# Pthreads or Processes: Which is Better for Implementing Global Address Space languages?

by Jason Duell

## Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

**Committee:**

Professor Katherine Yelick
Research Advisor

(Date)

* * * * * * *

Professor Susan Graham
Second Reader

(Date)

# 1 Introduction

Global Address Space (GAS) languages—such as Unified Parallel C (UPC), Titanium (a Java dialect), and Co-Array Fortran—are a promising set of parallel programming languages [13][16][21]. One defining feature of GAS languages is that they provide the programmer with a single, shared global address space, thus allowing a shared memory-style of programming that is generally considered more user-friendly than message-passing programming. Another defining feature is that (unlike some earlier shared memory programming environments, such as DSM systems) they provide programmers with language-level awareness of the locality of each piece of memory within this global address space, so that code can be optimized to use local memory whenever possible on systems where memory access time across the global address space is non-uniform.

A great deal of the research on GAS languages has been concerned with how to optimize accesses to the slower, 'remote' portion of the shared address space. This concern has been especially well-founded in cluster environments, where a single shared address space is not physically provided by the memory system, and must instead be provided within the language runtime by using a network, which is typically at least several orders of magnitude slower in latency than regular memory accesses (and also often incurs significant software overhead per network data transfer). Various solutions and optimizations have been explored, ranging from compiler optimizations that transform sets of fine-grained accesses to smaller sets of larger, more network-efficient (and possibly also pre-fetched) bulk transfers, to software caching systems that do similar aggregation within the language runtime, to modifications to the programming environment (such as providing non-blocking put/get operations) which allow programmers themselves to aggregate network transfers and mitigate the effects of higher latencies via both manual pre-fetching and overlap of communication with unrelated computation or communication [2][6][7][14][23].

The research described by this report focuses instead on how best to provide shared memory for GAS languages *within* shared memory multi-processor systems, whether it be a standalone SMP machine, or SMP nodes that are part of a cluster. At a high level, the answer to this question is obvious: when multiple processors physically share memory, one ought to implement the GAS shared address space among them using the physically shared address space of the memory system. This allows accessing a variable that has locality to a thread running on a different CPU to be as simple as issuing a regular load or store instruction, just as for one's own data[1]. Alternatives, such as using

---

[1] This is in the absence of any additional operations that may be needed to enforce synchronization or language consistency rules. In practice, most UPC programs overwhelmingly use 'relaxed' puts/gets, which allows them to be implemented as simple memory put/get operations on a shared memory multiprocessor. "Strict" operation may occasionally be needed for synchronization, and in this case atomic operations and/or memory flushes, etc., are typically also needed.

the same network API that is used to access memory on remote machines, are unlikely to provide the minimal instruction cost, low latency, and high bandwidth of CPU puts/gets[2].

But while using shared memory is a given, how to implement it is not, particularly for a language implementation—such as Berkeley UPC—that seeks to be portable across the wide range of existing platforms in use in scientific computing. There are two obvious solutions. The first is to run multiple threads within a single process, using a threading library (such as the nearly ubiquitous, POSIX-standard pthreads library). The other is to create a separate process for each logical GAS thread of control, and use a mechanism such as System V shared memory or a shared memory map (mmap) to establish shared memory among them. This report describes how the Berkeley UPC project has dealt with this choice, first selecting to implement a pthreaded runtime, and now, for this report, developing an experimental runtime using shared memory among processes to explore the differences and tradeoffs between the two approaches.

Why is the choice of how to implement shared memory important, particularly if remote accesses are orders of magnitude more expensive? There are several reasons. First, although local accesses are much faster than remote one in GAS languages, they are also the common case (at least for well-programmed codes). Any differences in a shared memory implementation that result in intra-node performance differences are likely to have a disproportionate effect on overall application performance.

Secondly, as computer architectures increasingly move towards SMP machines as the norm, even for commodity desktops and laptops, there is increased interest into what is the correct programming model for such machines. As some prominent architectures (most notably the AMD Opteron) move towards a SMP interconnect model that relies on a network between processors, rather than a single memory bus, non-uniform memory access times within computers are becoming the norm, rather than an exotic behavior found only on certain high-end supercomputing machines. Given GAS languages' built-in awareness of NUMA architectures and control over memory layout, it is possible that they may prove a superior model for SMP programming than common models in use today (such as OpenMP) which implicitly assume a uniform memory model. If GAS languages are to compete in this space, however, they need to be as well-tuned as established paradigms like OpenMP, and this includes making sure that the most

---

[2] It is possible that bulk put/get operations might be faster when using a network API. On a system where the network interface has its own processor attached to the memory bus, for instance, it may be able for it to perform a memcpy-like memory transfer while freeing the original CPU to perform other work. But this would only work for bulk transfers that were large enough to amortize the cost of setting up the network interface transfer, and benefits would only be realized if the host CPU's immediately following instructions were not dependent on the completion of the transfer. It is thus not a method that one would want to use generally, although it might be worth investigating to optimize certain special cases.

performant method is used to set up computation over multiple processors with shared memory.

Finally, it turns out that the threads versus processes with shared memory choice also involves tradeoffs with respect to interoperability with existing programming models (such as MPI) and scientific libraries, as much of this existing infrastructure is incompatible or difficult to use with pthreads. This is an important consideration in scientific computing (which has a long and continuing history of "dusty deck" codes that remain in use for many years after their creation), at least for a project like Berkeley UPC, which seeks to be not just an academic research vehicle, but also a widely used, production-ready programming environment.

Much of the material described in this report summarizes work that has occurred over many years in the Berkeley UPC project, and so it is worth noting which portions of this report are new contributions to the field. Three areas stand out here: 1) this report examines the tradeoffs between using pthreads and processes with shared memory in a overall fashion that has not previously been fully synthesized or documented; 2) the implementation of the Berkeley UPC runtime using processes with shared memory instead of pthreads was done entirely as work for this report; and 3) the performance evaluations comparing the pthreaded and process-based UPC runtimes are new, and show that the processes version is no slower than the pthreads version, and in some circumstances is significantly faster.

# 2 Pthreads versus Shared Memory for GAS languages: the design space

In many ways there is little fundamental difference between using processes with shared memory and using pthreads. Both provide the ability for multiple logical threads of control to co-exist on different CPUs, with each thread of control able to access some subset of the other threads' memory. At this level, there is no inherent reason why either method ought to be faster.

There are many minor differences, however, between processes with shared memory and pthreads, and these lead to subtle tradeoffs, making the choice between pthreads or processes a complicated decision. In this section we explore these differences, first discussing factors that favor a pthreads implementation, then those that favor processes with shared memory. The discussion here focuses on UPC, but many of the issues raised are highly relevant to other GAS language implementations and/or to GAS language designers.

## *2.1 Advantages of using Pthreads*

Table 1 summarizes the advantages of using pthreads over using processes with shared memory for UPC. We then discuss each factor in detail.

| Table 1: Advantages of using Pthreads | | |
|---|---|---|
| | **Pthreads** | **Processes with Shared memory** |
| **Portable setup of large shared region** | Yes | No |
| **Ability to share all memory** | Yes | No |
| **Speed of provided synchronization** | Medium (mutexes) | Often slow (semaphores) |
| **Number of network endpoints** | 1 per node | 1 per CPU |

### 2.1.1 Portable setup of large shared memory region

One practical requirement for any performant GAS language implementation is that the various GAS threads on a single node be able to physically share a large amount of memory among them. One can, of course, implement the illusion of an arbitrarily large physically shared memory space with a small shared region, by using the small shared area as a "bounce buffer" area. In this case, puts turn into messages that are stored in the shared area, for the destination thread to eventually read and copy into its address space. This strategy, however, loses much of the benefits that the GAS model promises. Unnecessary copies of data need to be made, and while the one-sided communication model can still be preserved at the programming interface, the actual runtime is forced into a two-sided, send-receive model which both consumes processing time for sends/receipts, and causes unpredictable latency times (a message is not received until the target thread gets around to looking at its message queue). This is not a recipe for high-performance, and so in practice a GAS implementation should avoid such buffering entirely, or at least confine it to be the uncommon case.

Ideally, then, the language runtime should be able to arrange it so that all of the memory that a GAS language designates as "shared" is implemented as actual shared physical memory. The amount that is needed ranges from a delineated subset of the address space (such as in UPC or Co-Array Fortran) to the entire user-addressable address space (for Titanium). While the former languages do not make any guarantees to users about the amount of shared data that will be available, and the amount needed by users will vary by application, in practice a large fraction of the memory space should be sharable, in the hundreds of megabytes to gigabytes range on current machines.

Threading libraries hold a clear advantage over processes with shared memory in this area. Since threads co-exist within a single process space, each has full access to the full memory space of the process. Since the POSIX-standard pthreads API is nearly ubiquitous on supercomputing systems (at least those with multiprocessor nodes), pthread_create() provides a very simple portable way to get a completely shared address space among the GAS language threads on a compute node.

Setting up shared memory among processes, on the other hand, has been plagued on many supercomputing systems by an unfortunate conjunction of the idiosyncrasies of the

Unix APIs for obtaining shared memory, and the fact that most parallel job launchers do not give GAS language runtimes control over process creation.

There are three standard ways to set up shared memory among processes on Unix:

1) Call mmap() with the MAP_SHARED and MAP_ANONYMOUS flags, then fork() (note: on some AT&T-derived systems, the same effect is achieved by passing a file handle to /dev/zero to mmap, then forking). This sets up a shared region for the parent and any children created with fork(), and is the most portable way to create large amounts of shared memory on Unix. Unfortunately, it also requires that one have control over the parentage of the processes that will share memory (it is not possible to use this method to set up shared memory among unrelated processes, or among related processes once fork() has already been called). Most parallel job launchers on supercomputing systems, however, do not provide such control over parentage—by the time the GAS runtime is invoked, all of the processes requested on a node have usually already been created, and if one tries to fork() more, the network will typically fail (each process is generally launched with the information—often in environment variables—needed to setup one network endpoint: if a forked child tries to initialize the network with the same endpoint information, failure results). It is still possible to create a shared region in this case with mmap using MAP_SHARED and a regular file on the file system, but this is not a portably well-performing solution: the operating system may choose to sync any changes to the memory to the file at any time (some man pages list intervals of every 30 or 60 seconds), causing significant amounts of entirely superfluous I/O to occur. A number of BSD variants provide a MAP_NOSYNC flag to mmap which tells the operating system to never perform these memory syncs unless msync() is explicitly called, but this is not portable. In sum, mmap() is not a portable way to create shared memory for processes in today's supercomputing environments.
2) Use System V shared memory. This is one of the oldest Unix interfaces for creating shared memory among processes. While the interface is somewhat awkward to use (in particular, memory segments are not automatically deleted when the last process using them exits), it is fairly ubiquitous, and can be used to create shared memory among unrelated processes. However, early implementations of System V shared memory were resource-intensive: the shared memory segment was always pinned in physical memory (unlike mmapped address spaces, which are typically not backed by physical pages until/unless they are touched, and are also swappable). As a result, fairly low limits were imposed on both the size of System V shared memory segments, and how many that were permitted to exist system-wide. While some operating systems have increased these limits over time (notably AIX, which recently increased maximum shared segment size to 64 gigabytes), others have not. Linux even today allows segments of only 32 MB by default, and many other operating systems (Tru64, Solaris 2.x, HP-UX) have similarly tiny limits. Increasing these limits is usually possible, but only if one has root permission. As a result, the installation instructions for professional databases (such as Oracle) and other systems that require large amount of process-based shared memory (including that of the Aggregate Remote Memory Copy Interface, or ARMCI, a one-sided network API

that is similar to GASNet [15]) are littered with recipes on how to increase the System V shared memory limits on various operating systems. While this is not a problem for machines which are administered by the potential user of a GAS language, for large, centrally administered systems (such as DOE supercomputing sites) requiring root-capability changes to a system can be a significant barrier to adoption, particularly for users who are merely interested in trying out a GAS language.

