Added Concurrency to Improve MPI Performance on Multicore

Humaira Kamal, Alan Wagner
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada
Email: {kamal,wagner}@cs.ubc.ca

***Abstract*—MPI implementations typically equate an MPI process with an OS-process, resulting in a coarse-grain programming model where MPI processes are bound to the physical cores. Fine-Grain (FG-MPI) extends the MPICH2 implementation of MPI and implements an integrated runtime system to allow multiple MPI processes to execute concurrently inside an OS-process.**

FG-MPI's integrated approach makes it possible to add more concurrency than available parallelism, while minimizing the overheads related to context switches, scheduling and synchronization. In this paper we evaluate the benefits of added concurrency for cache awareness and message size and show that performance gains are possible by using FG-MPI to adjust the grain-size of a program to better fit the cache and potential advantages in passing smaller versus larger messages.

We evaluate the use of FG-MPI on the complete set of the NAS parallel benchmarks over large problem sizes, where we show significant performance improvement (20%-30%) for three of the eight benchmarks. We discuss the characteristics of the benchmarks with regards to trade-offs between the added costs and benefits.

***Keywords*- Fine-Grain MPI, Message Passing, MPICH2, Multicore, Performance, Concurrency, Over-decomposition.**

I. Introduction

MPI has been very successful in High Performance Computing for implementing message-passing programs on compute clusters.There are many applications and a variety of libraries that have been written using MPI. Many of these programs are written as SPMD programs where the program is parameterized by "N" the number of MPI processes. Parameter N determines the granularity of the program and gives the amount of available concurrency. In executing MPI programs, one typically matches the number of MPI processes to the number of cores, the amount of available parallelism.

Matching the concurrency to the available parallelism fixes the granularity of the program to make it as coarse-grain as possible. However, maximizing the granularity is not always optimal because of the effect it has on the cache behavior and the number and sizes of the messages sent and received. There are also MPI programs where N is partly determined by the problem size and may not exactly match parallelism available in the machine. For these reasons it should be possible to be able to adjust the granularity independently from the amount of parallelism and be able to expose more concurrency than can be executed in parallel.

Introducing added concurrency by over-decomposing the problem is a well-known performance optimization technique for SPMD scientific computing programs. Since data decomposition is typically hard-coded and difficult to change, over-decomposition for MPI programs depends on its runtime environment. One simple technique that is commonly used is to oversubscribe the number of cores by starting more MPI processes and thereby more OS-processes. A recent paper [1] studied the use of oversubscription on multicore machines and report a 10% performance degradation for the NAS benchmarks with MPI. There are also systems like Adaptive MPI (AMPI) [2] that support added concurrency by implementing MPI on top of Charm++, an object-based runtime system. The performance advantages of AMPI has been reported on a subset of the NAS benchmarks of size A and B [3], but not on a wider range and bigger sizes of the benchmarks and they do not discuss issues pertaining to multicore. In this paper we describe and evaluate FG-MPI, an extension to the MPICH2 middleware, that allows us to adjust granularity at runtime independently from the available parallelism. We evaluate FG-MPI by reporting results for the NAS benchmarks on a cluster of multicore machines.

The key issue in a system that adds more concurrency than available parallelism is to minimize the overheads in order to maximize the benefits and make it effective over a wider range. FG-MPI does this by making it possible to have multiple MPI processes executing concurrently inside an OS-process. FG-MPI uses coroutines to create a small kernel running inside each OS-process that is tightly integrated into the MPI middleware. Coroutines were used to implement light-weight processes that were non-preemptively scheduled by our kernel. This allowed for fast context switches and also made it possible to introduce a user-level scheduler that worked in concert with the MPI middleware. The tight integration is a key difference between FG-MPI and other systems (see Section V) and made it possible to address overhead issues related to context switches, MPI-aware scheduling, as well as added synchronization costs, which was also highlighted as a problem in [1]. It also gives us control over the mapping and scheduling of computation to cores and machines and scales up, if necessary, to expose massive amounts of concurrency (millions of MPI processes).

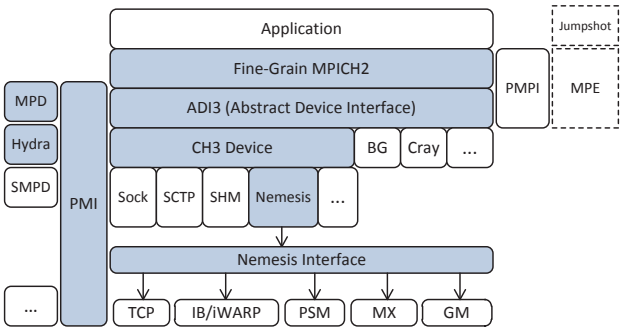


Fig. 1. FG-MPI Architecture. Shading shows the layers of MPICH2 that FG-MPI is integrated into. Figure adapted from [4].

This paper makes the following contributions:

- It introduces an alternative non-layered approach for adding concurrency to MPI programs by making it possible to have multiple MPI processes inside an OS-process.
- We describe the key difference, the tight integration of FG-MPI into the MPICH2 middleware, and measure the overheads and benefits on multicore processors.
- We evaluate the benefits of added concurrency for cache awareness and message size and show that performance gains are possible by using FG-MPI to adjust the grain-size of a program to better fit the cache and potential advantages in passing smaller versus larger messages.
- We evaluate over-decomposition using FG-MPI on the complete set of NAS benchmarks over large problems sizes where we show significant performance improvement (20%-30%) for three of the eight benchmarks. We discuss the characteristics of the benchmarks with regards to trade-offs between the added costs versus benefits.

In Section II we give an overview of the system and changes made to MPICH2 to support FG-MPI. In Section III we describe and measure the overheads and potential benefits to adding concurrency to MPI programs using FG-MPI. The performance results for adding concurrency to the NAS benchmarks programs for problem sizes B, C and D are presented in Section IV. In Section V we compare FG-MPI to related work and present the conclusions in Section VI.

II. FG-MPI Overview

FG-MPI implements a user-level runtime integrated into MPICH2 to allow for multiple MPI processes within one OS-process. In order to avoid any ambiguity we will use the term "*OS-process*" when referring to operating systems processes and at all other places the terms *process*, *fine-grain process* and *MPI process* will be used interchangeably. MPI processes sharing the same address space are referred to as *collocated* processes.

