TiNy Threads on BlueGene/P: Exploring Many-Core Parallelisms Beyond The Traditional OS

Handong Ye, Robert Pavel, Aaron Landwehr, and Guang R. Gao
University of Delaware
Department of Electrical and Computer Engineering
Newark, Delaware
{handong, pavel, alandweh, ggao}@capsl.udel.edu

Abstract

Operating Systems have been considered as a cornerstone of the modern computer system, and the conventional operating system model targets computers designed around the sequential execution model. However, with the rapid progress of the multi-core/many-core technologies, we argue that OSes must be adapted to the underlying hardware platform to fully exploit parallelism. To illustrate this, our paper reports a study on how to perform such an adaptation for the IBM BlueGene/P multi-core system.

This paper's major contributions are threefold. First, we have proposed a strategy to isolate the traditional OS functions to a single core of the BG/P multi-core chip, leaving the management of the remaining cores to a runtime software that is optimized to realize the parallel semantics of the user application according to a parallel program execution model. Second, we have ported the TNT (TiNy Thread) execution model to allow for further utilization of the BG/P compute cores. Finally, we have expanded the design framework described above to a multi-chip system designed for scalability to a large number of chips.

An implementation of our method has been completed on the Surveyor BG/P machine operated by Argonne National Laboratory. Our experimental results provide insight into the strengths of this approach: (1) The performance of the TNT thread system shows comparable speedup to that of Pthreads running on the same hardware; (2) The distributed shared memory operates at 95% of the experimental peak performance of the machine, with distance between nodes not being a sensitive factor; (3) The cost of thread creation shows a linear relationship as threads increase; (4) The cost of waiting at a barrier is constant and independent of the number of threads involved.

1. Introduction

Operating Systems are a cornerstone of modern computing systems, powering virtually all general-purpose computers today. The basic functions of contemporary operations systems are based on fundamental research from the 1960s and 1970s where the target machines were sequential computers based upon a sequential execution model (the von Neumann model) developed in the 1940s. We have witnessed the rapid progress of the multi-core/many-core chip technology that allows a parallel computer system to be designed and implemented on a chip. To this end, we argue that the conventional OS model must adapt to the changes of underlying hardware platforms so as to exploit the many levels of parallelism in both hardware and software.

Modern supercomputers, like Blue Gene/P (BG/P), utilize high throughput networks of lower frequency processors to reduce power consumption. The execution model of this many-core architecture is drastically different from the sequential execution model used to develop the traditional OS. As such, this type of architecture provides many challenges for the traditional OS model. One such challenge is balancing the interference of OS scheduling and interrupts with computation. Many methods have been proposed to adapt the traditional OS model to account for these issues. One method is to modify the Linux kernel by using a variety of approaches. For example, reducing TLB misses, so that the OS noise can be reduced while keeping the abundant flexibilities of Linux. ZeptoOS [1] is an example of this. Another method is to replace Linux with a very lightweight runtime kernel such as IBM's Compute Node Kernel (CNK) on BG/P [2]. CNK removes the virtual paging related issues by statically mapping the virtual address into TLB. This is in addition to other proprietary methods that

IBM utilizes. Generally, these approaches try to reduce the OS noise from the kernel side without requiring awareness of the programming model. However, we believe the root cause of OS noise is that current runtime systems do not cooperate with the program execution model.

Another issue is that such systems lack a universal address space between nodes, meaning that internode communication typically occurs via message passing. Message passing provides another detail that the programmer must be aware of and cater the algorithm to in order to maximize performance. For example, programmers must decide the size of each MPI task and distribute the tasks among different nodes while considering the load balance. Moreover, programmers need to consider data locality issues and also need to change the program to overlap the communication and computation in order to further improve performance.

As the originator of this paper, we believe that neither the programming model nor the kernel alone can fully exploit the opportunities for parallelism and performance in high performance computing. To do this, we isolate the traditional OS functions, specifically those not directly relevant to parallelism (such as I/O), to a single core of the BG/P node. All other cores are dedicated to parallel computation and are entirely managed by the runtime software that is optimized to realize the parallel semantics of user applications according to our parallel program execution model. We have ported the TNT (TiNy Thread) execution model from IBM's Cyclops 64 [3] and it's runtime libraries to allow for further utilization of the BG/P compute cores. Finally, we have expanded the design framework described above to a multi-chip system designed for scalability to a large number of nodes.