3 ) POSIX Shared Memory. The POSIX 1003.1b Real Time Extension Specification introduced a updated version of the System V shared memory interface for creating shared memory among unrelated processes. Somewhat cleaner and easier to use than System V shared memory, it also is typically implemented to allow shared segments that are as large as those provided by mmap. Is it thus the preferred interface of the three Unix APIs for creating shared memory, at least for the purposes of a GAS runtime. However, despite being issued in 1993 (several years before the pthreads POSIX specification), the standard has been slow to catch on among Unix vendors. It was not widely available on Linux systems at the time the Berkeley UPC runtime was being designed (it required both Linux 2.4 and glibc 2.1.1 or greater), and is still not available on some major operating systems (including AIX). It thus did not provide the silver bullet of portability that it promised to offer, though the situation has improved in recent years.

### 2.1.2 Ability to share all memory

As mentioned above, the pthreads API offers each thread it creates a full view of the entire address space of the process. The various methods for creating shared memory among processes, on the other hand, cannot provide this. Any memory that has already been mapped into the process (such as the stack and the program's static variables) cannot be included in such a shared region. For GAS languages (such as Titanium) that have a "shared everything" model (i.e. any data whatsoever can potentially be the target of a one-sided memory get/put), this limitation is a potentially major disadvantage: either the stack and/or global data segments would need to be moved into the shared region, or any puts/gets to stack or global /static variables would need to be implemented using bounce buffers (as described above), which impede performance. For UPC and Co-Array Fortran, which have a clearly segregated shared data region, this is less of an issue.

### 2.1.3 Speed of provided synchronization

Pthreads and the shared memory APIs for processes also differ in the primitives that they provide for synchronization among CPUs. Pthreads provides 'mutexes,' which are generally moderate in speed (involving at least a function call overhead), while processes often have only Unix 'semaphores,' which are commonly implemented as more expensive system calls. The pthreads API provides a version of mutexes that can also be used in shared memory among processes, but it is an optional feature that is not found on all operating systems.

It is not clear how important this speed difference is for GAS languages, however. The most commonly used synchronization constructs in GAS language programming are barriers, and in a supercomputing environment (where a job commonly has dedicated

access to the CPUs it is running on) these can usually best be implemented within an SMP as spin-locks (which are not provided by any Unix standard, but can be implemented by hand from atomic operations). Certain other constructs, such as UPC locks, map more naturally to mutexes/semaphores, but these too can be implemented more quickly using atomics in shared memory. Finally, in a cluster environment, any differences among local synchronization methods are likely to be trivial compared with the much larger cost of performing network synchronization.

### 2.1.4 Number of network endpoints

One of the key differences between pthreads and processes is that pthreads share the same set of file descriptors, including any network connections/handles/endpoints (the terminology and actual number of file handles used—if any—varies among network APIs, so I will use the term 'endpoint' to capture the general concept). Processes, on the other hand, will each have their own network endpoint. In effect this results in a difference of 1 network endpoint per cluster node for pthreads, versus 1 per CPU on the node for processes. This difference has a number of tradeoffs that are likely to vary across different network architectures and APIs, and while it cannot be definitively said at this point that the end result favors pthreads, there are reasons to suspect that it may.

One important consequence of pthreads sharing a single network endpoint is that it allows a GAS application to use a network with a smaller number of addressable nodes. At the network API level, all of the pthreads in a remote process show up as a single network endpoint. This can provide speedups and/or memory savings for network APIs that use n-squared or other superlinear algorithms in their implementation. For instance, it is common for messaging APIs (including Active Messages, which is used within GASNet) to reserve a small buffer region for each remote peer to write messages into. This is an n-squared usage of memory, but it has typically been manageable on past systems. This may be changing as supercomputing systems get increasingly large machine/CPU counts. For example, on a machine currently being built at the University of Texas, with of 3,288 16-core nodes (for 52,608 processors total [19]), having each process in a full-system job reserve even just a 4K buffer for each peer process would result in over 3 GB of buffer space needed per node! Using pthreads, by contrast, would reduce the number of buffer sets per node by 16, and also reduce the number of remote peers by 16, requiring only 1/256$^{th}$ of the size (a much more manageable 13 MB). Similar considerations apply to the amount of internal state (and/or hardware resources) that a network API may use per endpoint. The Infiniband network used by the Texas machine also has a limit of 64K connections per host: while this is fine for pthreaded jobs (each host would need 3,287 remote connections), using processes would exceed this limit (16 processes per node, each of which would need over 52K connections), effectively limiting individual job size to less than a tenth of the processors, unless a (more complex and presumably slower) non-fully-connected approach were used for communication.

Another result of pthreads sharing a network endpoint is that any of the pthreads, when polling the network, can handle message traffic targeted at any of them, including even

replying to requests that were logically for one of the other threads. This ought to lower the average latency time for handling requests from other nodes.

A more ambiguous, but still significant difference is that some network APIs (notably IBM's LAPI, and also our implementation over Infiniband) create a separate pthread for each network endpoint that is created (generally this is to do polling and otherwise guarantee network progress). This means that a pthreaded GAS runtime creates only one such pthread, while a process-based one creates one per CPU. The end result of this on performance is not clear-cut: on the one hand, the presence of more threads on the system could increase the amount of context-switching, lowering performance. On the other hand, having a dedicated thread per process could potentially result in lower amount of cache eviction when a context switch is made (the user thread and its dedicated communication thread would likely be operating on the same data).

One possible negative side-effect of sharing a single network endpoint is that some network APIs are not re-entrant, and thus a pthreaded GAS runtime needs to do its own locking to ensure that only one thread enters the network API at a time. While concurrent access to network resources always needs to be handled at some level (in the hardware if nothing else), even when processes are used, doing this in software is likely to be slower and involve memory flushes, etc, that may not be strictly necessary.

## 2.2 Advantages of using Processes + Shared Memory

We now describe a number of factors that make using processes with shared memory more attractive than using pthreads. These are summarized in Table 2.

**Table 2: Advantages of using Processes + Shared Memory**

| | **Pthreads** | **Processes with Shared Memory** |
|---|---|---|
| **Thread-local data detection and conversion needed** | Yes | No |
| **Implementation complexity** | More complex | Less complex |
| **Use of existing scientific libraries** | Only if reentrant | Yes |
| **MPI interoperability** | Awkward | Straightforward |
| **Scheduling of CPUs** | Sometimes complex? | Simple |

Most of these "advantages" are a result of various complications or limitations that result when pthreads are used to implement UPC, and so this section might just as easily be called "disadvantages of using pthreads."

### 2.2.1 No thread-local data detection/conversion needed

An inherent difference between using multiple processes versus a single, pthreaded process is that in the latter case, only a single copy of any static variables (including, in C, both global variables and variables declared as "static" within functions) in the program exists, and is shared among the threads. In other words, when "int foo" is declared at global scope in a program, under pthreads, only one "foo" exists, and is

shared among the pthreads. When processes are used, by contrast, each process gets its own copy of "foo." The fact that pthreads share global variables is a problem when sharing is not what is needed. In particular, UPC, as an extension of C, allows the programmer to specify whether a given file-scope or static variable is 'shared' or not: if not, each UPC thread in the application is guaranteed to have its own, completely private copy of the variable. Thus "int foo", in UPC, requires that each UPC thread have its own copy of "foo." Processes support this automatically, but in a pthreads environment it requires converting the single copy of each such variable into N copies, one per pthread. While the GCC compiler provides a special '__thread' attribute that does this automatically (and some other compilers have adopted it), this is not a portable solution. Berkeley UPC thus needed to hand-roll a portable solution to this issue. The details of this implementation are described in a later section, but the general point to be made is that the need to perform this conversion in the first place required a significant amount of work in both the runtime and the UPC-to-C translator. Also, while we took great pains to ensure that our solution could logically be implemented with minimal or no runtime cost, the actual code that we needed to generate was fairly "perverse" (no human would ever program in such a manner), using many casts and address operations that exposed us to the risk that back-end C compilers would not compile it as efficiently as they would straightforward accesses to regular variables (which is the style that a process-based implementation can use). The work done for this report includes some benchmark comparisons that aim to capture this cost, and the results of these are given in the performance results section.

An additional difficulty in the UPC case is that the language environment must distinguish between "UPC"global variables (that must be made thread-local), and "C" global variables (that must not). For example, if a UPC program #includes <stdio.h> and then calls "fprintf(stderr, …)," the UPC runtime must not assume that multiple copies of "stderr" exist, because the C standard library only provides a single instance per process. Resolving this problem requires distinguishing whether a variables is declared within a "C" or "UPC" context . This is somewhat similar to issues that C++ encounters when referencing C variables, in which C++ code must be informed that a variable is "extern C" in order to reference it properly. However, since there is no equivalent in UPC to "extern C" (most likely this is because it is not needed when processes are used, and the UPC language designers did not anticipate the possibility of pthreaded implementations), we trained Berkeley UPC to infer it by whether a header file appears to be UPC or C code (i.e., whether it contains UPC keywords, and/or #includes other headers that do). This solution generally works well, but required significant work to implement.

### 2.2.2 Implementation complexity

Implementing the pthreaded version of the UPC runtime proved significantly more complex than the processes version. In large part this was due to the thread-local data detection and conversion needed, as just described. In addition, many internal data structures within the runtime needed to be duplicated for each pthread, and the code to access them #ifdef'd to handle the threaded versus non-threaded cases (we did not wish to impose any overheads on non-threaded instances of the runtime—such as those that would run a single UPC thread on each cluster node—and so made sure that any logic

needed to support pthreads that might impact performance was defined away when pthreads were not used).

In a backhanded manner, this additional complexity was actually one of the reasons we decided to begin with a pthreads implementation, rather than processes and shared memory. We figured that it would be easier to design in the pthreads support in the initial implementation, rather than try to shoehorn it in later.

### 2.2.3 Use of existing C libraries

One major point of distinction between pthreads and processes where pthreads comes up short is the ability of GAS programs to take advantage of existing scientific libraries. Many if not most scientific programming libraries are not re-entrant, and thus cannot be used by more than a single pthread at a time. This renders such libraries useless if the library is used in an inner loop, as all parallelism is lost.

These reentrancy issues can also arise when an existing serial application is converted to UPC. Some parts of the application may prove to be reentrant, but this is not always obvious to the programmer, and considerable confusion may result. We later describe an issue of this type that we encountered during our benchmarking efforts.

### 2.2.4 MPI Interoperability

A similar problem for pthreaded implementations is interoperability with the MPI (Message Passing Interface) library, which is nearly ubiquitous in parallel environments today. Almost all MPI libraries are not thread-safe, and so cannot work with a pthreaded runtime. Even if thread-safety is available, and/or the pthreaded runtime synchronizes accesses to the MPI library (or, as might be necessary, arranges to access MPI from the same thread for all calls), the interface between MPI and UPC would still be quite awkward, as MPI rank is assigned per process, while UPC thread number is assigned per pthread. Programs would not be able to send MPI messages to a given UPC thread (only to sets of threads), and the programmer would need to be careful to only enter MPI collectives from exactly one pthread in each process, etc.

Using processes, the re-entrancy issue is avoided, and MPI and UPC thread/rank can be easily arranged to be identical. [Note: this does not solve all issues with MPI interoperability—there are still important network issues that need to be dealt with to avoid deadlocking in the worst case. But these issues are more easily solved than the reentrancy issue, and the rank/thread mismatch is fundamental to the API when pthreads are used].

### 2.2.5 Scheduling of CPUs

One last possible reason why processes with shared memory might have an advantage over pthreads for GAS implementations is that pthreads are often scheduled differently than processes, and this might result in disadvantages for the pthreaded case. While there is no inherent reason for pthreads to be scheduled any differently than processes, on many operating systems, process and pthreads are scheduled by different code. This may or may not result in observable differences.