Figure 2 shows the integration of FG-MPI in the layered modular architecture of MPICH2. The MPICH2 ADI3 layer represents the data structures and functions that are provided by an implementation. Representation in this layer is in terms of MPI requests/messages and the functions for manipulating those requests. One of first considerations in integrating FG-MPI in MPICH2 was to support large amounts of concurrency through scalable sharing of MPI structures among the coroutines. To this end, a large number of MPI storage structures such as posted receive queues, unexpected messages queues, communicator and request pools are shared by the coroutines. Devices in MPICH2 are communication mechanisms, which are paired with channels that represent specific modes of communication. In FG-MPI, we leverage the Nemesis `CH3` channel since it is designed for scalability and is a highly optimized, communication subsystem that provides multi-network support. It provides low latency, lock-free shared memory queues and high performance communication for both intra-node and inter-node communication [5]. Other structures which are an integral part of MPI are communicators and groups and their scalability and sharing is essential to FG-MPI. In past work [6] we discuss in detail how we share these structures and scale to hundreds and thousands of MPI processes.

The MPICH2 implementation couples the naming of an MPI process with an OS-process which, in turn, is tied to its communication endpoint. In FG-MPI, we decouple MPI process from its points of attachment to allow multiple MPI processes to share the same address space. This required creating a 2-level namespace, where the OS-processes are named separately from the MPI processes. As well, extending the MPI message matching to represent the new hierarchy and multiplexing and de-multiplexing of messages.

The MPI progress engine is responsible for message progression and in our case, the coroutines can cooperatively progress messages for other collocated coroutines. It also enables us to implement optimized communication among collocated processes. As shown in Sections III-B and III-C, it is important to take advantage of the single shared address space for communication involving collocated processes and not rely solely on MPI's point to point communication. Due to the direct integration in MPICH2, our runtime scheduler is aware of the MPI events, such as arrival of messages, occurring inside the progress engine and can block or unblock collocated MPI processes when any pending requests can be progressed. The tight integration enables us to achieve the performance benefits that would not have been possible by layering light-weight threads or a runtime on top of the MPI middleware.

The Process Manager Interface (PMI) is extended to support an `nfg` (**n**umber of **f**ine-**g**rain) flag to `mpiexec`. Using `-nfg` the user can choose how many MPI processes to run per OS-process in combination with the `-n` flag specifying the number of OS-processes. We introduce the following notation to describe the mapping of processes in FG-MPI. We specify the hierarchical structure of an MPI execution in terms of P, the number of MPI processes per OS-process as given by `-nfg`, O, the number of OS-processes per machine, and M, the number of machines. N = P×O×M is the number of MPI processes in a [P;O;M] execution. The standard one MPI process per OS-process model corresponds to a [1;O;M] execution. In general, the number of fine-grain processes (P)

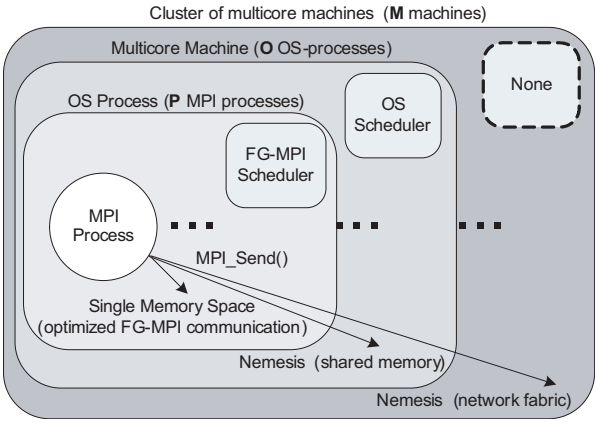


Fig. 2. Hierarchical view of FG-MPI runtime environment with N = P×O×M MPI processes.

inside an OS-process can vary, but in the remainder we will assume the SPMD model with the same P for every O. Given this notation, concurrency can be added by over-subscription (increasing O to be larger than the number of cores per machine) and/or increasing P.

When $P > 1$ a FG-MPI kernel, with a scheduler, is started inside the OS-process. The scheduler is non-preemptive where, as is the case in MPI, a process runs until it makes an MPI call. In addition to scheduling, the runtime accesses the middleware to progress messages for all other collocated processes. Because the runtime is non-preemptive, accesses to the middleware are guaranteed to be atomic, avoiding the overhead of locking, and we only need to ensure that at every scheduling point the middleware is left in a consistent state. One effect of non-preemptive scheduling is that a process that is busy computing blocks the progress of all other collocated processes. One assumes that as long as the process is busy it is making progress, however, we did add `MPIX_Yield()` to handle cases when a fairer scheduling is needed.

This approach may require some changes to the source code. The user needs to be aware that any global variables become shared variables and these variables should be localized. There are a variety of tools available to help removing globals. In the case of NAS benchmarks, which are written in Fortran, we used Photran [3], a tool developed by the AMPI group for privatizing variables. The advantages and disadvantages of user-level threads are well-known and solutions like scheduler activations [7] exist. Scheduler activations are not supported in Linux, however, it would be interesting to use FlexSC [8], a Linux kernel extension, to schedule and batch system calls to the kernel making it possible for the user-level scheduler to coordinate with the OS scheduler.

In conclusion, FG-MPI runtime with a scheduler, sharing of the middleware, and optimized communication for collocated processes benefits from the tight integration to MPICH2. These benefits cannot easily be obtained by layering a system on top of MPI with no visibility into the state of the middleware. FG-MPI runs on commodity operating systems and does not require any special support.

III. FG-MPI Runtime

There are advantages and disadvantages to introducing more concurrency. There are additional costs for context switching, scheduling as well as additional messaging. In terms of benefits there is better cache behavior and potential advantages in passing smaller versus larger messages. Adding concurrency is only useful when the benefits out-weigh the overhead that results from the over-decomposition. In this section we describe the FG-MPI runtime and measure the overheads associated with context switching and messaging. For block-structured algorithms we measure the potential gain that can be achieved by better cache behavior.

In Section III-A, we describe the scheduling of processes inside the OS-process and its interaction with the MPI middleware. We show that FG-MPI is an order of magnitude faster than OS-level context switch. In Section III-B we measure the overheads of the extra message passing that can occur to show that it is similar to that of a memory copy. Adding concurrency also affects the communication time for collectives since now these collectives are over a larger collection. In Section III-C we describe the use of location-aware implementation of collectives that takes advantage of the single address space to speed up the collectives for collocated processes. We measure this added overhead for the `MPI_Barrier()` call. Finally in Section III-D we look at the potential performance improvements for better cache behavior.