An implementation of our method has been completed on the Surveyor BG/P machine operated by Argonne National Laboratory [4]. Our experimental results indicate the feasibility and strengths of our approach: (1) The performance of the TNT thread system shows comparable speedup to that of Pthreads running on the same hardware; (2) The distributed shared memory operates at 95% of the experimental peak performance of the machine, with distance between nodes not being a sensitive factor; (3) The cost of thread creation shows a linear relationship as the number of threads increase; (4) The cost of waiting at a barrier is constant and independent of the number of threads involved.

The organization of this paper is as follows. Section 2 introduces the basic concepts of the original TNT model. Section 3 explains the detailed design of our system. It covers the three levels of thread management, the design of the runtime kernel, and the shared memory layer based upon message passing. Section 4 presents the results of our experiments. Section 5 thanks those who assisted us. And section 6 presents our conclusions.

2. Background of TNT

TNT presents a thread virtual machine which was originally introduced by Juan D. Cuvillo [3], [5] with the goal of replacing the conventional OS with a non-intrusive runtime system. The TNT thread virtual machine consists of a thread model, memory model and synchronization model. The programming model is designed with this execution model in mind. The TNT programming model provides an API that is very similar to Pthreads but utilizes non-preemptive threads. Because of these similarities, many Pthreads applications can be ported by simply changing the prefixes of function calls and removing unneeded parameters.

3. System Description

Our design allows the programmer to utilize a distributed computer as though it were an SMP machine by transparently mapping the traditional parallel programming model to the underlying distributed systems. The initial overall design was built on BG/P. A software stack was created which maps the TNT model to BG/P. The kernel handles TNT operations on each Compute Node.

Our system is built on the BG/P compute nodes which are used for user computation work. Each compute node has four PowerPC 450 processors sharing a 2GB memory. Between different compute nodes the communication passes through the Torus network. IBM provides the Deep Computing Messaging Framework (DCMF) library for communication on the network.

The software stack was designed to allow the TNT programming model to be applied to BG/P. At the top of the software stack is the user application which uses the programming interfaces provided by the TNT library. Below the application layer resides the TNT library which provides threading, synchronization, shared memory, and utility system calls. The library manages threads across all Compute Nodes, handles synchronization, and provides a global address space. The software layer that provides the global address space is named the TNT Distributed Shared Memory (TDSM) and is built on top of DCMF. This in turn is built on top of the BG/P System Programming Interface (SPI), which closely interacts with the kernel.

The kernel handles lower level operations on a Compute Node.There are four processors in a Compute Node. Processor zero runs a version of ZeptoOS [6] modified to support TNT threads. In BG/P, each Compute Node needs a set of control and I/O processes to communicate with I/O nodes. By design, processor zero runs the control and I/O service processes to communicate with I/O nodes. It also performs management tasks, such as thread-scheduling and synchronization. TNT threads will be scheduled run on the three remaining processors. Because threads are non-preemptive, they will execute to completion on the processor they are being executed on.

The thread model will be explained in section 3.1, the kernel design that makes this possible in section 3.2, the memory model in section 3.3, and the synchronization model in 3.4. Lastly, we will describe the interface in section 3.5.

3.1. System Overview

Our system is divided into four key components. The first is the thread management system. The second is the underlying kernel that allows TNT to run on BG/P. Third is TDSM, which provides a shared memory that is required for the TNT execution model. Fourth is the synchronization constructs, which provide determinism.

3.1.1. Thread Management Overview. From the user perspective, thread management follows the tradional Pthreads interface. Our system internally manages the distribution of threads across the system without user intervention.

In order to schedule threads across nodes, we designate a core on each node to handle internode communication. The details of communication depend on the thread scheduling model being used and are covered in the following subsection.

Within a node, the kernel executes the non-preemptive threads. Details on local thread management are covered in the corresponding subsection.