The POSIX specification for pthreads deliberately left many fundamental details up to the library writers. Depending on the implementation of the pthreads library, pthreads may

be scheduled by the operating system identically to the way processes are (this is the case on Linux, for instance), scheduled entirely within the pthreads library, with no operating system involvement (this does not appear to be common, as it limits all pthreads to run serially on a single CPU, but it is a valid implementation), or scheduled in some hybrid library-OS fashion (as on the Sun Solaris operating system, which has a complex system of mapping library-level pthreads to "lightweight processes" that exist as kernel entities). While we saw no reason to suspect that these differences would affect performance in the ideal case—when a job is the only program running on a node, and runs as many or fewer pthreads as CPUs—we were interested to see if pthreads were treated equally to processes on systems where contention for CPUs is present. Anecdotally, we did not see any evidence that they were treated any differently than processes on busy systems, but did not have time to design any tests to rigorously examine the issue.

# 3 The design decision: pthreads first, processes with shared memory later

As mentioned, the Berkeley UPC development team decided to go with pthreads rather than shared memory with processes for the initial implementation of Berkeley UPC. Why was this design choice made?

One major reason was that it seemed the most reliable route to a fully portable UPC implementation: it was not clear that we would be able to successfully set up processes with large amounts of shared memory on all of our systems of interest, without requiring installers to have root control.

Another reason was momentum from the Titanium language project at UC Berkeley. The GASNet networking layer used by Berkeley UPC was also used by Titanium, which, as a Java language dialect, naturally fit well into a pthreads implementation. The GASNet development team members (some of whom joined the Berkeley UPC effort) were thus already experienced with a pthreads implementation, and GASNet itself was already designed with support for pthreaded clients across a wide range of networks. It did not have any support for creating shared memory among processes, though, and so moving towards a process-with-shared-memory model would have required an extensive reworking of GASNet.

Finally, as described above, there were reasons to suspect that a pthreaded UPC implementation would achieve better network performance and/or scaling, but this was not clear (and might well vary by network). It thus seemed likely that we would wind up implementing both pthreads and processes-with-shared-memory implementations eventually. Pthreads already worked with GASNet. It also seemed likely to be easier to add a process version into an existing pthreads-based runtime interface than vice versa. For these reasons we decided to begin Berkeley UPC with a pthreads runtime implementation.

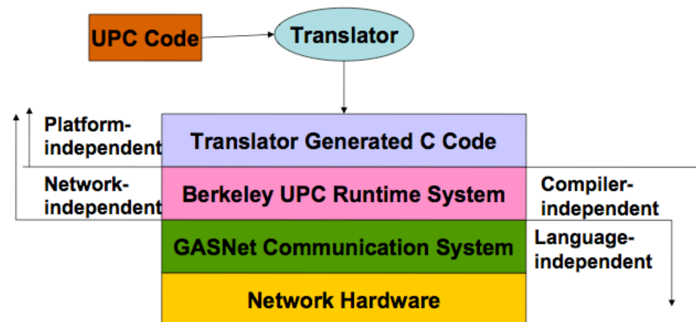
For this master's report, I have also implemented an experimental version of Berkeley UPC that uses processes with shared memory. This work was undertaken partly because it was recognized from the start that a process-based implementation would be needed to get MPI interoperability and support for scientific libraries. But it was also begun because our pthreads implementation had begun to acquire a reputation (of uncertain merit) of being slow. One user posted to our bug system that he was seeing significant

slowdowns in his application when running with one pthread (i.e. with the pthreads infrastructure used, but only one pthread) versus running with a single non-pthreaded process. Some of our own application writers were running their benchmarks on clusters of SMP using only a single, non-pthreaded process per node, as they found it gave them better performance and/or more reliable results, or because they found that their code did not work with pthreads because of re-entrancy issues. One easy way to find out whether the pthreads implementation deserved the bad reputation is was getting was to write a processes with shared memory implementation, so that parallel performance between the two could be compared.
The next section provides details of how both the pthreads and process-based runtime for Berkeley UPC were implemented. In each case, I wrote the initial implementation, with considerable design support from the rest of the Berkeley UPC team (the pthreaded implementation, which is our current production codebase, also has had many pieces of code added by others). I then proceed to show the results of some benchmarking comparisons between the two runtime versions.

# 4 The Software implementation

## *4.1 UPCR/GASNET Architecture*

The diagram below shows the basic architecture of the Berkeley UPC system.




The UPC-to-C translator (based on the open-source Open64 compiler [17]) takes in UPC code, and emits C code containing calls to the UPC runtime to implement language features specific to UPC (such as shared variable accesses, barriers, shared memory allocation, etc.). The UPC runtime, in turn, uses the API provided by GASNet to perform

any activity that involves network traffic. GASNet then calls then appropriate low-level network APIs that actually cause needed network activity.
All of the work described here took place in the UPC runtime and GASNet. The pthreads implementation was primarily at the UPC runtime layer (since GASNet already supported pthreaded clients, it needed few changes). The processes-with-shared-memory implementation, on the other hand, has consisted almost entirely of changes within GASNet, to support setting up shared memory segments and to provide a communication layer between the processes that share them.

## *4.2 The Pthreads implementation*

The first decision we made with respect to doing a pthreaded implementation was that we wanted as much as possible to maintain a single interface between the UPC-to-C translator and the UPC runtime. This would allow the translator to emit the same C code, regardless of whether pthreads or processes were being used, and encapsulate the logic needed to handle their differences entirely within the runtime. In this we were largely successful, primarily because we designed the UPC runtime interface as we designed the pthreads implementation. There were a number of subtle details in the implementation that resulted in unusual requirements in the runtime API (particularly with regard to initialization). It is quite unlikely that if we had started with a process-based implementation, we would have been able to shoehorn in a pthreads design in later without major changes to the interface. The recent addition of a process-based runtime, on the other hand, has not required any major changes to the API at all.

### 4.2.1 GASNet changes to support pthreads

GASNet's design model is based around providing a logical network endpoint to the layer above it. It was thus decided that GASNet would have nothing to do with launching pthreads, or handling communication among them, etc., other than to ensure that pthreads were safe to use by clients. GASNet thus provides a 'parallel' version for each of its network libraries, with the code modified to provide any locks and thread-local data needed to use the underlying network safely.

### 4.2.2 UPC runtime support for pthreads

The work needed within the UPC runtime to support pthreads was more considerable. Some of this work was to perform shared-memory versions of the functionality that GASNet already provides among network nodes: for instance, the UPC barrier function is implemented by first having all the pthreads within a process do a shared memory barrier, then have one call into GASNet to do the barrier across the network. Some work was also done in the shared pointer representation used by the runtime, in order to allow immediate accessing of shared data belonging to other pthreads via a simple pointer dereference, rather than a call to GASNet.
The largest part of the effort, however, was adding the thread-local data conversion of static unshared variables. The reasons why this conversion is necessary have already been discussed above, and so here I will only briefly describe the implementation, highlighting the parts that are relevant to performance (the full, gory details are available elsewhere [10]). The UPC runtime API was modified to require that initialization and accesses of static unshared variables were done through a set of macros, which when

pthreads are not used, expand into regular C initializations or accesses. When pthreads are used, however, 'upcc' (the user-visible program for compiling Berkeley UPC programs) arranges to declare a single large C-language 'struct' that contains a field for each static unshared variable declared within the program, and the macros for accessing the user's static variables wind up indexing the same-named field of the large struct. One such struct is created for each pthread, and a pointer to the struct is stored using the pthreads_setspecific() function, which is the pthreads standard mechanism for storing thread-specific data for a pthread. Since the corollary function for retrieving the pointer, pthread_getspecific(), is relatively costly, we went to some lengths to ensure that it is called infrequently: only at the beginning of every user function. Within the runtime, the pointer to the struct is passed along as an extra argument to any functions that need it, resulting only in an extra stack argument.
As a result, the pointer to the structure should generally be present, hopefully within a register. Accesses to the fields within the struct should theoretically be as simple and fast as an indexed load instruction (the base address of the struct would be one of the indexed load arguments, and the other would be the compile-time constant offset to the given field in it). However, we had some reasons to suspect that not all compilers would correctly optimize the code, for two reasons. First, due to C not having any rules about maintaining a uniform type namespace across compilation units ('typedef float foo' in one .c file can coexist with 'typedef char * foo' in another: the linker does not care), we had to discard the types of the user's static data when we created the large struct, and instead store everything as arrays of char (of the appropriate length and alignment, of course). It is up to the UPC-to-C compiler to cast the result of the macro call to the original type provided by the user. In a further wrinkle, the rules for C casting are fairly restrictive for scalars: for instance, the following code (which is analogous to what we would like to do to write to a float in the struct) is illegal:

```
struct bigstruct {
   char myfloat[4];    // assuming sizeof(float) is 4 bytes
   …
};
struct bigstruct *my_data = magic();
((float)(my_data->myfloat)) = 3.14159;     // error!
```

One cannot simply take an array of four char and cast it to a float, even if a float is the same size. The casting rules are quite loose for pointers, however, so changing the last line of code to the following is legal and achieves the same effect:

```
*(float *)(&my_data->myfloat) = 3.14159;
```

This is essentially the code that the back-end C compiler sees for every access of an unshared static variable in Berkeley UPC when pthreads are used. While there is no reason why an optimizing compiler could not generate code for this that is as efficient as simply referencing a global "float myfloat" variable, we had many reasons to fear performance degradation. First, we have seen some evidence that for many C compilers, simply taking the address of a variable (for instance to pass it to an inline function which

promptly dereferences the pointer, without doing any address arithmetic, etc.) causes significant loss of performance. We theorize that this may be that the compiler no longer thinks it can keep the value in a register. Secondly, we feared that casting between types might interfere with compilers' alias analysis and/or other optimization passes. Finally, it seemed quite unlikely that this sort of code construct would be in any compiler vendors' list of test cases to optimize, given that is it not the sort of code that human beings would write.

How much impact could such degradation have? This depends heavily on whether (and how much) thread-local data is used in a program. The cases we were most afraid of were programs where large static unshared arrays are declared at file scope, then used in inner computational loops. While this is not generally considered to be a good programming style, it is a somewhat popular idiom with Fortran programmers, and is also common in benchmark codes.

As mentioned earlier, another issue that had to be solved within the pthreads implementation was to determine which global variables in a program needed to be treated as thread-local (those declared in UPC source/header files), versus ones which needed to be treated as per-process variables (global variables within C files and/or libraries). Unfortunately for our purposes, UPC does not enforce using a different file extension than C does: UPC source files may have .c/.h extensions instead of .upc/.uph (to be precise, the UPC specification is silent on file extensions, and existing UPC implementations had chosen to allow .c/.h file extensions, making it impractical for Berkeley UPC to enforce a different convention). We thus ran into the problem of how to determine whether a file is UPC or C. The heuristic we came up with treats a file as UPC if it contains any UPC keywords (such as 'shared' or 'strict'), and/or #includes any header files which contain such keywords (it is possible for UPC code to #include C code, but not vice versa, as a C compiler will die when it sees UPC keywords). We implemented this heuristic by writing a custom lex-based parser, named "detect-upc", which traverses the output of the preprocessor, building up a logical tree of which files #included which (by tracing the "#line" directives emitted by the preprocessor), and noting which contain UPC keywords. Files #including files containing UPC constructs are then recursively marked as UPC by adding a special "#pragma" directive as the first line of the file, which we trained our UPC-to-C translator to recognize.

