Optimizing a Parallel Runtime System for Multicore Clusters: A Case Study *

Chao Mei
Department of Computing Science
University of Illinois at Urbana-Champaign
chaomei2@illinois.edu

Gengbin Zheng
Department of Computing Science
University of Illinois at Urbana-Champaign
gzheng@illinois.edu

Filippo Gioachin
Department of Computing Science
University of Illinois at Urbana-Champaign
gioachin@ieee.org

Laxmikant V. Kalé
Department of Computing Science
University of Illinois at Urbana-Champaign
kale@illinois.edu

ABSTRACT

Clusters of multicore nodes have become the most popular option for new HPC systems due to their scalability and performance/cost ratio. The complexity of programming multicore systems underscores the need for powerful and efficient runtime systems that manage resources such as threads and communication sub-systems on behalf of the applications.

In this paper, we study several multicore performance issues on clusters using Intel, AMD and IBM processors in the context of the CHARM++ runtime system. We then present the optimization techniques that overcome these performance issues. The techniques presented are general enough to apply to other runtime systems as well. We demonstrate the benefits of these optimizations through both synthetic benchmarks and production quality applications including NAMD and ChaNGa on several popular multicore platforms. We demonstrate performance improvement of NAMD and ChaNGa by about 20% and 10%, respectively.

1. INTRODUCTION

Multicore clusters based on various architectures have become the most popular option for new HPC systems due to their scalability and performance/cost ratio. From Terascale to Petascale and beyond, the number of components (cores, interconnect, storage) within such HPC systems is growing enormously. It is estimated that in the near future very large multicore clusters with core numbers in the range of one-hundred thousand to one million and more will appear. It is obvious that these highly parallel systems will raise questions about parallel software development and especially how to write parallel software that runs efficiently on these multicore architectures.

Threads are considered a convenient and efficient mechanism to exploit multicore nodes. However, dealing with shared process resources with locking and serialization creates a great burden on application developers. This may limit the scalability when not properly handled. Further, a real problem that multicore programming faces is the limited bandwidth of memory access. Efficient handling of data locality is often the key to optimal performance. High level parallel programming languages with underlying runtime systems that can effectively manage resource such as threads and communication sub-systems and exploit data locality in the multicore environment on behalf of the applications are desirable for application developers.

Several popular options for programming multicore systems are available. Some are MPI on distributed memory, OpenMP [1] on shared memory, and MPI+OpenMP [2] on hybrid shared/distributed memory architectures. Additionally, Partitioned Global Address Space (PGAS) languages, such as Unified Parallel C (UPC) [3], are emerging alternatives that allow shared memory-like programming on distributed memory systems. Performance studies [4] have shown that MPI is still the best performer in a hybrid setup of multicore clusters when compared with OpenMP [1] and UPC [3] thanks to MPI's efficient handling of data locality. Another example is CHARM++ [5], which supports migratable object-based programming model via a powerful runtime system (RTS) for various hybrid shared/distributed memory architectures. With the encapsulation of data and work in objects, CHARM++ naturally promotes the data locality needed to exploit multicore architectures.

Using the CHARM++ runtime system as an example, we address the following key questions: 1) What are the issues that impact multicore/SMP performance inside a runtime system? 2) How can we optimize a runtime system to overcome these issues? 3) How do these optimizations work on different architectures?

In this paper, we identify several performance issues that one may encounter when optimizing a runtime system in the scenario of multicore systems, and propose a series of optimizations to overcome the problems found. The techniques we use include: a scheme based on CPU affinity to improve the locality of memory access and reduce the conflict in performing load balancing; a lock free scheduling scheme that uses memory fences to minimize contention among threads; and techniques to handle processor private variables using Thread Local Storage (TLS). These techniques work on a variety of multicore architectures through a portable API we defined. While the techniques employed to overcome the performance bottlenecks are relatively simple, we demonstrate significant performance improvement using the proposed techniques.

The remainder of the paper is organized as follows. Section 2 presents an overview of CHARM++ for multicore clusters. In Sec-

*This work was supported in part by the NSF Grant OCI-0725070 for Blue Waters, and by the Institute for Advanced Computing Applications and Technologies (IACAT).

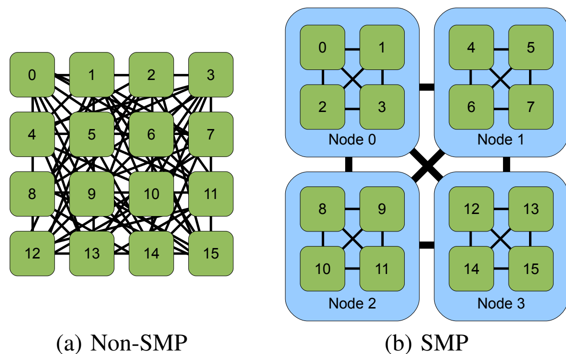
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

tion 3 we illustrate a series of performance issues we encountered when optimizing CHARM++'s runtime system, and the corresponding techniques we used to overcome these problems. A synthetic benchmark is used as an example. Later, in Section 4, we demonstrate the performance improvement using two real-world applications, NAMD and ChaNGa. We discuss some related work in Section 5, and conclude with future plans.

2. ARCHITECTURE OVERVIEW OF CHARM++ FOR MULTICORE CLUSTERS

CHARM++ [6] is a parallel programming system based on a message-driven migratable-objects programming model. In this model, the programmer decomposes his application into fine grain objects, called *chares*, which perform the computation, and communicate through asynchronous method invocation by sending each other messages. These chares can be grouped into *arrays* for ease of indexing, and are free to migrate between processors when needed, for example, for load balancing purposes. Over the years, it has been successfully used to develop several highly scalable parallel applications, such as NAMD [7], ChaNGa [8], and OpenAtom [9]. Furthermore, CHARM++ is a highly portable parallel runtime system, and is deployed on the vast majority of existing parallel machines, many of which are provided by TeraGrid resources.