3.1.2. Global Thread Management. We have implemented multiple thread scheduling models so as to support the greatest variety of programs. Because of the wide variety of parallelization techniques, as well as the added costs of communicating across the torus network, we felt that the optimal approach was to provide the user with a choice of scheduling methods. We provide two types of scheduling, Workload-Aware scheduling and Workload-Unaware scheduling. In Workload-Aware scheduling, threads are assigned to nodes based upon the current distribution of threads in the system. In Workload-Unaware scheduling, the current state of the system is irrelevant to the assignment of threads. We feel that these methods are sufficient for benchmarking the system, but additional methods can be added at a later date. The scheduling method used is selected at compile time, thus allowing users to determine the scheduling method most beneficial to their program.

The first scheduling model is the Workload-Unaware model. When tnt_create() is called, the requesting thread will select a node to spawn and execute the thread. When tnt_join() is called, the joining thread will contact the spawning node to alert it to a request to join, and join if the thread has finished executing. When tnt_exit() is called, the spawning thread will check to see if a join request has already been made, and if so, proceed to signal the joining thread that the join is complete. Otherwise, it will finish executing. The benefit of this model is that the communication overhead is minimal. The only communications are between the requesting thread and the spawning thread, and the joining thread and the spawning thread.

We provide two methods of scheduling with this model; Round-Robin and Random assignments. The former should only be used in programs in which the core running the main function spawns all threads, and all threads run for the majority of the program. The latter allows programs in which threads spawn additional threads to benefit from the minimized communication overhead while still having a more even distribution of work.

The second scheduling model is the Workload-Aware model. When tnt_create() is called, the requesting node contacts the global management node which then consults a table to determine which node has the fewest threads assigned to it at any given time. That node then spawns and executes the thread for the requester. When tnt_join() is called, the joining thread will contact the global management node and request to join. When tnt_exit() is called, the spawning thread will contact the global management node and indicate that the execution has completed. Once both of these have occurred, the global management node will signal the joining thread.

As mentioned before, there is no perfect model for all parallel programs. For coarse-grained parallelism, the Workload-Aware model will provide better performance due to the impact of thread creation and synchronization being small compared to computation. However, with a large number of nodes, the Workload-Aware model will present a bottleneck. On the other hand, the Workload-Unaware model can suffer from

starvation.

3.1.3. Local Thread Management. On the individual node level, the local thread management system provides efficient local thread scheduling for the global thread management system. Any thread creation requests received on a node are handled by the kernel via a thread creation system call, which causes the kernel running on the node to schedule the thread. When a thread exits, the kernel handles cleanup. In order to minimize OS noise, we move the task of local creation and thread management to the kernel and employ a non-preemptive thread scheduling algorithm.

Other optimizations are used in order to reduce the synchronization cost of scheduling on a node. For instance, the TNT queue is designed as a per processor data structure with a lock for each queue. When a Local Thread Manager creates a thread, it reads the status of each TNT queue without lock protection and writes the thread information into the appropriate queue with lock protection. This avoids the need for synchronization when reading the status of the queues, but introduces the possibility that the queue status may be altered between the time the status of the queue was read and the time the queue was updated. This is an acceptable tradeoff because it removes the need for synchronization when reading the status of the queue.

3.2. Kernel Design

Processors 1 through 3 run a light weight runtime kernel, which takes thread information from the thread queues and sets up the process context for the new TNT thread and then context switches to the thread. After the thread ends, it calls tnt_exit() and switches back to the kernel.

Currently, TNT threads are created using Linux's expensive clone mechanism. To minimize the cost, we reuse the process context by leveraging the non-preemptive feature of the TNT thread model. Non-preemption guarantees that there is, at most, one TNT thread running on a processor at any given time. Because all of the TNT threads share the same program image, we preserve the process context after the thread ends. In each processor we only clone the application program image when the first tnt_create() is invoked, and the context is destroyed only when the program ends.

3.3. TNT Distributed Memory

We have built a shared memory layer that encapsulates distributed memory and the underlying message passing mechanisms in order to provide the programmer with a shared memory view. This allows the programmer to leverage the usability of traditional shared memory parallel programming models.