This relatively simple idea was complicated by the fact that certain vendors' C preprocessors behave differently. In particular, several omit #line directives when exiting a file unless they are needed to mark variable declarations, and this can make it impossible to determine post-facto which file #included which. On such systems we were forced to use GCC's preprocessor, which we trained to "look like" the vendor's preprocessor (i.e. the same set of predefined macros are in place, and so the preprocessor output should always be semantically identical to the vendor's preprocessor's). We also discovered that many preprocessors omit any mention of a file being #included if it expands into nothing but whitespace. This is commonly the case if a file is #included multiple times by different files (C programmers conventionally add an "#ifdef" around all code in a header file to avoid multiple declaration errors if the header is #included multiple times). This behavior occasionally caused files that should have been marked as UPC (because of files they #included) to be missed, since the #include of the UPC file was not visible in the preprocessor output. To fix this, we wound up having a summer

intern write another small program (upcppp: the "UPC pre-pre-processor") that is run before the preprocessor to determine which files #include which, providing a database that detect-upc can then use to mark all the appropriate files as UPC when UPC constructs are detected.

## 4.3 Implementing processes with shared memory

Modifying Berkeley UPC to use processes with shared memory on SMPs, instead of pthreads, mainly involved work within the GASNet layer. The network-less 'smp' version of the library was the initial target for the changes—this by itself is enough to support standalone SMP systems, and provides a testbed to see if thread-local data (or other facets of the pthreads implementation) cause performance degradation. Care was taken, however, to make as much of the work done in the 'smp' conduit reusable by the various networked versions of the GASNet library.

### 4.3.1 Startup logic altered to use mmap/fork

Since the 'smp' conduit runs on a single machine, we do not need to worry about parallel job launchers, and have complete control over how the processes in the UPC application are created. We thus chose to use the most reliably portable way to get large amounts of shared memory on Unix, namely mmap() with MAP_ANONYMOUS (or, on some systems, /dev/zero), followed by fork() calls to create the rest of the processes in the job. This was the least reusable part of the implementation, since this strategy for gaining shared memory will not work portably once we wish to run jobs with a network, for the variety of reasons discussed earlier in this paper. It does, however, make the 'smp' variant of GASNet quite portable.

### 4.3.2 Implementing Active Messages within shared memory

The largest part of the implementation effort was the next step, which was to write an implementation of Active Messages (AM) [22] that works within shared memory. GASNet relies on Active Messages to do any work that requires the logical equivalent of calling a function on another thread (as opposed to just reading/writing data in a one-sided manner, which does not inherently require the involvement of the other CPU). Even though most GASNet operations (such as all reads and writes) between processes that share memory can be implemented as simple memcpy() calls or pointer dereferences, a functional AM implementation is still needed for features like remote memory allocation.

To implement AM within shared memory, a lower-level 'sysvnet' layer was first written ('sysv'—short for "System V"—is the abbreviation we have chosen in GASNet for work related to processes with shared memory: the most obvious candidate—'shmem'—was already taken by a network API named 'SHMEM' used on some Cray and SGI systems). The goal of this layer was simply to support a set of fast message queues between processes sharing memory. Since the number of CPUs within an SMP is generally small, we used an algorithm where each process creates a separate set of receive buffers for each other process to write into. Though this consumes an N-squared amount of memory space, both N and the amount of memory needed per buffer are fairly small, while the benefit is gained that processes never need to contend or coordinate with any process other than their target to send a message. This also keeps the implementation fairly

simple, with only a few atomic operations and a single read/write flush needed in a send/receive transaction.

Once ‘sysvnet’ was available, implementing AM within shared memory became fairly simple. Two separate sysvnets are created (one for requests, and one for replies), and the logic for constructing, sending/receiving, and dispatching Active Messages was largely cut-and-pasted from one of the network-based implementations. Much of the work consisted of cutting out the network-specific logic and replacing it with sysvnet calls, plus changing the data structures used (which painstakingly stuffed the message header information into as few bits as possible, to save payload space) with more straightforward (and within an SMP, faster) accesses to 32 bit fields. Deadlock (which in sysvnet could occur if two peer processes fill each other’s inbound queues with requests, then block waiting for the other to reply, without checking their own queues while they wait) was fairly easy to avoid: all that was needed was to ensure that the sysvnet ‘poll’ function is called whenever a slot on the target’s queue cannot be acquired, and that only the ‘replies’ network is polled when this occurs during a reply. (Handling a new request in AM when one is already blocked waiting to be able to send a reply is counterproductive, as any requests handled might require replies, exacerbating the situation).

While this version of AM could have been a ‘one-off’ implementation used only in the pure shared memory (i.e., with no network) configuration, we wished to make sure that it could still be used when a network is also present. While Active Messages are not typically the bottleneck in a well-tuned GASNet layer, they can affect performance, and some properties of sysvnet (such as the fact that no copies of messages are made) make it likely to perform better than network-based AM within a node. This was achieved by renaming all of the AM functions for sysvnet with an ‘AMSYSV’ prefix, and encapsulating as much of the implementation as possible to avoid code duplication. (The one major implementation detail that was left to be implemented on a case-by-case basis was allocating the shared memory region needed for sysvnet). Some minor changes were also made to the GASNet API to allow the AMSYSV layer to interoperate with the opaque ‘token’ types that are used within GASNet, but implemented differently for each network type. Finally, the general GASNet function for polling the network was modified to also poll sysvnet when shared memory is being used, and a function was added to test whether a particular GASNet peer is accessible via shared memory or not.

Once the thorny issue of how to set up processes with shared memory is worked out for a given network and operating system, the rest of the work needed to support processes with shared memory in GASNet primarily consists of simply adding checks wherever AM calls are made, and diverting them to the AMSYSV layer if possible, and likewise, of converting get/put operations in the extended GASNet API to simple memcpy operations when shared memory is present.

Finally, while we do not anticipate ever wanting to support UPC programs that mix both the pthread and processes with shared memory models, we did not wish to preclude GASNet from supporting this type of configuration. Code was thus added to make sysvnet and AMSYSV safe for use within a multithreaded client. This consisted of adding a number of locks to the code, which are preprocessed into no-ops when pthreads are not used.

### 4.3.3 Minimal UPC level changes

One pleasant surprise from the implementation work was how well our initial design for the UPC runtime handled the transition from pthreads to processes with shared memory. Excluding changes to our build infrastructure, supporting processes with shared memory required changing less than a dozen lines of code (primarily to initialize a table of which nodes are to be considered 'local' and which 'remote'). While there is still some further work that could be done to optimize certain constructs (such as implementing barriers via atomics in shared memory instead of using Active Messages), this code will be common to both the processes with shared memory and pthreaded versions of the runtime.

# 5 Performance results

Because we had observed very baffling performance numbers in the past when benchmarking the UPC runtime on SMPs, we decided to begin our comparison of pthreads and processes with a very simple benchmark. We would take an embarrassingly parallel benchmark (no communication), using only regular C pointers in the inner loops (no overhead from UPC pointers), and see 1) how the program scaled; and 2) how pthreads compared with processes. Theoretically, performance should scale almost linearly, and there ought to be no difference between pthreads and processes, other than that perhaps caused by the effect of our thread-localizing of static data (described earlier). To try to capture this latter effect, we decided to make the benchmark modifiable so that it could be run with varying numbers of global variables used in the inner loop. If there were a performance cost, we would expect to see it grow as more global variables were placed in the inner loop.

## *5.1 Benchmark 1: Apex-map*

The first benchmark that we chose to use is one called Apex-Map, developed at Lawrence Berkeley National Lab [18]. It is a synthetic benchmark, which seeks to measure how computational performance is impacted by different memory access patterns. Three main parameters can be independently adjusted: memory size (M), spatial locality (L), and temporal locality (T). A large array of size M is allocated (in our tests, we used a fixed size of M = 16 million doubles per UPC thread). The benchmark then repeatedly hops to various offsets within this array, and performs a simple computation over a block of L continuous doubles. The choice of where to hop in the array is generated by a power function that takes the T parameter as input. When T=1, the offsets in the array are chosen with uniformly random access (i.e. any location in the array is equally likely to be the start of the computation). As T is lowered toward 0, the selection mechanism becomes increasingly non-uniform, and the likelihood that it will choose a location near the start of the array becomes increasingly high.

By running Apex-Map over a wide range of both the L and T parameters, one gets a picture of performance across a wide range of application types. For instance, as L and T both approach 1, the benchmark becomes very similar to the GUPS (Giga UPdates per Second) benchmark, in that it is accessing individual double values randomly throughout the memory (and performance is thus dominated by access time to main memory, since the uniformly random accesses defeat both caching and prefetching). As L grows (up to

64K in our tests) but T remains at or close to 1, the working set is too large to fit into cache, but caching and prefetching both operate to mitigate memory latency effects. Finally, with large L and low T, the algorithm almost always selects blocks that are near the beginning of the large array, resulting in the working set fitting into cache, and performance that is compute-bound and much closer to processor peak.

This range of memory/computation patterns made Apex-Map appealing to us, as it promised to provide the basis for both a simple, communication-free benchmark that could provide data on whether pthreads and processes perform differently on different systems, and/or determine whether our thread-local conversion of global data incurs a performance penalty.

We began by taking an existing version of Apex-Map and converted it to be an embarrassingly parallel UPC program, in which each UPC thread has its own M-sized array, and performs computation only on its own data. We then used #ifdefs to create several versions of the code, in which different numbers of the variables in the inner loop were made to be global variables, in order to attempt to measure the cost (if any) of thread-localizing.

## 5.2 Thread-local data in the Apex-Map kernel

The Apex-Map kernel logically consists of the following line of code, contained within a double-nested loop construct that modifies the 'j' and 'k' variables:

```
res +=  c*array[j + k];
```

where 'c' is a constant value, 'array' is a pointer to the UPC thread's M-sized large memory array; 'j' is the location in the array (selected by the power function) that marks the start of the vector to traverse; and 'k' iterates from 0 to L.

However, input from the Apex-Map developers (and our own experience running early versions of the benchmark) showed that most compilers did a suboptimal job on the above code, and that manually unrolling the loop four times yielded better performance. The actual code used thus looked like this:

```
res0 += c*array[pos0 + k];
res1 += c*array[pos1 + k];
res2 += c*array[pos2 + k];
res3 += c*array[pos3 + k];
```

I.e., the 'res' and 'j' values are separated out into four separate variables. Four versions of the code were run: 'ALL', in which all of the above variables were declared at file scope; 'TWO', in which 'array' and 'c' were declared as global, 'ONE' in which only 'array' was made global, and 'NONE', in which all variables were declared within the function. The number of distinct global variables and the total references to them are summarized in Table 3.

| Test configuration | # of global variables | # of references to global variables |
|---|---|---|
| **NONE** | 0 | 0 |
| **ONE** ('array') | 1 | 4 |
| **TWO** ('array', 'c') | 2 | 8 |
| **ALL** | 11 | 20 |

**Table 3: Global references in different configurations of Apex-map test.**

While the ALL case is unrealistic (it is very unlikely that a programmer would declare all the above variables at file scope), it provides a "worst case" scenario. The other cases are entirely possible in programs that declare one or more arrays, etc., at file scope, and then refer to them within the inner loop of an application.

### 5.2.1 Expected results

We were interested in two dimensions of the benchmark's performance. First, we wanted to see how processes and pthreads compared to each other in performance. Second, we wanted to observe the cost of thread-localization.