For the experiments the test setup consisted of a cluster with 16 nodes connected by a 10GigE Ethernet interconnection network. Each of the nodes in the cluster is a quad-core, dual socket (8 cores per node) Intel Xeon® X5550, 64-bit machine, running at 2.67 GHz. All machines have 12 GB of memory and run Linux kernel 2.6.18-194.8.1.el5.

A. FG-MPI Context Switch

One important cost that is a consequence of adding concurrency is the time taken to switch between processes. FG-MPI's non-preemptive runtime is build on top of coroutines, each with it own stack. The runtime is partially derived from Capriccio [9], a scalable thread library for high-concurrency servers. We use Toernig's coroutine (`coro`) library [10], which provides highly efficient yield for switching context between coroutines. We did extend the system to be able use any similar type of coroutine package and now provide the option for using PCL (Portable Coroutine Library) [11].

In FG-MPI every MPI call is a potential de-scheduling point where, depending on the call and the state of the middleware, the scheduler chooses the next process to run. MPI communication calls provide a natural yield point for switching between coroutines where one process when it enters the middleware can progress messages for all of the collocated processes. A "progress engine" coroutine that remains on the runnable queue whenever there is a receive that could be matched by a message from a remote process ensures that, when necessary, we poll the external link for more data.

Switching between processes involves switching to the runtime scheduler, selection of next runnable MPI process by the scheduler and switching to the new process. We measured this switching time for both of the coroutine packages that FG-MPI can use. The `coro` library provides the fastest switching time (0.13 μs) while the PCL library switching time is 0.81 μs. This time is an order of magnitude faster than OS-level context switch which takes 6.85 μs. Our results are similar to the numbers reported for threading benchmarks in Capriccio. In a comprehensive study, they demonstrated that coroutines outperform NPTL (Native Posix Thread Library) and Linux-Threads (Linux kernel threads) for both raw performance and scalability [9].

In conclusion, coroutines provide very efficient context switch time in comparison to other types of threads and this makes it easier to scale to support massive amount of concurrency. We have tested FG-MPI with thousands of collocated processes and, when not constrained by memory, have used it for debugging with `gdb` to run an entire MPI application in a single OS-process. The non-preemptive runtime model is a natural match for message passing systems as the communication routines provide the natural yield points in a cooperative threading environment.

B. Messaging Costs

In FG-MPI one of the effects of adding more concurrency is that we send more smaller sized messages. In the case of communication between collated processes it is possible to use the single address space to optimize point-to-point by simply doing a `memcpy`. Our objective was to achieve low messaging overhead through synchronous communication between collocated processes and avoid intermediate system copies.

For collocated processes, there are two cases to consider depending on whether the sender or the receiver executes first. The two cases are symmetric, therefore, we describe in detail the one where a receiver process executes a receive call before it can be completed; i.e., the sender process has not yet executed a matching send call.

In this case, the receiver process first determines that the sender has not yet executed a send call by looking in the unexpected-message-queue and queues its request in the posted-receive-queue. It then blocks waiting for the data arrival. When the sender process executes the send call, it first checks in the posted-receive-queue and finds the matching receive. The sender then copies the data directly into the receiver's buffer and unblocks it. Note that there is only one memory copy in this operation and no intermediate system copies are made. The additional overheads over a simple `memcpy` in this case are the context switch costs, looking in the unexpected-message-queue and queueing the receive request in the posted-receive-queue by the receiver and finding the matching request in the posted-receive-queue by the sender.

For non-collocated processes, MPICH2 treats messages of 64 KBytes or above as long messages that require a rendezvous protocol for communication. For collocated communication we avoid the extra cost of rendezvous and the communication between two collocated processes is the same as described above, irrespective of the size of the message.

In order to measure the cost of message communication within collocated MPI processes, we designed a benchmark that compares sending MPI messages, picked randomly from a memory location, between two collocated processes with the cost of doing a memory copy inside one process. We measured the overhead (difference between a message send/receive and a `memcpy`) for different messages sizes. The reason for randomly picking from a memory location was to isolate the effect of messaging only (i.e. without any cache effects). We also ran a test where the same message is being sent and the overhead of messaging over `memcpy` compared to the random selection case was less, but we report the numbers for the random selection as it is closer to what we would expect in an application. Our results showed that the average overhead of collocated messaging in comparison to a `memcpy` operation was 0.43 μs with a standard deviation of 0.065 μs for messages ranging in size from 2 bytes to 128 Kbytes. For messages in the range 256 Kbytes to 1 Mbytes, the message overhead increased from 1 μs to 2.3 μs. Note these measurements include the context switching and scheduling time to complete the operation.

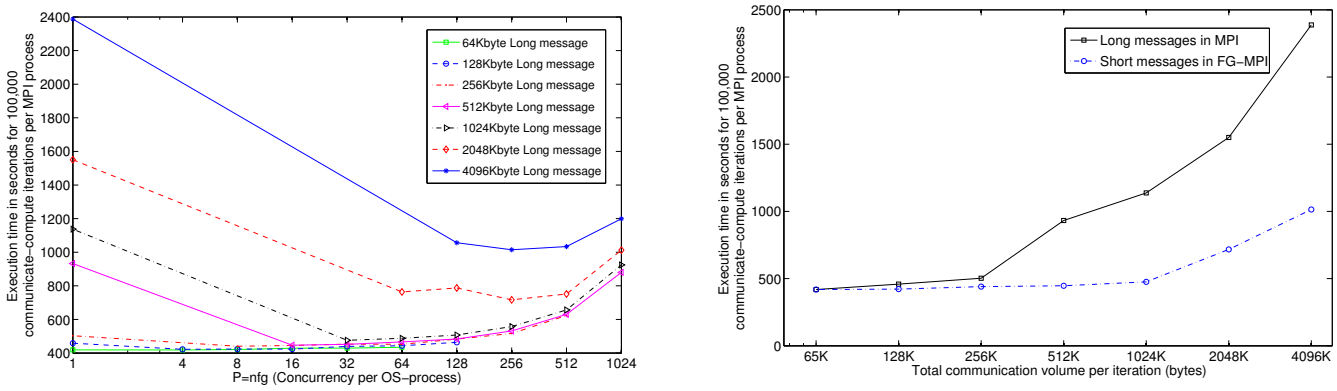
Using the same program, we measured the collocated messaging cost for AMPI. For message sizes in the range 2 bytes to 256 bytes, the difference between messaging times for FG-MPI and AMPI was in the range 0.12 μs to 0.15 μs, with AMPI slightly faster. For messages 512 bytes and above the difference increased sharply with FG-MPI outperforming AMPI by a large margin as shown in Table I.