TDSM was designed with accessibility as the primary objective, but we also wanted to minimize communication costs. Our overall design presents how we achieved this. First, the library provides users with a single logical address space that is translated into physical distributed memory addresses for easy pointer use and to minimize read and write costs. Second, a load balancing of communication is used when allocating memory to minimize communication costs. Additionally, load balancing of communication was used for synchronization constructs to further minimize the cost of communications. Lastly, to ease the burden of the programmer, a C compatible memory allocation interface was designed to reserve memory. To do this, we built TDSM on top of IBM's DCMF. We avoid consistency issues by leveraging DCMF's internal memory consistency models.

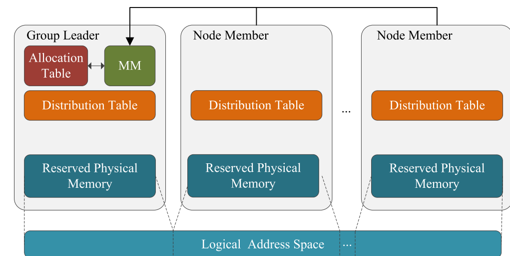

Figure 1. TDSM System

3.3.1. Single Logical Address Space Design. Figure 1 shows the basic components of a single group. There is one group leader with any other number of member nodes within that group. A Memory Management (MM) component is attached to the group leader which handles the allocation requests inside of a group. The reserved physical memory blocks on these nodes make up the logical address space. As such, there are two types of addresses involved: virtual addresses in the process context and logical addresses that reside in the single logical address space provided by TDSM. For convenience, we refer to them as ***virtual addresses*** and ***logical addresses*** respectively.

A distribution table for addressing is created during system initialization. The table contains the information needed to translate the logical address space into virtual addresses across each node. Specifically the table contains an entry for each node, with every entry containing a handle to the reserved memory block and

the size of said block. The entries are in ascending address order and ascending node order. In addition, each node has a copy of the table. These properties allows the programmer to use pointer arithmetic to address memory across all nodes. Additionally, the information in the table allows the programmer to write to any memory location on any node. This ensures that communication happen directly between two nodes, minimizing overhead. The read and write requests can be sent to locations directly using one sided communication and DMA. Accordingly, synchronization costs are removed when reading and writing to memory.

3.3.2. Memory Operations Design. We divide memory operations into two categories: memory allocation and read/write memory operations.

When a thread running on a node requests an allocation, it is sent to the group leader's Memory Manager. The MM performs the memory allocation and returns the logical address. The allocation algorithm used by the Memory Manager can be specified by the user. The Memory Manager also maintains the availability information in the Allocation Table. When a free request is sent to the MM, it simply updates the Allocation Table. If the allocation requests cannot be satisfied inside the group the Memory Manager will transfer the request to the MM in the next group.

When a read/write memory operation is performed on a logical address, the requesting node maps the logical address to the node(s) where the memory block(s) resides and the virtual address(es) inside of the node(s). The requesting node then sends the appropriate request message directly to these nodes. If the operation is a write all receiving nodes will asynchronously write the data to their memory using DMA. If the operation is a read, all receiving nodes will asynchronously read the memory using DMA and send the data back to the requesting node. Then, the requesting node writes all the data it receives into its memory using DMA. Regardless of whether the operation is a read or write, operations involving DMA do not require any interrupting of the computation being done on the receiving nodes.

3.4. Synchronization

The design of synchronization for use with the shared memory focuses on minimizing communication while maintaining similar functionality as conventional synchronization mechanisms. For this reason, we provide two forms of synchronization: mutexes and barriers.

Mutexes function in a similar manner to memory allocation. Each mutex is associated with a Mutex Management (MuM) component. The Mutex Manager handles lock and unlock requests to that mutex from all nodes. Each group has a Mutex manager that manages all mutexes created by threads run on the nodes in that group. When a lock or unlock request is made, a request is sent to the MuM associated with the mutex being accessed. The MuM then responds with the state of the mutex, changing the status from locked to unlocked, or vice versa, as needed.