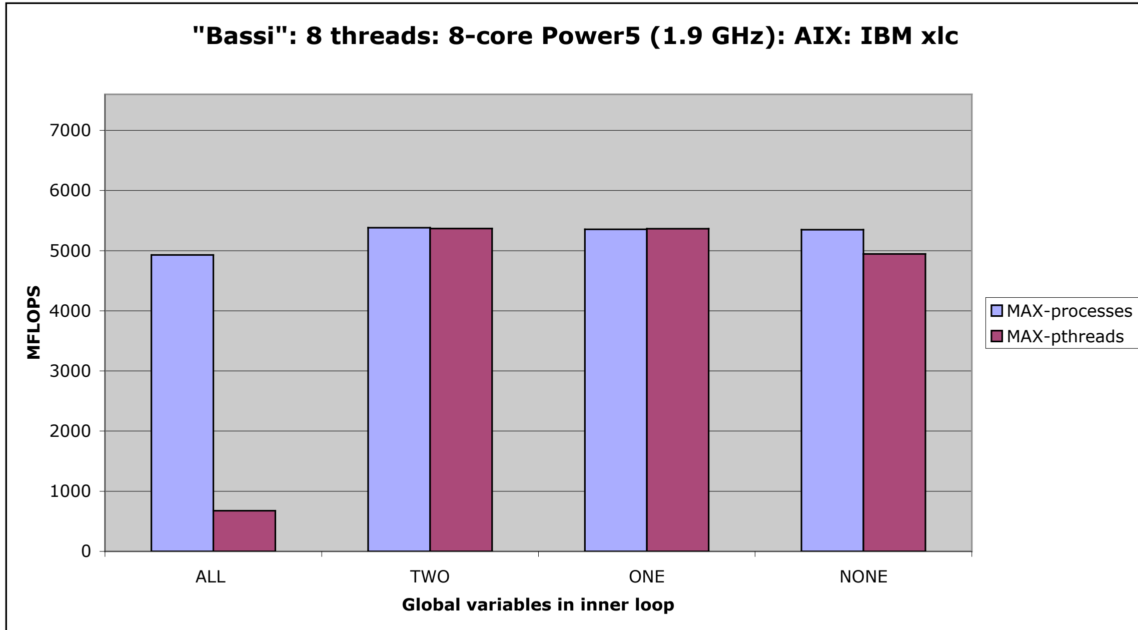
Serial C versions of Apex-Map generally achieve a large fraction of processor peak (as L reaches an optimum size—often different from machine to machine—and T approaches 0, so the data all fit into cache). Since our code was essentially the same, we expected similar performance, at least for runs with a single UPC thread. Since we consciously chose to write our version to be embarrassingly parallel, we also expected it to scale close to linearly—at least for the highest-performing tests, in which most memory accesses hit the cache (avoiding memory bus contention)—unless something (such as a pthreads-unfriendly scheduler) prevented efficient utilization of the processors.

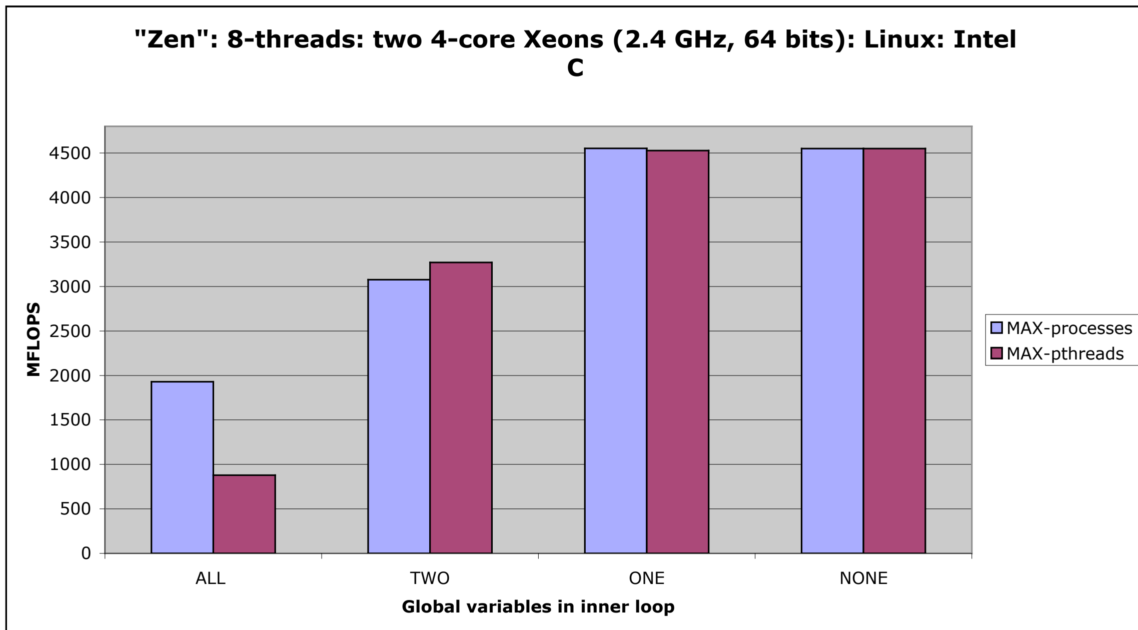
We were unsure whether thread-localization would cause performance degradation. However, if it did, we expected that it would manifest more in cases where more thread-local data was used.

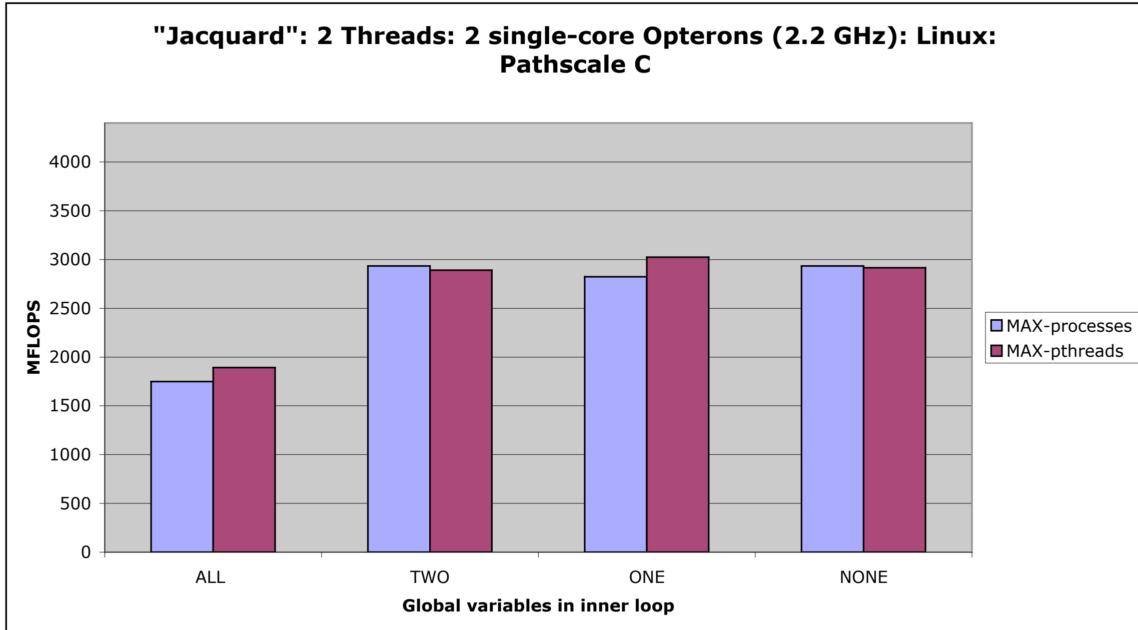
### 5.2.2 Actual results

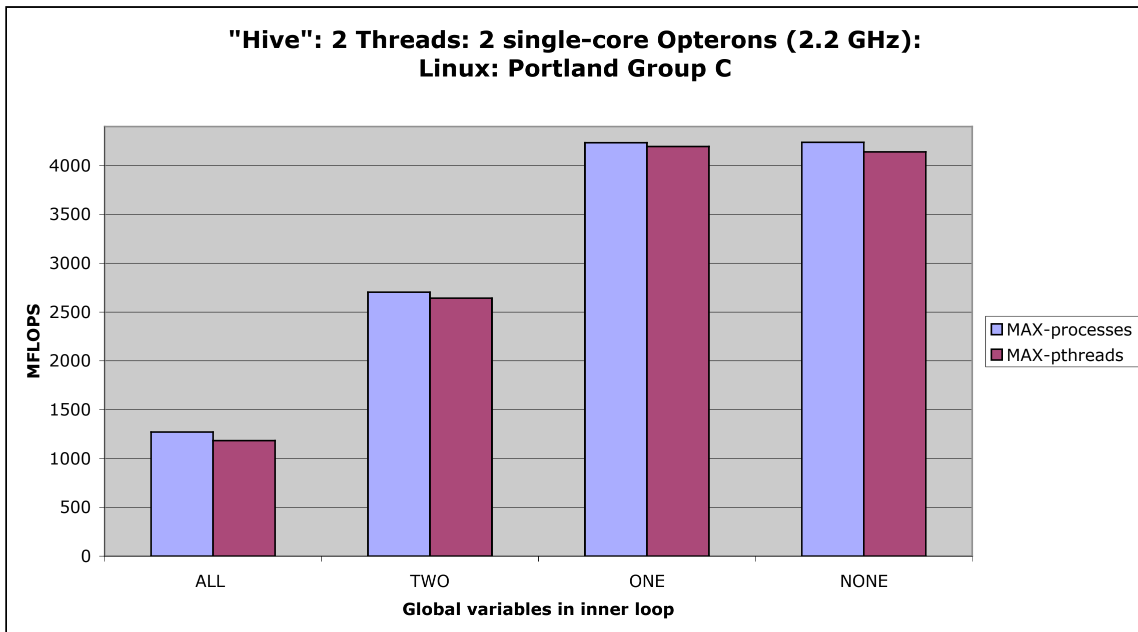
The tables below show the results from a number of SMP systems that are part of larger scientific clusters in production use. As it turns out, our code did scale linearly as UPC thread count increased, and so we only present the run with the largest thread count (equal to the number of processors on the system, with the exception of the Itanium system, on which were unable to get exclusive access to all 16 processors, and so we used 8).

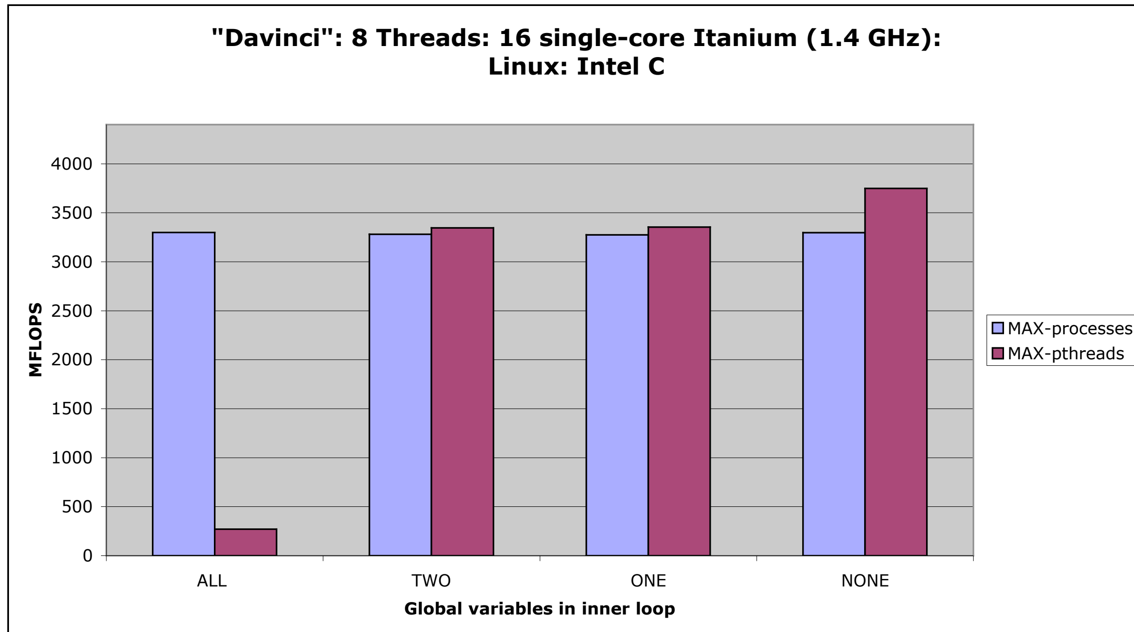
The bars represent the best Megaflop performance achieved by any of the Apex-Map runs (i.e. across the full range of L and T values tested). Megaflop rates are given per-CPU. In order to give a sense of the percentage of peak achieved, the top of the Y-axis in each chart is the theoretical peak Megaflop rate of the CPU.


