review: Operating and Runtime Systems for High-end Computing Systems*, 40(2), April 2006.

[43] G. Zheng, G. Kakulapati, and L. V. Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, New Mexico, April 2004.

[44] G. Zheng, A. K. Singla, J. M. Unger, and L. V. Kalé. A parallel-object programming model for petaflops machines and blue gene/cyclops. In *NSF Next Generation Systems Program Workshop, 16th International Parallel and Distributed Processing Symposium(IPDPS)*, Fort Lauderdale, FL, April 2002.