Message size	FG-MPI time	AMPI time
1 Kbytes	0.6 μs	0.8 μs
8 Kbytes	1.4 μs	3.6 μs
32 Kbytes	4.23 μs	15.1 μs
64 Kbytes	7.9 μs	fails

TABLE I
COMPARISON OF FG-MPI AND AMPI COLLOCATED MESSAGING TIMES.

Interestingly, AMPI failed to execute for message sizes 64 Kbytes and higher, saying that it cannot allocate memory. With FG-MPI we successfully tested up to 8 Mbytes on the same machine without any problems.

One potential benefit to adding concurrency for communication between remote processes is that by sending more smaller messages, rather than one large one, we can avoid the long-message protocol that requires a rendezvous between the processes. In effect, the added concurrency acts like packetization where the packets (smaller messages) can be pipelined between the two OS-processes. Messages can be split up into smaller pieces and the MPI processes at the receiver can begin working on the smaller pieces without having to wait for the entire message to arrive. In order to study this effect, we designed a benchmark where processes alternate between computation and communication for a number of iterations. In the coarse-grain MPI case we have two processes doing a



(a) (b)

Fig. 3. Long Messages in MPI versus Short messages in FG-MPI

fixed amount of computation (Cp) and communication (Cm) per iteration. In the finer-grain case, Cp and Cm per OS-process is divided among the number of processes defined by the nfg flag while keeping the total volume of computation and computation the same in both cases. In coarse-grain MPI fewer number of large messages are communicated whereas in FG-MPI the size of messages is smaller (divided by P=nfg), but the total number of messages communicated is multiplied by the nfg parameter.

Figure 3(a) shows the effect of sending fewer long messages versus several short messages[1]. The advantages of avoiding the rendezvous protocol are clear in this figure. We also see that the overhead of additional messaging in FG-MPI is small (even for substantially large values of nfg) and clearly offset by the advantage of avoiding the rendezvous protocol. Figure 3(b) shows the best times achieved in FG-MPI for different communication volumes with MPI.

Further optimization is possible with collocated processes in a single address space by using copyless message passing where processes pass a reference to the data. Systems like Singularity OS [12] have explored the use of contracts to guarantee safe passing of references.

In conclusion, FG-MPI makes it possible to adjust the message size independently from the size of the machine and the added concurrency results in a more fluid communication with more messages flowing potentially requiring less synchronization with more overlap between communication and computation. We expect this benefit to become more evident on programs computing on larger datasets.

C. Collective Communication

Concurrency adds overhead to the collectives since it results in more MPI processes and hence more messages and more context switches. The cost of synchronization has a significant impact on MPI programs that use collectives and may offset any of the potential advantages of added concurrency. We reduce the synchronization overhead by taking advantage of the single address space and optimizing the collective communication for those processes that are collocated. We demonstrate this approach for the MPI_Barrier operation.

For MPI_Barrier, one leader per OS-process is selected from the collocated MPI processes and one leader is selected for all of the OS-processes on the machine. Inside the OS-process, a shared variable is used to count the collocated processes that enter the barrier. Processes inside the OS-process increment the counter and block waiting for the collocated leader to clear the barrier and re-schedule them. Similarly, the OS process leaders uses MPICH2 Nemesis's shared memory to coordinate with the leader of all the OS-processes on the machine. Once processes on each multicore node have synchronized then all those leaders communicate after which the leader of the OS-processes signals that they can leave the barrier which in turn allows the collocated processes to leave the barrier. This type of location-aware implementation of the MPI_Barrier is optimized for each level of the communication hierarchy and is supported by the scheduler to minimize its interactions with the middleware. The naive approach of simply relying on the MPI's point-to-point can take advantage of memcpy for the communication among collocated processes but it results in more interactions than necessary with the middleware and progress engine.

P (nfg)	[P;1;1] time (μs)	[P;1;1] N	[P;8;1] time (μs)	[P;8;1] N	[P;8;16] time (μs)	[P;8;16] N
1	-	1	1	8	122	128
2	1	2	1	16	126	256
4	1	4	2	32	129	512
8	1	8	3	64	129	1024
16	3	16	4	128	137	2048
32	5	32	10	256	149	4096
64	10	64	20	512	183	8192
128	22	128	41	1024	259	16384
256	45	256	84	2048	438	32768

TABLE II
BARRIER LATENCY TIME (μS) WITH VARYING CONCURRENCY (P) FOR THREE [P;O;M] EXECUTIONS, WHERE P IS THE NUMBER OF MPI PROCESSES PER OS-PROCESS, O IS THE NUMBER OF OS-PROCESSES PER NODE AND M IS THE NUMBER OF NODES. N=P×O×M IS THE TOTAL NUMBER OF MPI PROCESSES.

Table II presents the barrier latency as concurrency (P) per OS-process in a [P;O;M] system is increased from 1 to 256

[1]MPICH2 treats messages of 64 KBytes or above as long messages that require a rendezvous protocol for communication. For the fine-grain results, The nfg parameter is chosen to divide the message size so that it is less than 64 KBytes

for [P;1;1], [P;8;1] and [P;8;16] executions[2]. The increase in barrier latency with concurrency is sublinear, e.g., on a single multicore node ([P;8;1]), the latency increases from 1μs to 84μs as concurrency increases from 1 to 256.

These results were obtained by iteratively calling the barrier operation several thousand times and averaging the result. As such, they are a lower bound estimate on the cost of barrier communication. Although an OS-process will continue to make progress as long as there is an MPI process to execute, due to OS-scheduling one process could end up idling waiting for other local or remote processes. By not over-subscribing the cores we reduce the influence of the OS-scheduler. In [1] over-subscription was found to have a major effect on the performance of MPI programs containing collectives.

It is also interesting to compare this approach to AMPI. In AMPI we have MPI on top of the Charm++ runtime which in turn uses MPI as a communication layer. Its implementation of MPI can still take advantage of the single address space, but the Charm++ scheduler and lower level MPI progress engine operate independently with the potential for the same types of delays that occur between the progress engine and the OS scheduler. Using the same program, we measured the cost of MPI_Barrier for AMPI on [P;8;1]. For P=1 the time was 18 μs, versus 1 μs for FG-MPI, and for P=256 was 662 μs versus 84 μs for FG-MPI. On multicore, this shows the advantage of reducing the layers by supporting concurrency inside the communication middleware.