In many programming models, such as MPI, there is no inherent concept of SMP. To take advantage of the shared address space provided by multicore machines, the programmer has to explicitly consider it in his code, for example by using the OpenMP programming model or threads directly. In contrast, in CHARM++ the user can take advantage of the shared address space without any modification to the source code. This is provided transparently by the runtime system itself. Shared variables are not a problem since in CHARM++ the programming model is object oriented, and global variables are inherently deprecated (CHARM++ provides *readonly* variables as a substitute).



(a) Non-SMP (b) SMP

The traditional way of running programs is shown in Figure 1(a)—which we call "non-SMP". Here, in an application running on 16 processors, each processor communicates with others via an external communicator. Communicators range from TCP or UDP connections to high speed interconnects such as Infiniband or LAPI. Note that processors are represented by system-level processes, and every time a message is sent, the message must be transmitted through the kernel to the other process address space. On a multicore-based cluster node where multiple processes share the physical memory, POSIX shared memory can be used to transfer messages more efficiently between the communicating processes within the node while the data still ought to be copied. This mode of CHARM++ is still categorized as "non-SMP" but with a different name as "PXSHM" which will be used in the rest of the paper. This PXSHM mode is fundamentally inferior, because it gives up many potential advantages of shared memory: read-only data exists as multiple copies, for example, and node-level shared data structures are not possible.

A more efficient way to handle communication in CHARM++ is to use the configuration represented in Figure 1(b). In this configurations, which we call "SMP" mode, the user still allocates 16 processors, but these are collected in groups of four to form *node*s. In this scenario, a node is internally represented by a system process, and each "processors" by a system-level thread. A node shares the same address space and therefore allows a very fast zero-copy message sending. When communicating to processors outside the shared address space, external communicators are used. Depending on the interconnection used, communication may be channeled through a single dedicated *communication* thread, or handled directly by each *worker* thread. Note that the number of external connections is reduced.

The combination of the object-based programming model and the SMP runtime makes CHARM++ a potentially suitable programming model for exploiting shared memory multicore nodes. Using objects, it respects data locality that improves cache performance. It supports node level shared data structures, read-only variables, and communication primitives that are designed to take advantage of the shared memory.

However, our first experience with CHARM++ on SMP led to the same conclusion that others have about MPI: using a separate process for each core was faster than using the SMP version [4]. For example, NAMD's performance decreased by about 10% on SMP version, while ChaNGa's performance decreased slightly by about 2% as we will see in Section 4. This motivated the work in this paper to investigate the performance issues in the CHARM++ runtime. As we will see, the culprit was not the application, but the runtime system itself.

This experience led us to embark on a series of analysis and optimizations that are the topic of this paper. We hope and expect that the optimizations we carried out are of use to other runtime systems, as well as to those applications that must deal with shared memory nodes explicitly.

3. OPTIMIZATION TECHNIQUES FOR CHARM++ RUNTIME SYSTEM

Optimizing the CHARM++ runtime system consists of two aspects: 1) performance on a single multicore node, and 2) performance on a cluster of nodes. This paper mainly focuses on the first aspect. During our analysis, we have extensively studied various synthetic benchmarks and real-world applications. However, due to space constraints, we will only use one synthetic benchmark as an example to walk through the series of optimization techniques we developed. Performance on real applications will be presented in the next section.

The synthetic benchmark we used in this section is "kNeighbor", a program written in CHARM++ using the *chare arrays* construct. kNeighbor creates a certain number of objects distributed on the parallel machine, and arranged in a 1-dimensional array. In each iteration, each chare element sends a message to its k neighbor chares on both sides in a wraparound fashion. When an object has received all the expected messages ($2 * K$), it proceeds to the next iteration. Throughout this section, k is set to 3 and the number of chare elements is equal to the number of cores used so that every core has exactly one element. The iteration time reported is averaged over ten thousand iterations.

The multicore platforms we studied are shown below:

- A: AIX 6.1/IBM Power 5, a 16-core (SMT=2) node

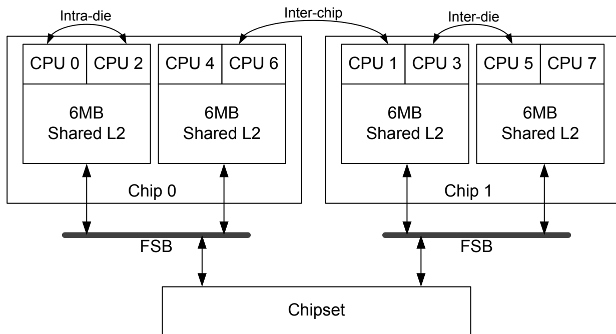


Figure 1: The architecture of the representative node and the CPU topology viewed by Linux.

- B: Ubuntu 8.04/Intel Nehalem Xeon E5520, a 8-core (SMT=2) node
- C: Ubuntu 8.04/Intel Harpertown Xeon E5405 , a 8-core node
- D: Ubuntu 8.04/AMD Barcelona Opteron 2356, a 8-core node
- E: CentOS 5.4/Intel Dunnington Xeon E7450, a 24-core node

Such hardware configurations for a compute node are quite common in TeraGrid machines. Since the result were similar for all the architectures, we will refer only to platform C in our discussions, but also report results on the other platforms when necessary. The eight cores of the representative platform C are illustrated in Figure 1 together with their topology. We categorize data transfers between CPUs as "intra-die", "inter-die" and "inter-chip", ordered by increasing data transfer latency. In our experiments, we left one core free to accommodate noise from OS daemons.