Barriers operate in a similar manner. Each barrier is associated with a Barrier Management (BM) component. When a barrier is initialized, the BM creates a table of all threads associated with the barrier. When a barrier is reached, each thread sends a signal to the BM stating that the barrier has been reached. Once the Barrier Manager determines that all threads have reached the barrier, the BM broadcasts a signal to allow all threads to continue execution.

3.4.1. Interfaces and Usage. Our interface was designed with the goal of providing users with an interface similar to the one found in traditional C programs while introducing a minimal amount of constraints. We provide users with functions similar to malloc, calloc, and free which have the same parameters as their C counterparts. However, programmers must initialize the total size of shared memory they will use at the start of their program. This constraint allows for the creation of a Distribution Table on each node. Additionally, when reading and writing to shared memory, users are required to use the provided read and write functions.

4. Experimental Results

In this section, we evaluate the efficiency of TNT on BlueGene/P by using a diverse set of benchmarks. We individually tested each major component of the TNT execution model through the use of microbenchmarks. Section 5.1 presents a summary of the results, and the remaining sections expand upon the summary. Section 5.2 focuses on the thread system local to a single node. Section 5.3 discusses the memory system's performance. Section 5.4 focuses on the multi-node thread system. And section 5.5 examines the performance of the synchronization constructs.

The experiments are conducted on the Surveyor BG/P machine at Argonne National Laboratories [4] with up to 1024 nodes.

4.1. Summary of Results

To demonstrate the performance of each of the available thread scheduling algorithms, we benchmarked the cost of the underlying communications of our thread scheduling algorithms. We also benchmarked the performance of the thread system. To measure the performance of the underlying thread model, the only viable direct comparison on BG/P is Pthreads. We intend to compare the performance of the global thread model to that of UPC and MPI, but our implementation as of the writing of this paper still needs additional work.

Observation 1 (See Section 5.2): The performance of the TNT thread system shows comparable speedup to that of Pthreads running on the same hardware.

Observation 2 (See Section 5.3): The distributed shared memory operates at 95% of the experimental peak performance of the machine, with distance between nodes not being a sensitive factor.

Observation 3 (See Section 5.4): The cost of thread creation shows a linear relationship as the number of threads increase.

Observation 4 (See Section 5.5): The cost of waiting at a barrier is constant and independent of the number of threads involved.

4.2. Single-Node Thread System Performance

In this section, we measure the performance of the underlying thread model. To provide good speed-up across multiple nodes, it is first important to provide good speed-up on a single node. As such, we benchmarked the performance of a single node of our system.

The algorithm we used is the Radix-2 Cooley-Tukey algorithm with the Kiss FFT [7] library providing the underlying DFT. As such, this limited the number of usable threads to a power of two. Our benchmark consists of multiple iterations of the FFT for varying sizes of input data. In order to demonstrate the speed-up, we ran each data size with one and two threads. We performed these benchmarks for TNT and the PThread library. The results can be found in Figure 2.

Our results demonstrate that the speed-up provided by our underlying thread model performs comparably to the POSIX standard when the number of threads does not exceed the number of available processor cores. Essentially, these results indicate that our underlying system is comparable to Pthreads.

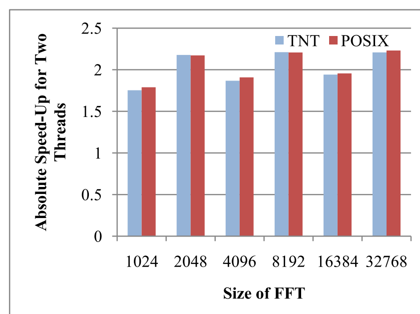

Figure 2. Speed-Up of FFT Benchmark on BG/P

4.3. Memory System Performance

To demonstrate the efficiency and scalability of our memory interface, we utilized microbenchmarks to measure the performance of the core operations of the TDSM.

4.3.1. Peak Performance of Read and Write Operations.

In this section, we measure the efficiency of the underlying communications involved in read and write operations of varying sizes. The communication between two nodes must be efficient so as to minimize latency when communication between more than two nodes is required. To measure the latency of this overhead, read and write operations of varying sizes were performed between two SMP nodes.