D. Cache Behavior

In order to demonstrate the potential benefits of using added concurrency to improve the cache behavior, we designed several experiments to measure the cache effects. We focus on block structured algorithms as they would provide the best case for measuring the extent to which we can improve performance.

Block structured algorithms in MPI are commonly used for parallel scientific computations. The blocks generally represent the working set size of a task and blocking is used to exploit both temporal and spatial locality for efficient execution. Today's systems of multiple cores on a single chip have a multi-level hierarchical memory model with smaller caches per core [13]. Applications need to be able to express fine-grain parallelism with smaller working set sizes to fit in caches. FG-MPI provides the ability to achieve better memory locality and cache hit ratios simply through the use of the `nfg` parameter on the command line.

The tests are run on a single multicore machine with the specification described in Section III. Each multicore machine has 8 cores with three-level cache; L1(data/instr) is 32K/32K, L2 cache is 256K/core and L3 cache is 8M /socket. Memory per core is 1.5G/core.

1) Experiment A: This benchmark algorithm takes two square matrices as input and partitions them into square blocks of size determined by the total number of MPI processes. It then assigns each pair of sub-matrices to the MPI processes. Each process computes on these sub-matrices and then exchanges its sub-matrices with its neighbours[3]. After the exchange, each process computes with new values in its sub-matrices and this process repeats for a number of iterations. This benchmark requires the total number of MPI processes to be a square because the sub-matrices are evenly distributed among them.

Figure 4 shows the effect of added concurrency on execution times compared with MPI on input matrices of size 4096. The P=`nfg` parameter on the bottom axis is the concurrency for different [P;O;1] executions for a total number of P$\times$O MPI processes. The hashed bars correspond to the traditional MPI [1;O;1] executions, while added concurrency is represented by solid bars. Notice that the hashed bars for MPI are not present for [1;2;1] and [1;8;1] because they are not equal to a square number of processes. FG-MPI, however, can use all the cores by appropriately setting P=`nfg` so that the product of P and O is a square.

As the total number of the MPI processes (P$\times$O) increases, the block sizes that each of them operate on decreases. Figure 4 shows substantial performance improvements by adjusting the block size through added concurrency, without any serial cache blocking optimizations. The best time obtained with FG-MPI is `12.55` seconds with a [512;8;1] execution. Note that our node has 8 physical cores and the FG-MPI best results correspond to the case where the amount of parallelism equals the number of cores (i.e. without over-subscription) and the effects of OS-scheduler are removed. For interest, Figure 4 also shows the effects of over-subscription for this example using [P;16;1] (two OS-processes per core). The time achieved with MPI is `70.49` seconds with [1;16;1] execution, which is more than 5 times slower than the best time of `12.55` seconds achieved with FG-MPI mentioned above. Note that serial blocking techniques can be used to improve cache hit ratios but such techniques require modifications to the algorithms and are tuned for particular architectures and are not portable.

In order to quantify the effect of cache on our results, we collected memory access data using Intel's® VTune™Amplifier XE [14] advanced hardware analysis for the Nehalem® micro-architecture. This analyzer has low overhead and uses event-based sampling for data collection.

	Time (sec)	LL Cache misses
MPI [1;16;1]	70.49	396,000,000
FG-MPI [512;8;1]	12.55	75,000,000

TABLE III
LL CACHE MISSES FOR THE BEST TIMES ACHIEVED WITH MPI AND FG-MPI BEST TIMES IN EXPERIMENT A.

Table III shows there is a significant difference in the LL cache misses[4] for the best times in Figure 4 with added concurrency,

[2]O (the number of OS-processes) is kept equal to the number of physical cores per machine.

[3]The computation is matrix multiplication in our benchmark.

[4]The last-level (LL) cache misses influence the runtime the most because it masks accesses to the main memory [15].

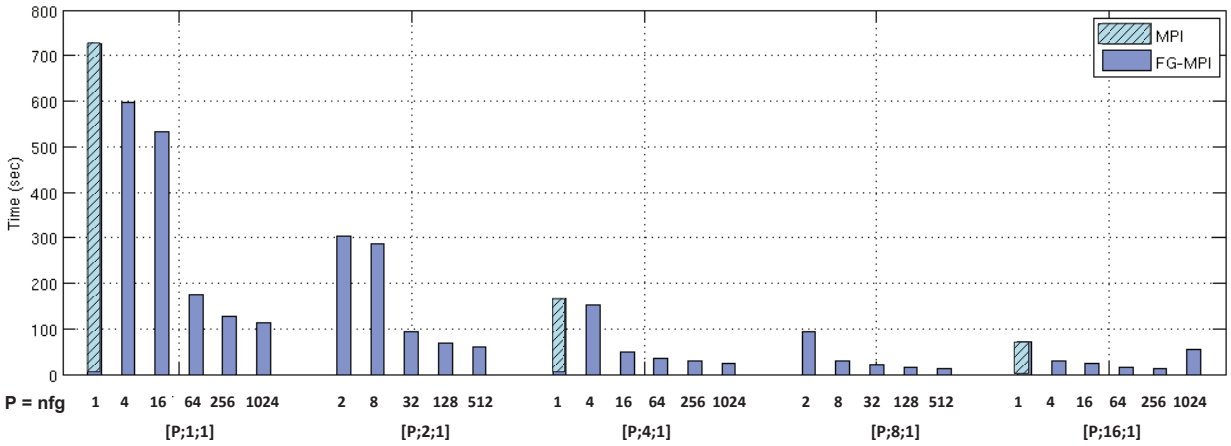


Fig. 4. Execution times (sec) with different values of concurrency (`nfg`) for Experiment A on one multicore node

which shows that the performance benefits are the result of better cache locality.

Matrix size	FG-MPI [P;O;1]			MPI [1;16;1]
S	P=nfg	O	Time (sec)	Time (sec)
2048	128	8	1.47	3.7
4096	512	8	12.55	70.49
8192	2048	8	134.5	944.58

TABLE IV
BEST EXECUTION TIMES FOR MPI AND FG-MPI ON DIFFERENT MATRIX SIZES FOR EXPERIMENT A.