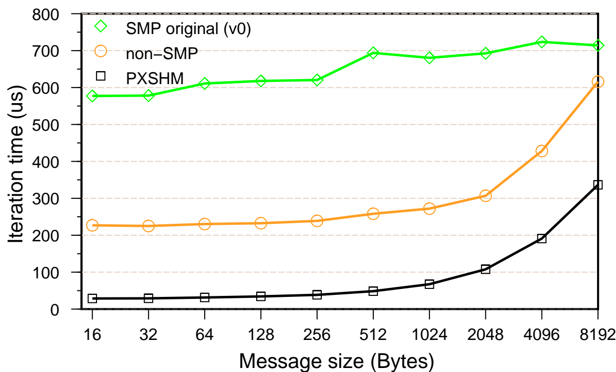


Figure 2: Initial Comparison between non-SMP and SMP.

We demonstrate the impact of each optimization technique by comparing the performance before and after the optimization is applied. Figure 2 shows the performance comparison between the non-SMP modes (non-SMP and PXSHM) and the very initial SMP mode in CHARM++ for the kNeighbor benchmark running on 7 cores with varying message sizes. Clearly, the performance of SMP mode lags far behind that of the non-SMP modes. This was a surprise, as we expected the SMP mode to outperform non-SMP modes due to the faster intra-node communication. Nevertheless, several other bottlenecks were preventing SMP from performing well.

3.1 Network Progress Engine Issues

Parallelizing the CHARM++ runtime system for multicore not only involves engineering the whole runtime to be thread safe, but also invites us to re-think the trade-off of dividing work responsibility between worker and communication threads. As a design choice, each worker thread calls the network progress engine after sending a message to make sure the outgoing message gets put on the network right away, and to check for incoming messages to improve responsiveness. Another choice is to let each worker thread put outgoing messages in a communication thread's waiting queue, and let the communication thread handle the message at a later time. This second choice reduces the contention of having multiple threads access the network engine, but at the cost of potentially increasing message latency.

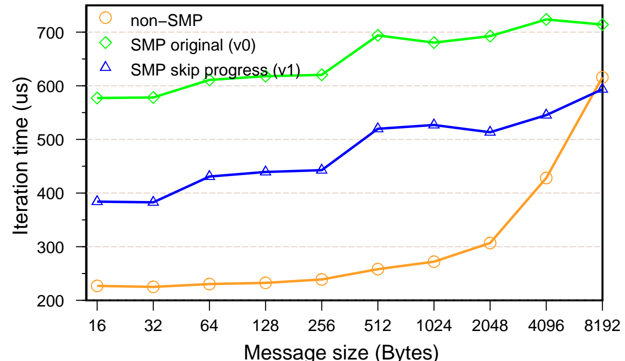


Figure 3: Performance before/after skipping network calls.

In the original implementation, which used the first design, we observed a very high overhead due to expensive network progress engine calls. As a trade-off, we decided to use a combination of the two schemes: we call the network progress engine only for inter-node messages, and skip the calls for intra-node messages. The results of using this optimized scheme as compared to the original scheme are shown in Figure 3. On average, we see about 35% improvement. With this optimization, kNeighbor performs better in SMP than in non-SMP for message sizes larger than 8KB. The SMP mode is still worse for message sizes below 8KB, leading to the further investigation.

3.2 Multi-threaded Performance Issues

We looked at performance issues involved in the conventional multi-threaded programming pitfalls as we optimize the intra-node communication. In retrospect, we found that efficiently handling locking and synchronization among threads in the runtime is the key factor for obtaining fast fine grained intra-node communication.

3.2.1 *Memory Management*

CHARM++ implements its own memory allocators for various reasons including being SIGIO safe in an interrupt mode, and supporting special purpose memory allocation schemes such as isomalloc [10]. The default memory allocator in the CHARM++ was based on a GNU memory allocator extracted from Linux glibc source seven years ago. Lacking of robust support for multi-threading at that time, we "protected" every GNU *malloc* and *free* call with a lock in CHARM++ SMP. It was no surprise that we found that kNeighbor performed poorly because of severe lock contention due to message allocation, amplified by the intensive communication. As multi-threading support in the memory allocator of glibc has now become good enough to be directly used in CHARM++, we switched to it in CHARM++ SMP. The second curve in Figure 4 illustrates kNeighbor's improved iteration time. We see an improve-

#thds	A(us)	A/M(us)	B(us)	C(us)	D(us)	E(us)
1	1.06	1.03	0.78	0.80	1.13	0.68
2	2.23	1.02	1.30	1.44	1.53	2.03
4	6.06	1.05	3.95	2.14	2.36	2.73
8	15.35	1.03	8.71	3.69	4.72	7.06
16	36.89	1.06	22.63	n/a	n/a	14.58
24	n/a	n/a	n/a	n/a	n/a	21.31
32	210.96	1.02	n/a	n/a	n/a	n/a

Table 1: Memory allocation time on different platforms.

ment of 2.4 fold on average after switching to the OS malloc, which is significant.