"Bassi": 8 threads: 8-core Power5 (1.9 GHz): AIX: IBM xlc
MFLOPS
7000
6000
5000
4000
3000
2000
1000
0
ALL
TWO
ONE
NONE
Global variables in inner loop
MAX-processes
MAX-pthreads


"Zen": 8-threads: two 4-core Xeons (2.4 GHz, 64 bits): Linux: Intel C
MFLOPS
4500
4000
3500
3000
2500
2000
1500
1000
500
0
ALL
TWO
ONE
NONE
Global variables in inner loop
MAX-processes
MAX-pthreads


"Jacquard": 2 Threads: 2 single-core Opterons (2.2 GHz): Linux: Pathscale C
MFLOPS
4000
3500
3000
2500
2000
1500
1000
500
0
ALL
TWO
ONE
NONE
Global variables in inner loop
MAX-processes
MAX-pthreads


"Hive": 2 Threads: 2 single-core Opterons (2.2 GHz): Linux: Portland Group C
MFLOPS
4000
3500
3000
2500
2000
1500
1000
500
0
ALL
TWO
ONE
NONE
Global variables in inner loop
MAX-processes
MAX-pthreads




The first interesting observation to make about this data is that the "ALL" case is much slower for most of the configurations tested, regardless of whether pthreads or processes were used. This was something of a surprise to the author, as when processes were used we expected to see little or no performance difference as the amount of global data in the inner loop was varied. And indeed, two of the configurations (Power5/xlc and Itanium/Intel C) do show almost constant performance across this spectrum. The other systems, however, suffer significantly when all of the inner loop variables are global, and two show noticeable degradation even when two variables in the inner loop are global. While the author is not a compiler expert, consultation with the compiler developers in the Berkeley UPC team resulted in the possible explanation that many compilers do not make a large effort to optimize references to global data, due to the difficulty of doing alias analysis on variables with program-wide scope. In any case, the results suggest that application writers would be wise to avoid using global variables in computational kernels

The second interesting point about the data concerns the effect of thread-localization. As predicted, we can see a performance hit for using thread-localized data in some instances. What is remarkable, however, is that this degradation does not appear to have an effect that is linear with either the number of variables that are thread-localized, or the total number of thread-localized accesses, as given in Table 3. In every case, making all 11 variables global (resulting in 20 thread-local references, though a good compiler ought to eliminate some of these through common subexpression elimination) results in significant degradation (in some cases, the effect is catastrophic—the Itanium and Power 5 systems lose 90% or more of performance). But as soon as the number of global variables is only one or two (and the thread-localized references down to 4 or 8), most of the systems are performing just as well as when no thread-localization is used. This suggests that the cost of thread-localization may not be so much an inherent per-access cost, but rather a "lost optimizations" cost: too many thread-localized variables may cause compilers to "throw up their hands" and miss out on optimizations that are logically possible.

In many ways, however, the Apex-map results are promising for our pthreads implementation. There does not seem to be any inherent performance problems with the pthreads implementation, as it generally runs just as fast as the process-based version, and both scale linearly on an embarrassingly parallel problem. Even the most likely cause of performance problems, thread-local data, appears to be negligible or small except when exacerbated in an artificial fashion. To be more confident of this last point, however, we would want to explore the reasons why thread-localization causes degradation, perhaps using other micro-benchmarks and/or information about compiler internals.

## 5.3 Benchmark 2: NAS Parallel Benchmarks

We next moved on to a more complex (but also more realistic) set of codes: the NAS Parallel Benchmarks. These consist of five programs containing scientific kernels and pseudo-application logic from computational fluid dynamics applications [1][8][9]. They are some of the more commonly used scientific application benchmarks.

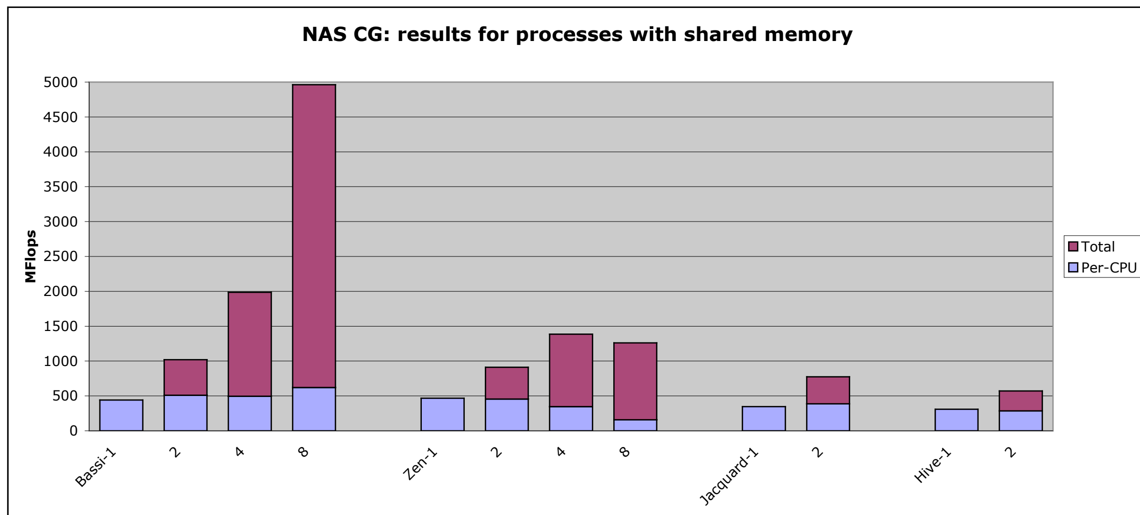
The version of the NAS Benchmarks used here is one that has been implemented in UPC by a team at George Washington University [11]. While they are not generally considered to be highly optimized implementations, our focus was less on absolute performance than on observing the relative performance of pthreads versus processes over a range of realistic (albeit somewhat simplified) application codes. The fact that the GWU codes contain all five of the NAS kernels thus made them the logical choice, despite the fact that more highly tuned implementations of particular NAS kernels exist (also, these tuned implementations have also focused on techniques—such as overlapping computation with communication—that are much more likely to improve performance in a networked rather than SMP environment) [2].

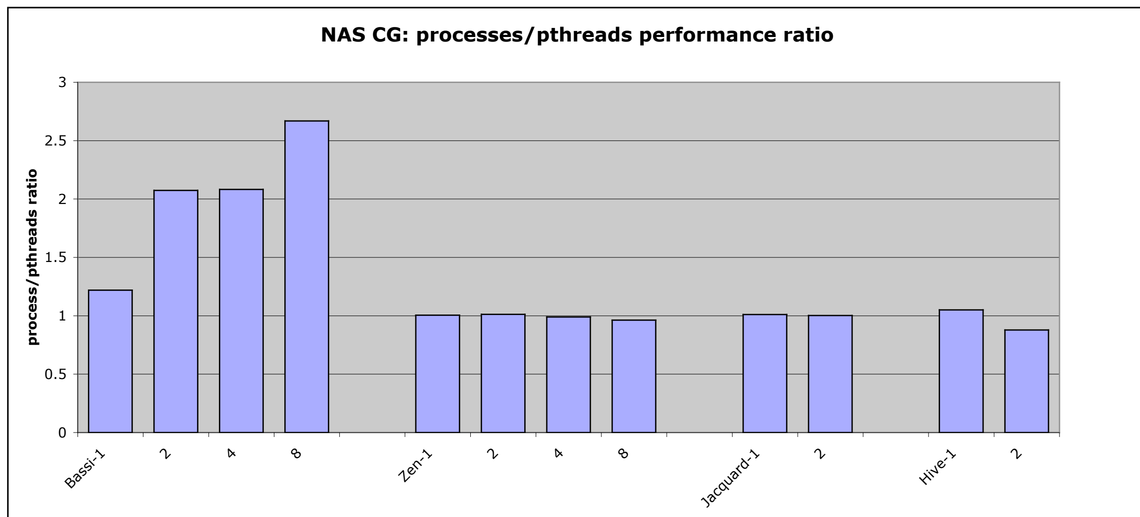
Compiling and running the GWU NAS code gave the author of this paper a taste for some of the subtle issues that can arise when a pthreaded UPC runtime is used, due to thread-local data. The GWU authors had begun their implementation from the code in the original MPI versions of the benchmarks. This included some C files with "timer_start" and "timer_stop" calls (for timing performance), which happen to use a global variable for the timer. The Makefile framework for the benchmarks used separate $UPCC and $CC variables for the UPC and C compilers, and compiled the timer code with the C compiler. As a result, a naïve "make" resulted in a pthreads version in which multiple pthreads stepped over each other's start/stop timing calls, resulting in confusing (and incorrect) timing results. The logical fix for this was to pass in CC=$UPCC to the "make" command, but this in turn caused another failure, as the C compiler was also used within the Makefile framework to build and run a diagnostic program, which died when it was not run with the "upcrun" UPC job launcher. The Makefile thus needed to be manually modified to make the application build and run correctly. (As a final insult, it turned out that another member of the Berkeley UPC team had already figured all of this out, and had inserted mechanisms into the Makefile framework for dealing with it which I had not noticed). While none of this was rocket science, it is also not the type of thing that endears a programming environment to users, and underlines some of the advantages of using processes with shared memory, which compiled and ran without any of these issues.

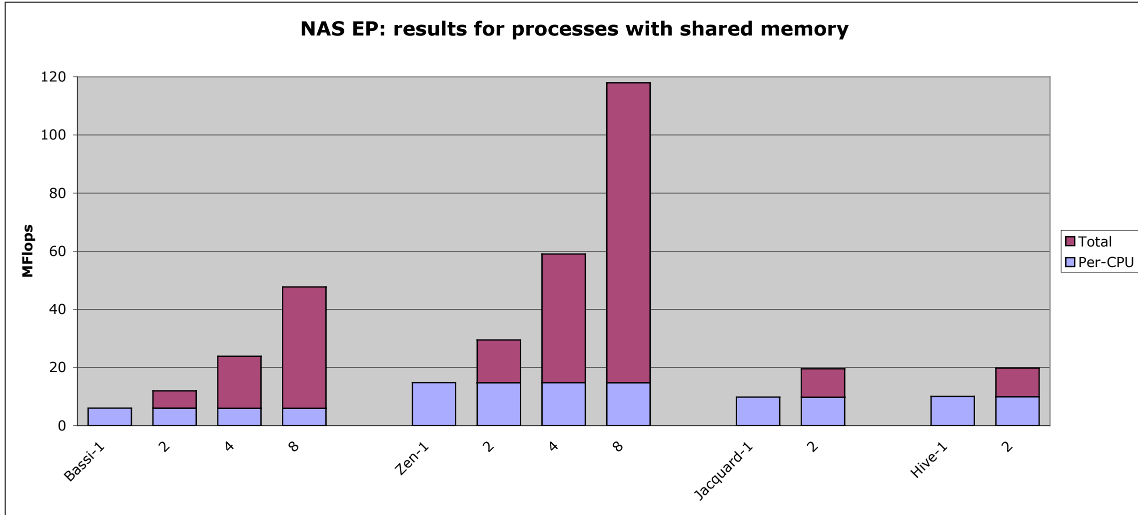
The five NAS benchmarks were run using the same systems and compilers as the Apex results, (except for the "davinci" system, which lacks a batch system and became extremely busy after we gathered our Apex results, making it impossible to get useful performance numbers). In order to mitigate for the effects of interference on some other systems (where we ran our code on the front-end nodes, which were sometimes in use by other programs), we performed between 10 and 20 runs (using CLASS=A, a relatively small problem size), and then averaged the top 20% of the results.