The effect of added concurrency was repeated in matrices of other sizes, as shown in Table IV, with the performance improvement over MPI increasing with larger matrix sizes. This is expected as in the MPI case, the block granularity is fixed to the amount of parallelism and with larger matrices the mismatch with the cache size increases. It is interesting to note from Table IV, that the $(\frac{S^2}{P \times O})$ ratio is the same for the best times achieved for different matrix sizes. This indicates that it may be possible to analytically determine the value of P.

2) Experiment B: In general block structured algorithms, especially matrix multiplication, use libraries like the BLAS for matrix operations. Although the previous experiment may be indicative of more general algorithms, there is the question of whether the use of these libraries may eliminate the need to add concurrency for block-structured matrix operations.

We implemented a version of the Cannon's Matrix Multiplication algorithm, where FG-MPI is used to decompose the input square matrices into blocks and each of the blocks are multiplied through the ATLAS (Automatically Tuned Linear Algebra Software) BLAS GEMM [16] serial routine. We set O to 8 to take advantage of all the cores on the multicore machine and increased the concurrency (`nfg`) from 2 to 128. Because the number of processes needs to be a square `nfg`=2 is the closest to executing it with just MPICH2 [5].
For the smaller matrix sizes using the minimum amount of concurrency is best, however, for larger matrix sizes Table V

Matrix size	FG-MPI			Intel MKL
	P=nfg	O	Time (sec)	Time (sec)
2048	2	8	0.31	0.25
	8	8	0.31	
	32	8	0.44	
	128	8	0.73	
4096	2	8	1.78	1.27
	8	8	1.81	
	32	8	1.88	
	128	8	3.20	
8192	2	8	12.70	8.14
	8	8	11.95	
	32	8	11.43	
	128	8	13.15	
16384	2	8	107.33	58.04
	8	8	86.02	
	32	8	80.03	
	128	8	81.39	

TABLE V
EFFECT OF ADDED CONCURRENCY WHILE USING BLAS GEMM ROUTINES ON A SINGLE MULTICORE MACHINE

begins to show an improvement centered around $P = 32$. Even with the use of the BLAS, for larger matrices, we obtain a performance improvement over the minimum concurrency execution.

It is also interesting to compare this to other techniques such as the Intel's MKL library that is specific to Intel architectures with a completely different runtime. The MKL library is optimized for the architecture on our Intel Xeon® system. The MKL library was compiled with optimizations enabled. In Table V, we report the results with Intel MKL's GEMM threaded parallel routine[6]. Although slower, the performance is within 70% of the Intel's optimized runtime. The Intel MKL library is based on OpenMP and runs on one machine whereas for FG-MPI there is the ability distribute computation across multiple nodes for matrix sizes greater than 16K, that may not fit on a single node.

[5]The maximum square matrix size tested was 16K due to the amount of memory available on our node.

[6]Threading in Intel MKL is based on OpenMP specification and the GEMM routines used all 8 cores on the node.

IV. NAS BENCHMARKS

The NAS Parallel Benchmarks (NPB2.4) [17] are a set of eight standard benchmarks that are used to evaluate the performance of parallel systems. Each of the eight benchmarks can be compiled for different problem classes (`CLASS`) and the number of MPI processes (`NPROCS`). The problem classes range from `A` (smallest) to `D` (largest). Class `D` was introduced to provide more challenging benchmark sizes for high-performance computer systems that have grown significantly in size and capacity in the last decade. Class `D` benchmark involves about 20 times as much work, and a data set that is approximately 16 times larger than the Class `C` benchmark. The NAS benchmarks do not contain an implementation for `IS` class `D`, and `SP` and `BT` run on a square number of MPI processes. In this section we evaluate the effects of added concurrency on the performance of the NAS benchmarks. For the experiments, we use the cluster described in Section III.

A. Performance of the NAS Benchmarks

We ran an extensive set of experiments for each of the benchmarks using different `CLASS` and [P;O;M] combinations. We explored the execution space for the 32 problem sizes (4 classes: A,B,C,D for the 8 benchmarks) with P (concurrency) ranging from from 1 to 1024, O (OS-processes per node) in the range 1 to 8 and M (number of nodes) varying from 1 to 16. The maximum number of OS-processes (O) per node was fixed to the number of physical cores per node. The mapping of OS-processes to nodes was done in blocks of eight, with the first eight on first node and next eight on the next, etc. We do not oversubscribe OS-processes to cores for reasons discussed in [1], which reports a 10% performance degradation for the NAS benchmarks with MPI. Our goal in these experiments is to focus on the effects of added concurrency on MPI performance.

We view `nfg` as a variable to the execution that can be used to adjust cache locality and message sizes independently from the size of the cluster. For each of the benchmark problem sizes, we experimentally varied `nfg` to determine the best FG-MPI performance for different [P;O;M] ($P > 1$) executions, and compared that to the MPI performance achieved with [1;O;M] executions. Note that when P=1, only the MPICH2 middleware executes in the OS-process, which is equivalent to an MPI execution without the added concurrency. We present our results by normalizing the MPI performance and reporting the FG-MPI results as a percentage increase or decrease. We roughly characterize problem sizes running under a minute as short-lived and above that as long-lived. For the benchmarks considered, our results show that added concurrency has different effect on short and long-lived problem sizes.

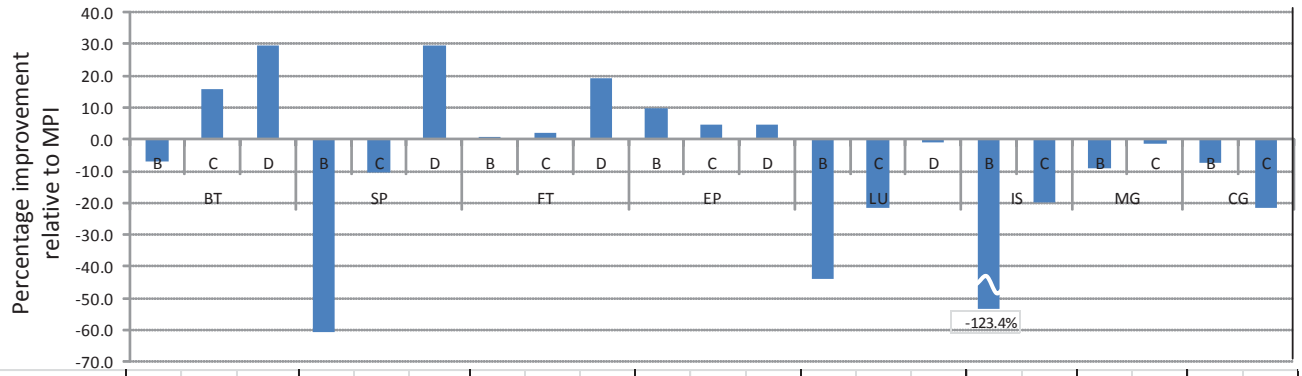
Figure 5 presents our results[7]. We are omitting results for class A as this is a small problem size that is mostly relevant for testing purposes. Each of the bars represent the percentage effect of added concurrency on the execution time of the benchmarks over the best performance achieved for MPI [1;O;M]. Cores=O×M represent the number of physical cores for the best MPI performance. Concurrency in [P;O;M] executions is the value of P=`nfg` per OS-process for the same values of O and M as above. Figure 5 also shows the MPI execution times in seconds to allow differentiation between short and long-lived applications.