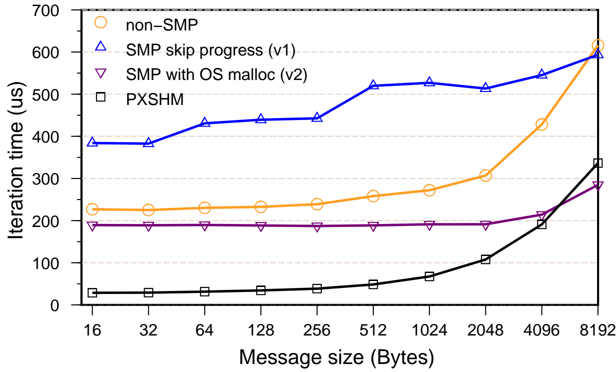


Figure 4: Performance comparison before/after using OS-provided memory allocator.

The above finding intrigued us to find how the OS-provided memory module performs when the number of threads increases in a process. We synthesized a benchmark that every thread continuously allocates memory of the same size and deallocates at the end for 100,000 times. We run the benchmark with the varying number of threads, and take the average memory allocation time on 5 different multicore platforms mentioned earlier.

Table 1 shows that the default OS memory allocators tend not to scale when number of threads increases up to the maximum number of logical CPUs on each platform. Among these, the AIX 6.1/Power 5 platform, illustrated by the second column, did the worst with the default setting. Setting system environment variable "MALLOCMULTIHEAP" under AIX, however, improves the performance significantly as shown in the third column which shows no contention at all. The trick here is to use multiple heaps, one serving several threads to reduce contention on heap allocation. Although this is at the cost of increased overall memory usage, it is a good optimization to consider.

We should be cautious to interpret these results as real applications may not be memory intensive. However, it very likely implies the upper-bound of overhead, and motivates the adoption of a better memory management on the Linux platform for multi-threaded programs, which will also benefit CHARM++ RTS.

It is obvious based on Figure 4 that kNeighbor in the current SMP mode becomes much better after switching to OS-provided memory module (third curve vs. first curve), even exceeds the performance in the non-SMP mode (second curve). However, compared with PXSHM (fourth curve), SMP mode is still worse (by 4.3 times on average) for message sizes below 4KB. We continued this journey by challenging ourselves to beat the performance of the PXSHM mode.

3.2.2 Granularity of Critical Sections

It is really enticing to use a big critical section to make runtime thread-safe, which however leads to a serialization of program and poor performance. In our development for SMP mode, we had also made such mistakes from the perspective of performance. After carefully reviewing the code base of CHARM++ runtime, we removed the locks that are unnecessary and reduced the granularity of critical sections to minimize synchronization overhead, e.g., we put the lock only around the part of function that is not thread-safe instead of blindly putting it around the whole function body. Indeed, this is a trade-off between productivity and performance, because larger critical sections tend to be safer, while reducing their scope requires analyzing complex interactions for race conditions, and tedious debugging efforts. However, for a runtime that is at the foundation of a parallel programming system, and one that is used very often, the effort to improve performance is worthwhile.

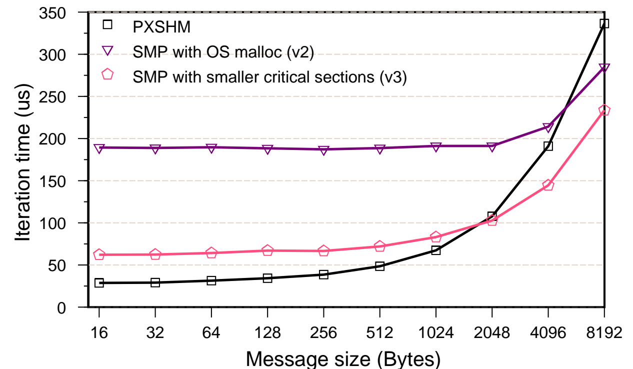


Figure 5: Performance before/after reducing granularity of critical sections.

This turned out to be a significant performance improvement as indicated by Figure 5. Compared with the previous SMP version, kNeighbor benchmark speeds up by an average of 2.7 times up to message size 2KB. Beyond that, we can observe a trend that the performance gain is diminishing, but still with 35.1% improvement on average for message sizes 4KB and 8KB. We believe such performance trend is caused by the fact that the execution time of kNeighbor begins to be dominated by touching every byte of the message on the receiver side. Comparing with the PXSHM version, we can see now the two performance lines cross at a message size smaller than 2KB. Additionally, it is clearly shown the PXSHM mode has a steeper execution time increase rate than that in SMP mode because we cannot avoid an extra copy from the POSIX-shared memory region to the user space for receiving the message, while such a copy is not needed in SMP mode as only the message pointer is passed to the receiver.

3.2.3 Message Queues

Producer-Consumer Queue (PCQueue) is a commonly used data structure in parallel language RTS to synchronize multiple threads. For example, it is used in the Cilk [11] scheduler for work-stealing. CHARM++ RTS uses PCQueue for various purposes to synchronize the worker threads and communication thread with messages. For example, the communication thread as a producer pushes a message into a worker thread's message queue for processing. The simplest way to ensure correctness is to enforce a lock every time a thread accesses the queue. This was the original implementation in CHARM++, and it suffered severe thread contention, especially when the number of producers increased.

Lock free implementation is possible for PCQueue, and has been used in the past [12, 13]. In CHARM++, each PCQueue always has a single consumer (the thread for which the messages are destined), but can have multiple producers (all the other cores in the node). Our first change was to use memory fences to preserve the correctness of the PCQueue operations while allowing producers and consumers to overlap. To take care of the multiple producers scenario, we still used a lock shared among the producers. Memory fence operations, however, are highly architecture specific. To simplify the implementation of PCQueue, we defined a portable API that consists of two functions: CmiMemoryReadFence() and CmiMemoryWriteFence(), which serialize the load and store operations respectively. These two APIs call platform specific memory fence instructions, for example *lfence* and *sfence* on X86-based platforms, *mf* for IA64 platforms, and *eieio* for PowerPC platforms. We use the lock-based scheme as a fallback implementation for the cases when memory fence is not supported. Replacing locks with inexpensive memory fence operations does improve the performance as shown in the second curve in Figure 6. We observe up to 9.7% improvement, especially for messages smaller than 2KB, compared with the performance before using memory fences, as represented by the top curve.