The program used for this benchmark consists of a number of operations on shared memory. The execution is timed and divided by the number of memory operations to calculate the estimated latency of the operation. The test is performed for reads and writes of both local and remote access to data in order to provide the best and worst case scenarios for each data size. The results are plotted in Figure 3.

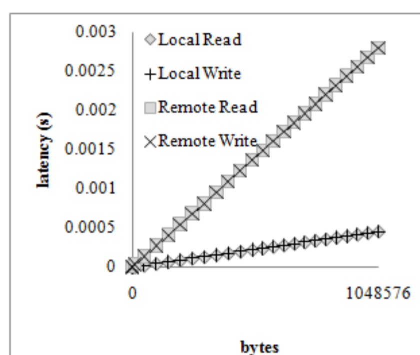

Figure 3. Latency of Read and Write Operations of Varying Sizes

The results demonstrate that the underlying communication used in TDSM is highly efficient. The relationship between the size of the data being read or written and the latency is a linear one, even out to one megabyte. For remote operations, data can be transferred at approximately 357 MB/s. This is compared to the experimental peak performance calculated by Kumar et al. [8] of 374 MB/s for the link between two nearest neighbors in the torus. This means that TDSM operates at approximately 95% of the maximum efficiency. For local operations, the rate is 2.14 GB/s. Furthermore, the latency of a read and write, of similar locality, is effectively equal, thus showing that TDSM performs equally well for both reads and writes.

4.3.2. Scalability of Read and Write Operations. In a system like TDSM, it is important to have low latency when accessing a memory block in a remote node. So in this experiment, we designate one node to read a fixed size of data from each of a number of nodes.

The program reserves a fixed size (1024 bytes) of memory on each node to create a shared logical address space, and then conducts the test using one loop. Each iteration of the loop will perform a read or write with the size of the whole logical address space, and this ensures each node is accessed. The execution time of the loop is recorded and divided by the number of iterations in order to get an average access time.

This benchmark was run for a wide range of nodes and is plotted in Figure 4.

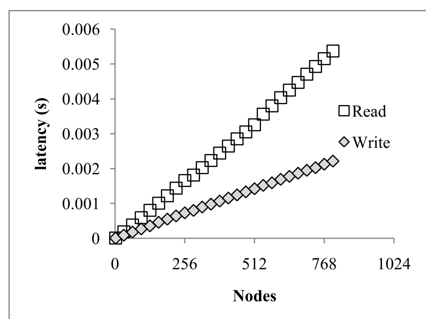

Figure 4. Latency of Read and Write Operations Across Multiple Nodes

The results show that the latency is a linear function of the number of nodes, or in other words, the latency linearly increases as the amount of data increases; therefore node distance is not a sensitive factor.

4.4. Multinode Thread Creation Costs

To demonstrate the performance of each of the available thread scheduling algorithms, we benchmarked the cost of the underlying communications of our thread scheduling algorithms. We intend to compare the performance of the global thread model to that of UPC and MPI, but our implementation as of the writing of this paper still needs additional work.

The microbenchmark consists of a single loop. Threads are created in this loop. The execution time of this loop is measured and divided by the number of iterations. This provides the average cost of thread creation. This microbenchmark was repeated for a varying number of threads. The results can be seen in Figure 5.

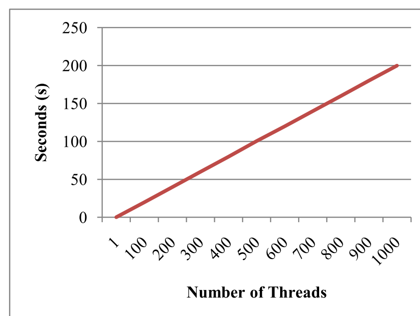

Figure 5. Cost of Thread Creation

It is interesting to note that the thread creation cost, approximately 0.2 seconds per thread, remained effectively constant between scheduling algorithms and when scaled. The latter indicates that the scalability of the system is such that it can be run with a very large number of threads and nodes, which is important for BG/P.

4.5. Synchronization Performance

For the purpose of benchmarking the synchronization model, we measured the performance of the construct most likely to scale poorly, the barrier. For a program to demonstrate high scalability, the performance of synchronization constructs must scale well.