[7]Results for `CG` and `MG` class D could not be obtained due to problems with their execution. The NAS benchmarks do not contain an implementation for `IS` class D.

B. Discussion of Results

Figure 5 shows a clear trend for short and long-lived applications, which are discussed below.

- For the long-lived benchmarks like BT, SP and FT the performance improvement increases with larger class sizes. The results discussed in Section III-D on cache behavior corroborate these findings that added concurrency can result in lower working set sizes and better cache locality for general programs with a mix of communication and computation. Also, as shown in Section III, these benefits outweigh the extra overheads due to messaging and context switches.
- We achieve substantial performance improvements of 30% for problem sizes like BT-D and SP-D which execute for more than 15 minutes under MPI and 20% improvement for FT-D which executes for around 6 minutes with MPI.
- The performance improvements for long-lived applications were achieved with concurrency per OS-process that ranged from 4 to 16.
- For short-lived applications like (IS, MG, CG), there was performance decrease with added concurrency ranging from 2 to 8. The amount of performance degradation, however, decreased with larger class sizes. Note that IS and MG both lie towards the lower spectrum of short-lived; i.e., their execution times for class B were a fraction of a second and class C less than 2 seconds. Because of the small data size and the relatively short execution time, the opportunity to take advantage of cache locality and smaller message is slight and, as expected, there is only the added costs of the added concurrency. The short execution time magnifies the overheads in terms of relative performance. In absolute terms, the degradation ranges from 0.03 to 1.99 seconds, however, in this case there is no advantage to using added concurrency.
- EP (Embarrassingly Parallel) application is computationally intensive and was mostly agnostic to added concurrency. We saw modest improvements of around 4% with this benchmark.
- LU-C and LU-D were the exception to the performance improvement trend seen for long-lived applications. LU-C which ran in around 18 sec and LU-D which ran in 336 sec (5.6 mins) with MPI, showed a performance decrease of 21% and 1% respectively. The amount of degradation, however, was substantially lower for the larger class D. One reason that LU benchmark behaves differently may be due to that it is more communication intensive [18],



MPI Time (sec)	15.9	63.2	1354.9	10.9	45.6	1021.8	5.5	17.0	365.0	0.7	2.6	41.4	6.6	18.9	333.8	0.47	2.3	0.3	1.6	3.9	9.2
Cores = OxM	64	64	64	64	64	64	64	128	128	128	128	128	128	128	128	32	32	128	128	128	128
P = nfg	4	4	16	4	4	4	4	4	4	8	8	8	2	2	2	2	8	2	4	2	4

Fig. 5. Percentage effect of added concurrency ([P;O;M]) on the end-to-end performance of NAS benchmarks normalized to the best MPI performance (1;O;M). O=8 per M (machine).

compared to others and does not have locality that could benefit from added concurrency. In this case, the added concurrency may give advantages for cache locality but not sufficient enough to overcome the added overhead in messaging for short messages and collectives.

There are two performance benefits to using FG-MPI that are not shown in Figure 5. The first one is due to the decoupling of the problem from the hardware, so for benchmarks like BT and SP that have a restriction that the number of MPI processes be square, it is possible for us to run on a non-square number of cores (e.g. 128) as long as the P×O×M product is a square. The performance improvements with [P;8;16] over MPI [1;8;8] go up significantly, ranging from 36.7% to 63.5%, with SP and BT, class sizes C and D.

Another interesting result is for the `CG` benchmark. We noticed FG-MPI can achieve performance comparable to MPI for CG while using fewer number of cores. That is, FG-MPI provides better resource utilization while achieving similar execution times.

CLASS		P=nfg	O×M	Time (sec)
A	MPI	1	32	0.14
	FG-MPI	2	32	0.17
B	MPI	1	128	3.92
	FG-MPI	2	64	4.03
C	MPI	1	128	9.17
	FG-MPI	2	64	9.35

TABLE VI
COMPARING MPI AND FG-MPI FOR CG BENCHMARK.

Table VI shows the results for `CG`, where FG-MPI uses half the number of cores (O×M) as compared to MPI. This indicates that communication costs may be the limiting factor for CG and keeping it more localized helps.

In conclusion, for the benchmarks considered, added concurrency showed substantial benefits for the long-lived applications and larger problem sizes. For short-lived applications and more communication intensive applications FG-MPI did not provide any improvements. FG-MPI makes it easy to add concurrency to the runtime execution of MPI programs, since it is a runtime parameter, it can be adjusted, without re-compilation, to different problem sizes. Large amounts of concurrency is possible, which may be useful for larger problem sizes, and one can always omit the `nfg` option when its use is not indicated. For existing MPI program FG-MPI provides a flexible runtime parameter to optimize cache locality and message-size independent from the number of cores and machines in the cluster.

V. RELATED WORK

There is past work on thread-based implementations of MPI which introduce their own runtime system. Prior to 1999, there were some projects [19], [20], [21], [22] that focused on thread-based MPI implementations targeted for multi-programmed shared memory environments. Some of the thread-based implementations required special prefixes for MPI routines [20] and none were actively supported and have remained incomplete.

The main focus of thread-based work was to represent an MPI process as a thread with the objective of reducing communication and context switching overhead among MPI processes on a single machine. These early attempts at using threads did not scale and were designed to run as a single Unix process. None of the thread-based approaches have reported experimental results scaling beyond tens of MPI processes. Much of this research focused on thread synchronization and locking mechanisms and avoidance of race conditions. Our approach in FG-MPI is very different since our use of non-preemptive scheduling avoids many of the synchronization and locking overheads that arises with threads [23].