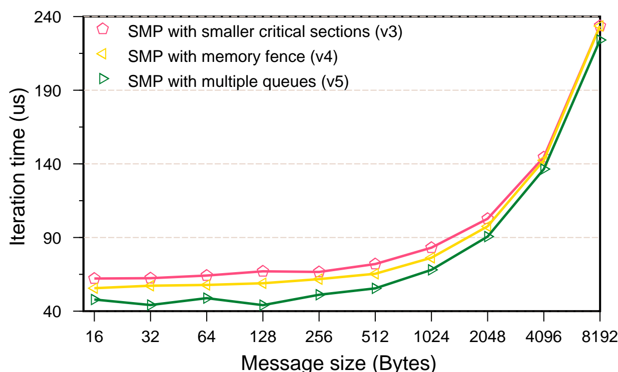


Figure 6: performance comparison with memory fence and multiple queues respectively.

Further, we looked at removing the lock still remaining on the producers side of the queue. While a totally lock-free implementation is possible for multiple-producer-single-consumer, the overhead associated with this implementation was significantly high, and the implementation was not stable, especially on some architectures. Therefore, we removed the lock by introducing multiple queues, one for each producer and consumer pair, so that the pure lock-free PCQueue could be used. Clearly, this optimization comes at the cost of a consumer having to poll all the queue pairs, which can be as many as the number of cores on a node. Therefore, the overhead increases as the size of a multicore node grows. This is observed in our experiments. On the representative platform C, we see that the benefit of removing locks outweighs the incurred polling overhead (about 19.5% improvement on average in kNeighbor benchmark), which is illustrated in the bottom curve of Figure 6. On the other hand, on platform E with 24 cores, kNeighbor shows no speedup at all with this optimization due to the higher polling overhead. As cluster nodes become increasingly large, further investigation are required to reduce the overhead of producers locks.

3.3 Handling Processor Private Variables

Similar to the thread private variables used in OpenMP, CHARM++ uses processor private variables that is used in its runtime implementation and user applications. Applications do not need to be changed to execute in the SMP mode, thanks to the annotation of variables for distinguishing between processor private and shared variables (noted by Cpv and Csv macros respectively). For example, "CpvAccess(*var*)" is to access a processor private variable. In non-SMP mode, Cpv variables are defined same as Csv variables, because there is only one single thread. In SMP mode, however, Cpv variables are done differently, which have one dedicated copy for each thread, similar to *thread* private variables.

Our initial solution (which was developed more than 5 years ago) is to use an array of size equal to the number of "processors" on a node to represent the processor private variables. Each rank of processors uses its own copy of the variable in the array, so accessing a processor private variable *var* is expanded to "*var*[*myrank*]". This solution has a significant disadvantage of cache "false sharing" in SMP mode, as we observed using Intel Performance Tuning Utility (PTU).

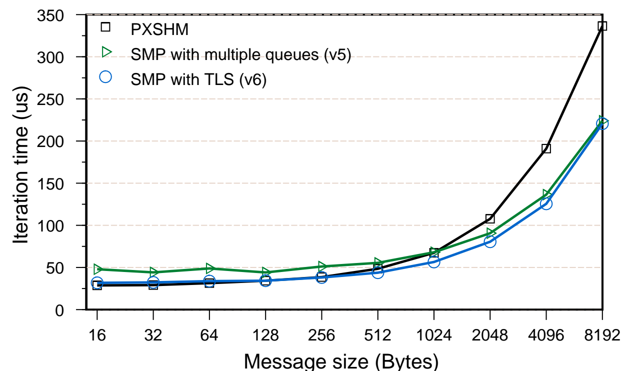


Figure 7: Performance before/after using TLS.

To avoid this, we used the thread local storage (TLS) scheme either implicitly if the "__thread" keyword is supported by the compiler and assembler, or explicitly through function calls such as "pthread_setspecific/pthread_getspecific" on Unix-like platforms or "TlsSetValue/TlsGetValue" on Windows. To set an idea of how TLS performs against the array-based solution, we evaluated the time taken to update a processor private integer variable on the 5 platforms mentioned before Section 3.3 with 8 "processor"s (i.e. 8 threads in total on the node). Table 2 illustrates the significant advantage of TLS over the array-based solution because the latter suffers greatly from cache false sharing induced by the cache-coherence protocol.

Platform	A (ns)	B (ns)	C (ns)	D (ns)	E (ns)
TLS	0.40	1.27	1.5	1.75	1.26
Array-based	51.58	17.52	10.03	9.61	8.50

Table 2: Time of updating processor private variable on different platforms.

Figure 7 shows kNeighbor's performance improvement by 26.5% on average after switching to TLS scheme for processor private variables. We can also see a decreasing return for this optimization for messages of relatively larger size due to the reason mentioned in the end of Section 3.2.2. In addition, we have moved the performance of SMP mode closer to that of PXSHM mode in that PXSHM only outperforms SMP for very small messages below 128B.

3.4 CPU Affinity

We noted that the way operating systems binding SMP threads to cores, and how they move the threads between cores has great impact on the multicore performance.