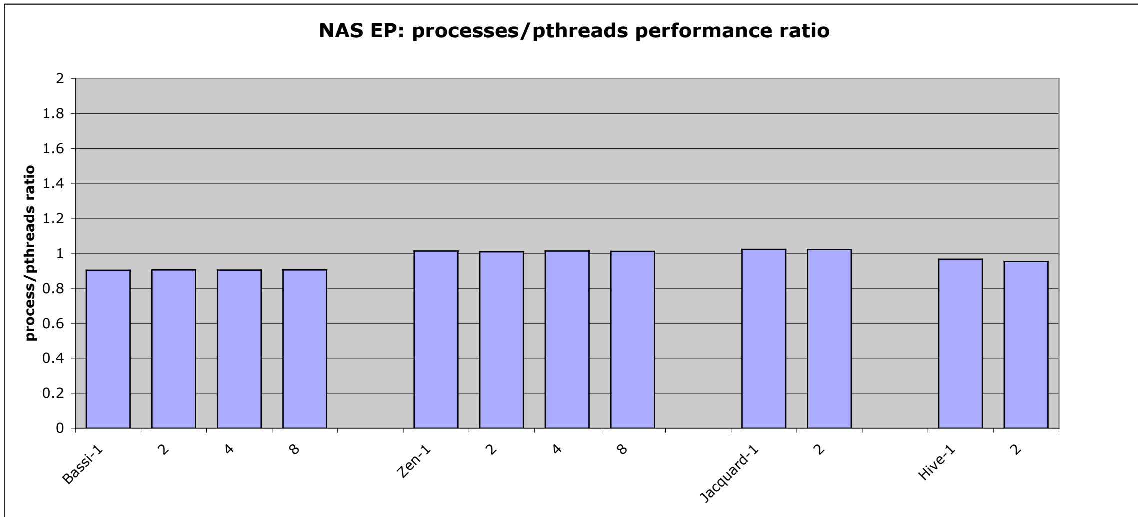
As with the Apex results, we did not have any theoretical reasons to expect major performance differences between the pthreads and processes versions, unless thread-local data caused performance degradation in some cases.

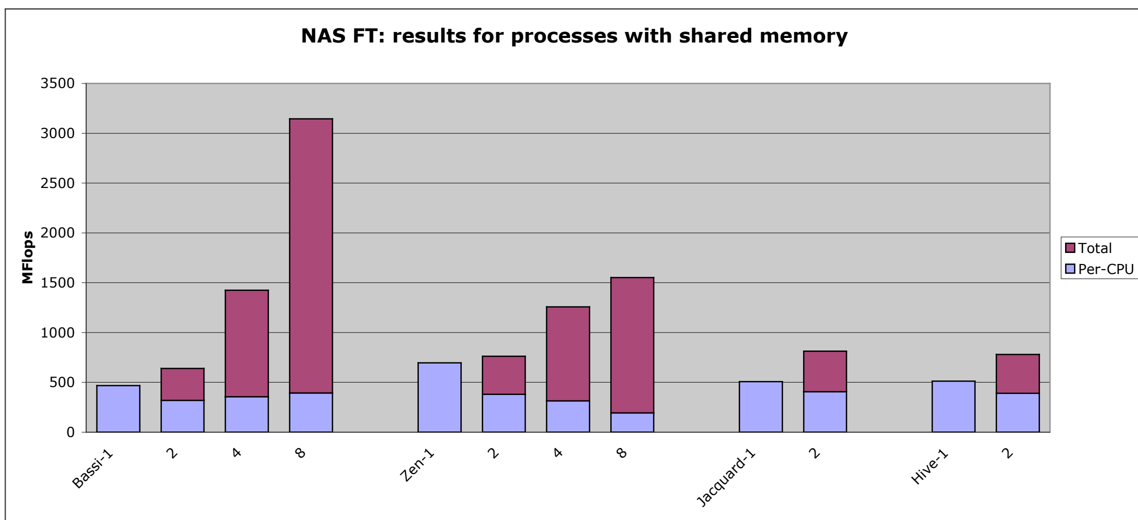
The results are shown below. Each NAS test has two charts: one which shows the per-CPU and total performance of the processes with shared memory implementation along the range of configurations and processor counts available, and another which shows the ratio of process/pthreads performance over that same range.

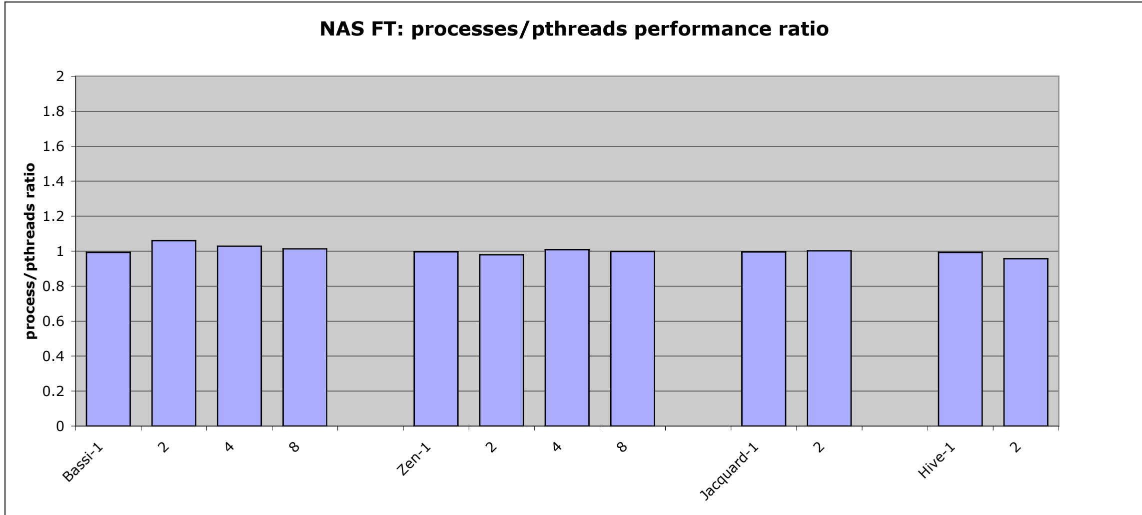


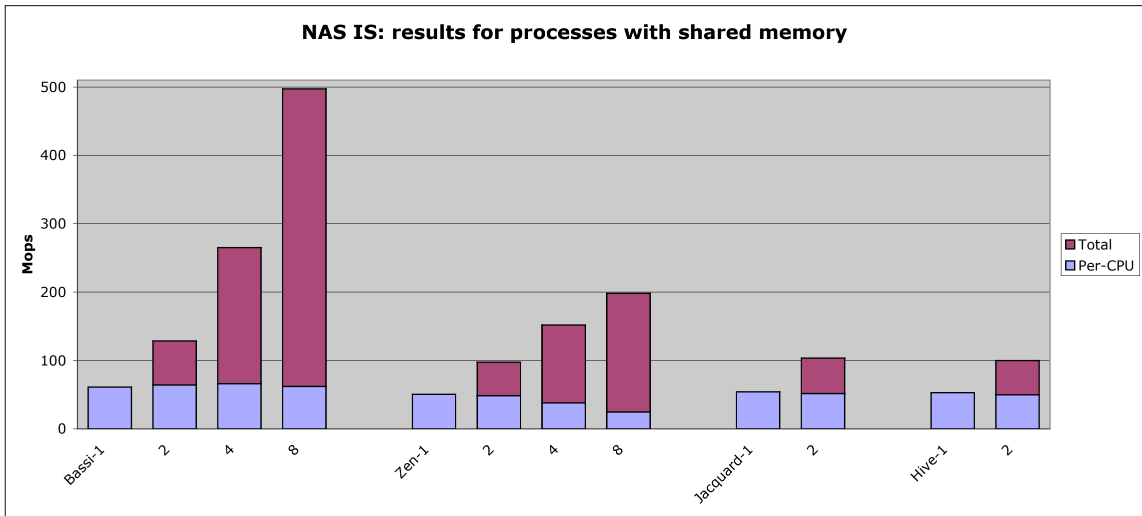



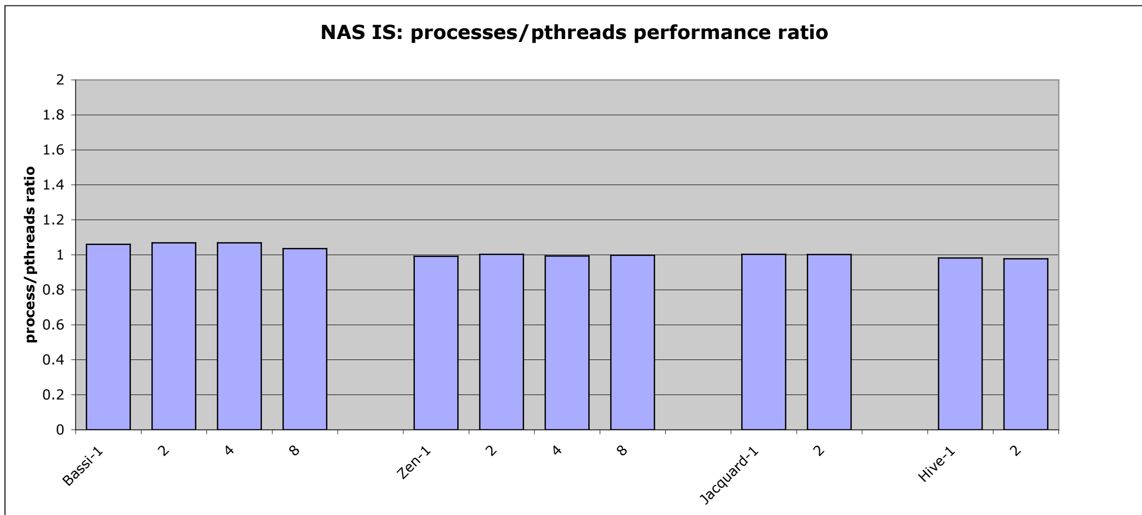

NAS EP: results for processes with shared memory
MFlops
120
100
80
60
40
20
0
Total
Per-CPU
Bassi-1
2
4
8
Zen-1
2
4
8
Jacquard-1
2
Hive-1
2

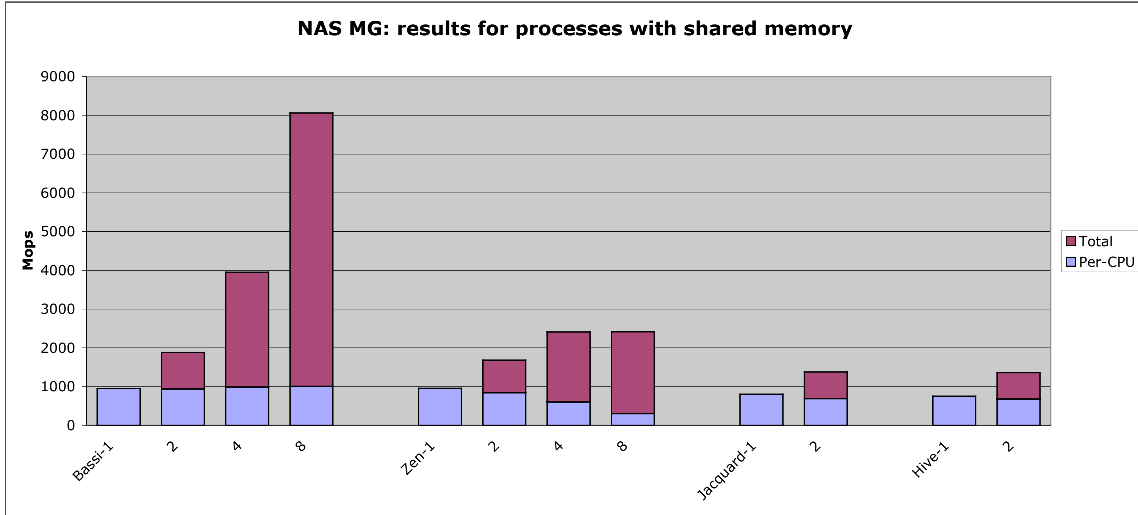

NAS EP: processes/pthreads performance ratio
process/pthreads ratio
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
Bassi-1
2
4
8
Zen-1
2
4
8
Jacquard-1
2
Hive-1
2