A recent paper [1] studied the issue of over-subscription on multicore. Although they report a 10% performance degradation for MPI they were able to obtain some benefit with over-subscription from using MPI and UPC, using pthreads instead of OS-processes. This was only used to obtain a modest amount of over-subscription (2 or 4) and, as discussed

in their paper, a problem with increasing the amount of over-subscription is that the OS scheduler is not aware of the cooperative nature of the parallel application on a dedicated machine. "Over-subscription" in FG-MPI is done through a user-level scheduler integrated with the MPICH2 progress engine and the effects of the non-aware OS scheduler is minimized by not over-subscribing cores.

Where possible, we have compared FG-MPI to AMPI [2] since it is the closest in terms of providing a relatively complete implementation of MPI that supports added concurrency. AMPI is built on top of Charm++ [24], an object oriented system based on C++, which uses remote method invocation to pass messages between processes. There are benefits to implementing MPI on top of a more flexible runtime system like Charm++ since it becomes possible to support process migration, which is the main focus of AMPI. FG-MPI takes a completely different approach that integrates added concurrency directly into MPICH2. This approach reduces the overheads that arises from over-decomposition so that the benefits of adding concurrency can hopefully be extended to more problems. Static load-balancing is possible in FG-MPI since the programmer has complete control on where and how much concurrency to use during execution, as yet, we do not have process migration capability to support dynamic load-balancing.

VI. Conclusions

FG-MPI extends the MPICH2 implementation of MPI to make it possible to have multiple MPI processes inside an OS-process. FG-MPI achieves the following. (a) Decouples the notion of a process from that of hardware and makes it possible to adjust the granularity of programs independently from the hardware. (b) Reduces the overhead of adding concurrency by integrating the FG-MPI runtime into the MPI middleware. (c) Provides a single unified programming model as an alternative to the hybrid MPI+X model. We measured the benefits of added concurrency for cache awareness and message size and described ways in which FG-MPI minimizes the resulting overheads for context switching and communication on multicore machines. Using the NAS benchmarks we showed that substantial performance gains are possible on some benchmark programs where the gains out-weigh the overheads. We discussed trade-offs between adding concurrency and communication to better understand which programs could make effective use of added concurrency. Runtime support for added concurrency in MPI programs helps raise the level of abstraction for MPI and makes it possible to change the execution behavior of the program to take advantage of the characteristics of machines like multicore architectures.

Acknowledgements: We wish to thank Stas Negara, who answered questions about Photran, which was used in privatizing the variables in NAS to run with FG-MPI.
This work was supported in part by the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

References

[1] C. Iancu, S. Hofmeyr, F. Blagojevic, and Y. Zheng, "Oversubscription on multicore processors," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, april 2010, pp. 1 –11.

[2] C. Huang, O. Lawlor, and L. V. Kale, "Adaptive MPI," in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing LCPC 03*, 2003, pp. 306–322.

[3] S. Negara, G. Zheng, K.-C. Pan, N. Negara, R. E. Johnson, L. V. Kale, and P. M. Ricker, "Automatic MPI to AMPI Program Transformation using Photran," in *3rd Workshop on Productivity and Performance (PROPER 2010)*, no. 10-14, Ischia/Naples/Italy, August 2010.

[4] Argonne National Laboratory, "MPICH2: Performance and portability," MPICH2 flyer at SC07. Available from www.cels.anl.gov/events/conferences/SC07/presentations/mpich2-flyer.pdf.

[5] D. Buntinas, G. Mercier, and W. Gropp, "Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the Nemesis communication subsystem," *Parallel Comput.*, vol. 33, no. 9, pp. 634–644, 2007.

[6] H. Kamal, S. M. Mirtaheri, and A. Wagner, "Scalability of communicators and groups in MPI," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 264–275.

[7] 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," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 53–79, Feb. 1992. [Online]. Available: http://doi.acm.org/10.1145/146941.146944

[8] L. Soares and M. Stumm, "Flexsc: flexible system call scheduling with exception-less system calls," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8.

[9] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: scalable threads for internet services," in *SOSP '19*. New York, NY, USA: ACM, 2003, pp. 268–281.

[10] E. Toernig, "Coroutine library," Available from http://www.goron.de/~froese/coro/coro.html.

[11] D. Libenzi, "Portable coroutine library," Available from http://www.xmailserver.org/libpcl.html.

[12] G. C. Hunt and J. R. Larus, "Singularity: rethinking the software stack," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 37–49, Apr. 2007.

[13] F. Alted, "Why modern cpus are starving and what can be done about it," *Computing in Science and Engineering*, vol. 12, pp. 68–71, 2010.

[14] Intel, "Intel VTune Amplifier XE," Available from http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[15] N. Nethercote, "Cachegrind: a cache and branch-prediction profiler," Available from http://valgrind.org/docs/manual/cg-manual.html.

[16] R. C. Whaley, "Automatically Tuned Linear Algebra Software (ATLAS)," Available from http://math-atlas.sourceforge.net/.

[17] N. A. R. Center, "Numerical aerodynamic simulation (NAS) parallel benchmark (NPB) benchmarks," Available from http://www.nas.nasa.gov/Software/NPB/.

[18] F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler, "Architectural requirements and scalability of the nas parallel benchmarks," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '99. New York, NY, USA: ACM, 1999. [Online]. Available: http://doi.acm.org/10.1145/331532.331573

[19] E. Demaine, "A threads-only MPI implementation for the development of parallel programs," in *Proceedings of the 11th International Symposium on High Performance Computing Systems*, 1997, pp. 153–163.

[20] P. Bhargava, "MPI-LITE: Multithreading support for MPI," Available from http://pcl.cs.ucla.edu/projects/sesame/mpi_lite/mpi_lite.html, 1997.

[21] K. Shen, H. Tang, and T. Yang, "Adaptive two-level thread management for fast MPI execution on shared memory machines," in *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM, 1999, p. 49.

[22] H. Tang, K. Shen, and T. Yang, "Compile/run-time support for threaded MPI execution on multiprogrammed shared memory machines," *SIGPLAN Not.*, vol. 34, no. 8, pp. 107–118, 1999.

[23] R. Thakur and W. Gropp, "Test Suite for Evaluating Performance of MPI Implementations That Support MPI_THREAD_MULTIPLE," in *Proc. of the Euro PVM/MPI*, September 2007, pp. 46–55.

[24] L. V. Kale and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," University of Illinois at Urbana-Champaign, Champaign, IL, USA, Tech. Rep., 1993.