There has been extensive research on the scheduling algorithms in operating systems for multcore systems to improve overall performance by using process and thread affinity [14]. However, optimizations performed in this category tend to improve the overall performance of a multicore system and its utilization, they may not benefit the particular application of concern. Instead of being intrusive, most operating systems on multicore systems adopt soft affinity, also called natural affinity, which is the tendency of a scheduler to try to keep processes on the same CPU as long as possible. However, this is merely an attempt; if it is ever infeasible, the processes certainly will migrate to another processor. For example, using the same kNeighbor benchmark, we observe that the OS keeps changing the core of a particular thread on a 8-core machine, as shown in Figure 8.

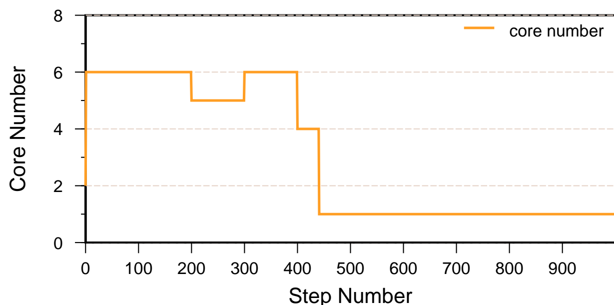


Figure 8: OS keeps changing the core of a thread.

Our first experiment was to see how performance is affected when fixing threads to their cores. There are a few reasons doing this. First, CPU affinity can optimize cache performance, since it avoids the OS moving a thread to a core which has cold cache. When threads bounce between cores, they constantly cause cache invalidation. Therefore, in performance critical situations, it makes sense to enforce the affinity as a hard requirement. Secondly, OS moving threads around may conflict with application load balancing effort. Moving threads and their work unexpectedly could result changing of the already balanced load.

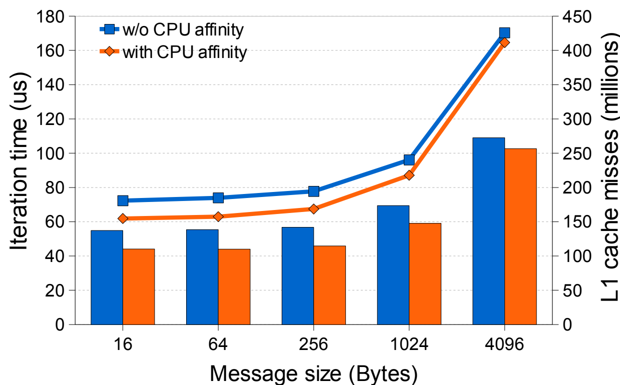


Figure 9: kNeighbor L1 cache misses and iteration time.

The result of fixing threads to cores is shown in Figure 9 (lines) using the same kNeighbor benchmark running on a 8-core multi-core desktop using various message sizes. We observe up to 15% performance improvement in the total execution time by just doing that! To better understand this, we measured the L1 cache misses for the same runs, as illustrated in the same Figure (bars). We see that for small messages, the number of L1 cache misses are reduced by around 20% by binding threads to their cores, while the reduction of cache misses decreased to around 10% for larger messages. We also observed similar performance boost with several other applications such as NAMD as well. This demonstrated that simply enforcing hard thread affinity is beneficial to applications.

With these encouraging results, we further consider the effect of affinity bindings on application performance. Intuitively, mapping communicating threads to closer cores in the memory hierarchy incurs less data transfer latencies between cores, which leads to better overall application performance.

The impact of bindings on performance really depends on the communication pattern of the application. For the kNeighbor benchmark when k=3 running on 7 cores of a 8-core machine, performance does not vary by the different bindings. This is because each element communicates with all 6 other elements on 6 different cores, making bindings unimportant. However, when k=1, every element only communicates with its two neighbors, the binding shows significant impact on the performance. For example, when message size is 256 bytes, a mapping of 0,1,2,3,4,5,6 yields iteration time of 13.37 us, while a mapping of 0,2,4,6,1,3,5 yields iteration time of 11.66 us.

The execution time difference is primarily due to the different number of inter-chip, inter-die and intra-die messages. In the case of mapping 1, there are 4 inter-die messages, and 24 inter-chip messages per iteration in total. In comparison, the second mapping caused fewer inter-chip messages (reduced from 24 to 8) with cheaper messages of the other two types (increased from 4 to 8, and 0 to 12 respectively). Therefore, the overall performance of the second mapping is better.

We extended CHARM++ RTS with a portable function API (`CmiSetCPUAffinity`) to allow programmers to manually bind pthreads to processors. The implementation uses the low level system call to bind threads, for example:

- `pthread_setaffinity_np` for Linux and pthreads,
- `bindprocessor` for IBM AIX, and
- `SetThreadAffinityMask` for Fibers on Windows.

In the future, we plan to extend this work with an automatic and adaptive affinity binding scheme at runtime. We also plan to study the effect of thread affinity on CHARM++'s load balancing.

3.5 Other Issues

We further analyzed the fine grained message performance using Intel PTU, trying to find the most expensive instruction blocks in terms of CPU cycles. We identified that the push/pop operations on the message queue still constituted a high overhead. These operations, despite being simple and short, still had very high cycles per instruction because they contain multiple memory accesses which are particularly expensive due to frequent execution. This overhead manifests itself when message sizes are small. We simplified the data structure of the message queue to reduce the number of memory accesses. Consequently, as demonstrated in Figure 10, the kNeighbor in SMP mode improved by 8.1% on average for messages up to 1KB. We have omitted data points from message size 2KB because this optimization shows negligible improvement due to the fact of touching message data mentioned in the end of Section 3.2.2. Compared with the PXSHM mode, it now performs

equally well for very small messages and much better for message sizes beyond 512B (due to the copy-free message delivery in SMP)!