The program used in this benchmark synchronizes a number of threads using a barrier. Each thread then runs a single loop. This loop consists of a single barrier operation. The execution time of this loop is recorded, and the result is divided by the number of iterations of the loop. By executing this loop a number of times, the average cost of a barrier can be measured. The results of this can be found in Figure 6.

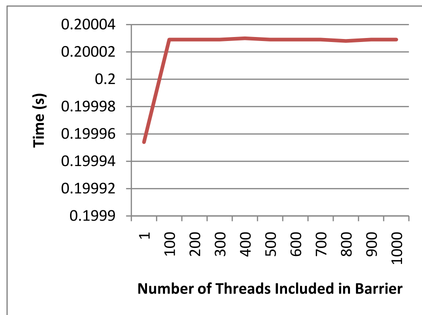

Figure 6. Latency of Barrier Operation

The results show that the performance of the barrier after the first one hundred threads is independent of the number of threads with an effectively constant 0.2 seconds regardless of the number of threads involved. This is important in that it allows our system to full utilize a BG/P machine without adding additional latencies to the synchronization constructs. While it is likely that more complex benchmarks with additional network traffic will have poorer performance, this still demonstrates that the underlying software barrier has minimal latencies.

5. Acknowledgements

This work was supported by NSF (CNS-0509332, CS-0720531, CCF-0833166, CCF-0702244), the Department of Defense, and other government sponsors.

We would like to thank the ZeptoOS team [6] for providing the ZeptoOS kernel for our modifications. In particular, we would like to thank Kazutomo Yoshii for answering our questions. We would also like to thank Argonne National Laboratories [4] for access to their Surveyor machine.

We would also like to thank ET International for providing the TNT kernel.

We thank all the members of CAPSL group at University of Delaware. We thank Joshua Suetterlein for valuable feedback, and we thank Joshua Landwehr for assistance in the implementation of this system.

6. Conclusion

This paper presents an execution model-driven approach to adapting the traditional OS model to many-core architectures. This approach isolates the traditional functions of the OS to a single core leaving the remaining three cores of a BG/P chip for parallel computation. These cores are managed by a runtime system that is optimized to realize the parallel semantics of the user application according to a parallel program execution model. This parallel program execution model is the TNT execution model ported to BG/P. Furthermore, we expanded upon this design to support highly scalable cluster systems.

In order to test the feasibility of our design we benchmarked a number of aspects of the system. First, we demonstrate a highly efficient single node operation comparable to the performance of Pthreads on the same hardware. Second, we demonstrate a distributed shared memory capable of operating at 95% of the experimental peak performance over the BG/P DMA communication layer with the distance between nodes not being a sensitive factor. Third, the cost of thread creation is linear as the number of threads increase. Finally, the cost of synchronization via barriers is constant and independent of the number of threads involved in the system. We believe that our results indicate that our approach is feasible and efficient for operating on a cluster of SMP systems.

References

[1] K. Yoshii, K. Iskra, P. Broekemaand *et al.*, “Characterizing the performance of big memory on blue gene linux,” in *Proceedings of the 2nd International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, 2008.

[2] IBM, *IBM System Blue Gene Solution: Blue Gene/P Application Development*. Vervante, 2008.

[3] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, “Tiny threads: A thread virtual machine for the cyclops64 cellular architecture,” *Parallel and Distributed Processing Symposium, International*, vol. 15, p. 265b, 2005.

[4] “Argonne leadership computing facility.” December 2009, `www.alcf.anl.gov`.

[5] J. del Cuvillo, “Breaking away from the os shadow: A program execution model aware thread virtual machine for multicore architecture,” Ph.D. dissertation, University of Delaware, 2008.

[6] “Zeptoos project.” September 2009, `http://www.zeptoos.org/`.

[7] M. Borgerding, “Kiss fft.” December 2009, `http://sourceforge.net/projects/kissfft/`.

[8] S. Kumar, G. Dozsa, G. Almasi *et al.*, “The deep computing messaging framework: generalized scalable message passing on the blue gene/p supercomputer,” in *ICS ’08: Proceedings of the 22nd annual international conference on Supercomputing*. New York, NY, USA: ACM, 2008, pp. 94–103.