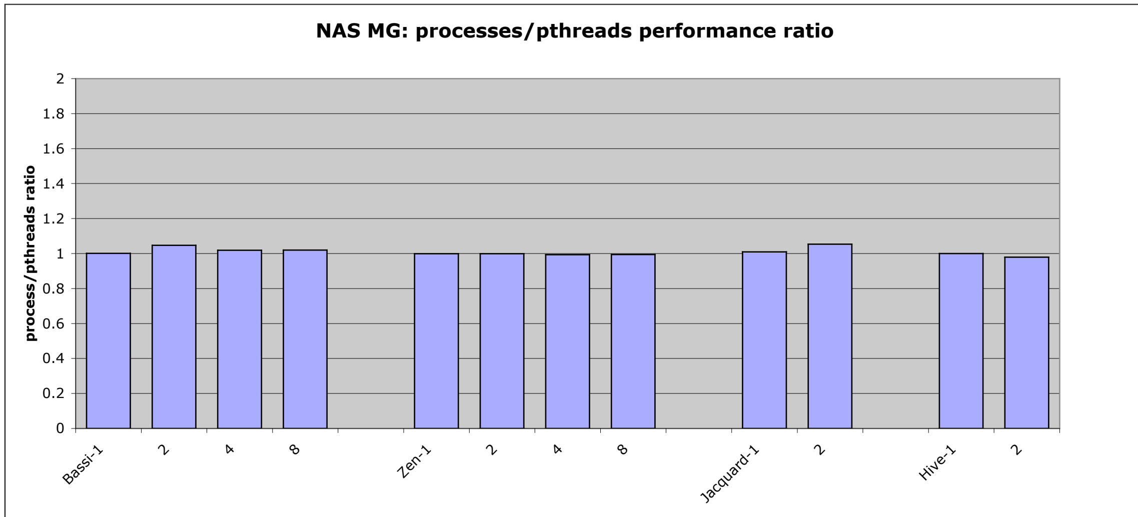

NAS FT: results for processes with shared memory
MFlops
3500
3000
2500
2000
1500
1000
500
0
Total
Per-CPU
Bassi-1
2
4
8
Zen-1
2
4
8
Jacquard-1
2
Hive-1
2


NAS FT: processes/pthreads performance ratio
process/pthreads ratio
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
Bassi-1
2
4
8
Zen-1
2
4
8
Jacquard-1
2
Hive-1
2


NAS IS: results for processes with shared memory
Mops
500
400
300
200
100
0
Total
Per-CPU
Bassi-1
2
4
8
Zen-1
2
4
8
Jacquard-1
2
Hive-1
2


NAS IS: processes/pthreads performance ratio
process/pthreads ratio
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
Bassi-1
2
4
8
Zen-1
2
4
8
Jacquard-1
2
Hive-1
2




The first set of charts reveals some interesting features of the processors used. We see that multi-core processors do not scale nearly as well as multi-socket systems, except for the EP application, which requires no communication and minimal memory traffic. Also, we see a number of cases (primarily on "bassi") where performance actually scales superlinearly as the number of processors increases. We suspect this is due to the working sets of the relatively small NAS "class A" benchmarks fitting into cache as the problem size is divided among more processors.

For the purposes of this report, however, the ratio results are of more interest. For the most part, the process/pthreads performance ratio for all of the tests hovers closely near 1, with mostly minor variance. Again, this is a vindication of our pthreads implementation, which performs and scales as well as (and in some cases, slightly better than) the processes implementation for every benchmark on almost every configuration.

The one exception to this is the performance of CG when pthreads are used on "bassi," a Power5/Intel C configuration. In this one case, the process implementation sees a 20% performance lead over pthreads at one CPU, which grows to over 250% as the application scales to 8 CPUs. This performance is replicable—the CG tests were re-run repeatedly on bassi to make sure this result was not caused by interference from other

processes, or other transient factors. A code examination revealed that fifteen global variables are referenced in the function containing the CG application kernel. While only a few of these are used at a time within inner loops, this still gave ample room to suspect that the performance loss was a result of thread-localization. This may in fact be the case, but we have so far been unable to verify it. As a preliminary investigation, the code was hand-modified to "localize" the global variables: a local copy of each global variable was created at the beginning of the function, and all uses of the variables in the kernel's inner loops were changed to use these local copies. This moved all of the thread-localization logic out of the inner loops, and ought to have eliminated any performance degradation it was causing. However, the hand-modified code still experienced the same performance degradation relative to processes. The cause of CG's poor pthreaded performance on bassi thus remains unknown at this time.

### *5.4 Performance summary*

The performance of the experimental UPC runtime using processes with shared memory is generally close to that of the pre-existing pthreads version, except when the number of global variable accesses becomes very high. Support for thread-local data needed by these global (and static) variables slows down the pthreads version by as much as 2.5x for the NAS CG and 10x for Apex-Map with all global references. Interestingly, although our UPC-to-C translator generates the exact same C code for all platforms, the impact of these global variables is not consistent across platforms: the Power5 system (Bassi) and the Itanium/Altix system (Davinci) show the most dramatic speedups for processes relative to threads. We suspect that much of these differences are compiler-driven, and relate to the unusual code generated by our system for thread-local accesses.

## 6 Conclusion and Future Work

The work performed for this report has obtained a number of useful results. First, we have at least provisionally shown that our pthreads implementation performs adequately. In the network-less SMP case, pthreads generally appear to perform just as well as a process-based runtime. To be completely confident of our the pthreads implementation, however, it will be important to examine if there is some negative interaction effect between pthreads and the various network APIs that GASNet supports. Performance comparisons with MPI and OpenMP benchmarks would also be useful. More benchmarking in these areas is needed, and may reveal areas where the UPC runtime and/or GASNet need to be tuned. There are also a number of potential optimizations for the pthreads implementation that may be worth pursuing. One would be to try using the "__thread" attribute pioneered by GCC to thread-localize variables, when it is available (a number of compilers besides GCC—including Intel C and Pathscale C—now support the attribute, although some concerns remain about the correctness of some of the implementations). Use of __thread might speed up thread-local accesses, and/or prove more palatable to compilers' optimizers than our current "big struct" solution (which would need to remain in place for compilers lacking __thread support). At least one OpenMP implementation has found a performance benefit from using __thread for similar purposes [3]. Another potential optimization would be to have the Berkeley UPC-to-C translator "localize" global variables used in functions in order to keep thread-

localization logic out of inner loops. This would be an automated way of performing the same work that we did by hand to the CG code (we would obviously need to see if such spilling ever makes a positive difference, as it did not for CG). Finally, having more understanding of what precisely causes the observed degradation in the Apex case might show the way towards a workaround that could avoid or mitigate cases where thread-localization causes drastic performance loss.

Another achievement of this work is to have completed the design work for processes with shared memory, and a first implementation for GASNet and the Berkeley UPC runtime. Although we have so far not found major performance improvements from this, the interoperability and usability advantages of a process-based implementation are significant: MPI interoperability by itself is a prize that makes a process-based solution worth implementing. It is thus it quite likely that Berkeley UPC will move towards using processes with shared memory in the future, as we determine if and how this can be made to work with all of the many systems and networks that Berkeley UPC supports. Fortunately, it appears that most major supercomputing platforms in use today that are based on shared-memory multiprocessors now provide either POSIX shared memory, or a System V shared memory implementation generous enough for UPC's purposes. The landscape is thus more favorable for a portable processes-with-shared memory approach than it was when the Berkeley UPC project was first started.

# 7 References


[1] D. H. Bailey and E. Barszcz and J. T. Barton and D. S. Browning and R. L. Carter and D. Dagum and R. A. Fatoohi and P. O. Frederickson and T. A. Lasinski and R. S. Schreiber and H. D. Simon and V. Venkatakrishnan and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*. 5(3):63-73, Fall 1991.

[2] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. *20th International Parallel & Distributed Processing Symposium (IPDPS),* 2006

[3] X. Martorell, M. Gonzalez, A. Duran, J. Balart, R. Ferrer, E. Ayguade, and J. Labarta. Techniques supporting threadprivate in OpenMP. In *Proceedings of the 11th International Workshop on High-Level Parallel Programming Models and Supportive Environments*. Rhodes Island, Greece. April 2006.

[4] The Berkeley UPC Compiler. 2002-2007. http://upc.lbl.gov.

[5] D. Bonachea. GASNet Specification. Technical Report CSD-02-1207, University of California, Berkeley, October 2002.

[6] W. Chen, C. Iancu, and K. Yelick. Automatic nonblocking communication for partitioned global address space programs. In *Proceedings of the International Conference on Supercomputing (ICS),* 2007.

[7] W. Chen, C. Iancu, and K. Yelick. Communication optimizations for fine-grained UPC applications. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT),* 2005.

[8] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey. Coarray Fortran performance and potential: An NPB experimental study. In *16th Int'l Workshop on Languages and Compilers for Parallel Processing (LCPC),* October 2003.

[9] K. Datta, D. Bonachea, and K. Yelick. Titanium performance and potential: an NPB experimental study. In *The 18th International Workshop on Languages and Compilers for Parallel Computing,* October 2005.
[10] Duell, Jason. Allocating, Initializing, and Referring to Static User Data in the Berkeley UPC Compiler. Technical specification. Available at http://upc.lbl.gov/docs/system/runtime_notes/static_data.html.
[11] T. El-Ghazawi and F. Cantonnet. UPC performance and potential: an NPB experimental study. In *Supercomputing 2002 (SC 2002),* November 2002.
[12] GASNet home page. http://www.cs.berkeley.edu/~bonachea/gasnet.
[13] P. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium language reference manual. Tech Report UCB/CSD-01-1163, U.C. Berkeley, November 2001.
[14] C. Iancu, P. Husbands, and W. Chen. Message strip mining heuristics for high speed networks . In *Proceedings of the 6th International Meeting on High Performance Computing for Computational Science (VECPAR),* 2004.
[15] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Proc. RTSPP IPPS/SDP'99*, 1999.
[16] R. Numrich and J. Reid. Co-array fortran for parallel programming. In *ACM Fortran Forum* 17(2), 1-31. 1998.
[17] Open64 Compiler tools. http://open64.sourceforge.net.
[18] E. Strohmaier and H. Shan. Apex-Map: A Synthetic Scalable Benchmark Probe to Explore Data Access Performance on Highly Parallel Systems. *EuroPar2005,* 2005.
[19] Texas Advanced Computing Center (TACC), “Ranger” Linux cluster. . http://www.tacc.utexas.edu/resources/hpcsystems/
[20] UPC consortium home page. http://upc.gwu.edu/.
[21] UPC language specifications, v1.2. Technical Report LBNL-59208, Berkeley National Lab, 2005.
[22] T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture,* May 1992.
[23] S. M. Yau. Experiences in using Titanium for simulation of immersed boundary biological systems. Master’s report, Computer Science Division, University of California, Berkeley, May 2002.
[24] K. Yelick, P. Hilfinger, S. Graham, D. Bonachea, J. Su, A. Kamil, P.C.K. Datta, and T. Wen. Parallel languages and compilers: perspective from the Titanium experience. In *The International Journal of High Performance Computing Applications,* 21(2), 2007.