Although this is demonstrated on an Intel architecture, we found our optimization generally helps kNeighbor on other platforms we have access to.

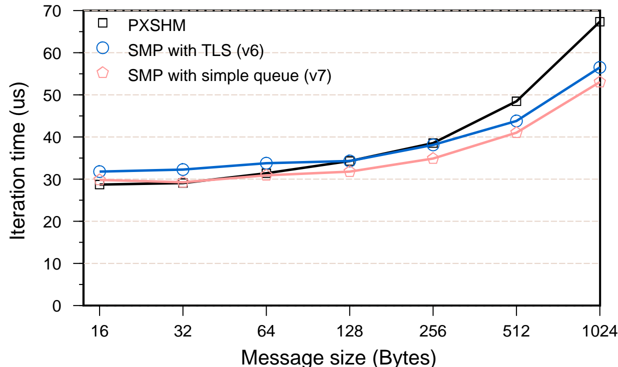


Figure 10: Performance before/after using simple PCQueue.

Up to this point, it is a triumph that SMP mode of CHARM++ achieves the best performance over the wide range of message sizes. It beats both the plain non-SMP mode and the PXSHM mode by about 486.6% and 20.7% respectively. Amazingly, we have obtained about 14.4 fold performance improvement for kNeighbor from the initial SMP implementation of CHARM++.

4. APPLICATION CASE STUDIES

In addition to synthetic benchmarks, we used two production-level scientific applications to demonstrate the performance impact of our optimization techniques. These two applications are NAMD and ChaNGa.

4.1 NAMD

NAMD [7] is a scalable parallel application for Molecular Dynamics simulations written using the CHARM++ programming model. It is used for the simulation of bio-molecules, and to understand their function. Figure 11 shows the performance results of NAMD for the standard benchmark Apolipoprotein-A1 (ApoA1) molecule system on two multicore platforms C and E described in the Section 3. On these two platforms, performance changes in NAMD, due to the switch in CHARM++ runtime mode from non-SMP to SMP and the aforementioned series of optimization techniques, are quite representative. On a platform having a smaller number of cores per node such as C, NAMD in SMP mode is better by 5.2% than it is in non-SMP mode. In contrast, on the platform that has a larger number of cores per node such as E, NAMD in SMP mode demonstrates more benefits as it beats the non-SMP one by 21.1%. Such difference in performance improvement is primarily due to the difference in the number of cores in the node. Since the SMP mode reduces the message latencies significantly within a node, a larger node size, implying more chances for an application to have messages sent within a node, will benefit more as demonstrated in this case. In addition, we can see from the figure that the optimization of having multiple queues mentioned in Section 3.2.3 incurs a slightly performance degradation for NAMD on platform E because of the increased polling overhead for message queues. Finally, we noticed that using the OS-provided memory management instead of the old memory module as mentioned in Section 3.2.1 alone contributes the most performance gains for both platforms.

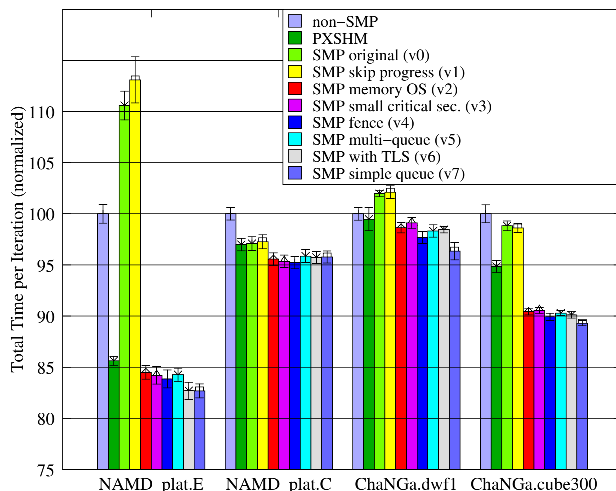


Figure 11: Applications Performance.

4.2 ChaNGa

ChaNGa [8] is a cosmological application used for the simulations of the evolution of the universe. It handles forces generated by both gravitational and hydrodynamic interaction. ChaNGa considers SMP optimizations at the application level, and makes extensive use of CHARM++ *nodegroups*. With these optimizations, during the computation of the forces it leverages the shared memory and avoids all intra-node communication. Communication is present during other phases of the iterative process, such as during construction of the global tree.

Figure 11 shows also two executions of ChaNGa with two different datasets on platform C. The first dataset consists of nearly five million particles highly clustered in the center of the simulation (dwf1); the second consists of about 110,000 particles uniformly distributed in space (cube300). The first system takes about 500 seconds to perform 3 iterations, while the second requires about 30 seconds to perform 5 iterations of the algorithm, and is more communication intensive. We can see that the performance of SMP was worse than non-SMP with the initial SMP version. As in NAMD, switching to the OS memory system provides the greater benefit to ChaNGa. This is due to two reasons: 1) since all threads allocate memory at the same time, by releasing the locks the total time spent allocating memory is greatly reduced; 2) since the memory is allocated from separate pools for different threads, the resulted memory blocks are less spread in the address space, and accessing it is faster.

For the first dataset, the performance benefit is only 4% from non-SMP mode (6% from the original SMP implementation). This is mainly due to the fact that the majority of the time is spent computing forces, without any communication. For the smaller dataset, which constitutes a more typical computation/communication ratio when scaling simulations to large machines, the improvement is 6% from non-SMP mode with POSIX shared memory, and 11% when the processes are communicating through the OS kernel.

We can see that by simply switching from non-SMP (with or without POSIX shared memory) to SMP mode the performance is automatically boosted by 9%. When applying the optimizations described in this paper, another 2% is gained. As mentioned, ChaNGa has many internal optimization for SMP mode, and therefore benefits less from the optimizations of the runtime system. However, the improvements just mentioned are still very significant considering

that the application did not need to be changed to obtain them.

5. RELATED WORK

Through running benchmarks, applications and simulations, studies have been conducted to evaluate the impact of current multicore chips for cluster computing, and identify the performance issues that include optimizing intra-node communication and relieving cache and memory contention[15][16]. This paper also describes such performance issues, but in the context of a parallel language RTS. In addition, we applied optimization techniques to actually resolve those issues and demonstrated their effectiveness.

Our work to optimize the CHARM++ runtime system shares a common purpose with new implementations[17][18][19] for MPI communication subsystem in that both utilized the shared memory of the compute node in clusters to optimize message latencies within a node. But the latter ones are achieved by using the existing[17] or newly[18][19] developed inter-process memory transfer techniques, while ours is achieved naturally by having threads instead of processes on each core despite the performance issues in shared-memory programming as described in the paper.

Much work has been done for parallelizing or optimizing user-level applications for multicore systems such as the work on lattice Boltzmann computation[20]. Such work probably shares a similar engineering process as we went through in this paper. However, since the language RTS is coupled with applications, the development of general optimization techniques for it becomes more complex and is likely to require more efforts.

6. CONCLUSION AND FUTURE WORK

In this paper, we described various performance issues we encountered when optimizing CHARM++ RTS on multicore systems. Many of these issues are common to other multi-threaded runtime systems. We present a series of optimization techniques that overcome these performance issues, focusing mainly on optimizing intra-node fine grained communication. The techniques apply to a variety of multicore architectures and operating systems.

In the future, we would like to extend our optimization techniques to improve inter-node communication performance on multicore clusters for scalability. We will also continue to explore the techniques of using CPU affinity to further improve the multicore performance. One idea is to develop adaptive load balancing strategies that take SMP memory hierarchy into account to minimize expensive data movement and communication between cores.

7. REFERENCES

[1] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Science & Engineering*, vol. 5, no. 1, January-March 1998.

[2] L. Smith and M. Bull, "Development of mixed mode mpi / openmp applications," *Scientific Programming*, vol. 9, no. 2-3/2001, pp. 83–98.

[3] T. El-Ghazawi and F. Cantonnet, "Upc performance and potential: a npb experimental study," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–26.

[4] D. A. MallÃşn, G. L. Taboada, C. Teijeiro, J. TouriÃśo, B. B. Fraguela, A. GÃşmez, R. Doallo, and J. C. MouriÃśo, "Performance evaluation of mpi, upc and openmp on multicore architectures." in *PVM/MPI*, ser. Lecture Notes in Computer Science, M. Ropo, J. Westerholm, and J. Dongarra, Eds., vol. 5759. Springer, 2009, pp. 174–184.

[5] L. V. Kale and G. Zheng, "Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects," in *Advanced Computational Infrastructures for Parallel and Distributed Applications*, M. Parashar, Ed. Wiley-Interscience, 2009, pp. 265–282.

[6] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[7] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale, "Overcoming scaling challenges in biomolecular simulations across multiple platforms," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008.

[8] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn, "Massively parallel cosmological simulations with ChaNGa," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[9] E. Bohm, A. Bhatele, L. V. Kale, M. E. Tuckerman, S. Kumar, J. A. Gunnels, and G. J. Martyna, "Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L," *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, vol. 52, no. 1/2, pp. 159–174, 2008.

[10] G. Zheng, O. S. Lawlor, and L. V. Kalé, "Multiple flows of control in migratable parallel programs," in *2006 International Conference on Parallel Processing Workshops (ICPPW'06)*. Columbus, Ohio: IEEE Computer Society, August 2006, pp. 435–444.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *J. Parallel Distrib. Comput.*, vol. 37, no. 1, pp. 55–69, 1996.

[12] H. Massalin and C. Pu, "A lock-free multiprocessor os kernel," 1991.

[13] M. M. Michael and M. L. Scott, "Fast and practical non-blocking and blocking concurrent queue algorithms," in *Proc. 15th ACM Symp. on Principles of Distributed Computing*, 1996, pp. 267–275.

[14] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–11.

[15] A. Kayi, T. El-Ghazawi, and G. B. Newby, "Performance issues in emerging homogeneous multi-core architectures," *Simulation Modelling Practice and Theory*, vol. 17, no. 9, pp. 1485 – 1499, 2009.

[16] L. Chai, Q. Gao, and D. K. Panda, "Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system," in *Seventh IEEE International Symposium on Cluster Computing and the Grid - Table of Contents*, 2007.

[17] D. Buntinas, G. Mercier, and W. Gropp, "Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem," in *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, Washington, DC, USA, 2006, pp. 521–530.

[18] L. Chai, P. Lai, H.-W. Jin, and D. K. Panda, "Designing an efficient kernel-level and user-level hybrid approach for mpi intra-node communication on multi-core systems," in *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, Washington, DC, USA, 2008, pp. 222–229.

[19] R. Brightwell, "Exploiting direct access shared memory for mpi on multi-core processors," *Int. J. High Perform. Comput. Appl.*, vol. 24, no. 1, pp. 69–77, 2010.

[20] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Optimization of a lattice boltzmann computation on state-of-the-art multicore platforms," *Journal of Parallel and Distributed Computing*, vol. 69, no. 9, pp. 762 – 777, 2009.