University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

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

Handong Ye, Robert Pavel, Aaron Landwehr, and Guang R. Gao

CAPSL Technical Memo 097
May 31, 2010

University of 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 the 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 manycore 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 the OS due to 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 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 like 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, and each TLB entry can cover up to a 256MB memory block. 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. Specifically in a supercomputer such as BG/P, the OS is not aware of the thread 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, to maximize performance. For example, programmers need to 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 for high end computing. To do this, we isolate the traditional OS functions, specifically those not directly relavent to parallelism (such as I/O), to a single core of the BG/P multi-core chip. The rest of the cores are dedicated to parallel computation and are entirely managed by runtime software that is optimized to realize the parallel semantics of the user applications according to a parallel program execution model. We have designed a way to port the TNT (TiNy Thread) execution model and its compiler/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 chips.

An implementation of our method has been completed on the Surveyor BG/P machine operated by Argonne National Laboratory[3]. 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 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 our weak non-preemptive thread model, an improvement of the historical TNT thread model which required strict non-preemption. Section 5 presents the results of our experiments, section 6 thanks those who assisted us, and section 7 presents our conclusions.

2 Background of TNT

TNT presents a thread virtual machine which was originally introduced by Juan D. Cuvillo[4, 5] with the goal of replacing the conventional OS with a non-intrusive runtime system. The TNT thread virtual machine is based upon the execution model; which consists of a thread model, memory model and syn-

chronization model. The programming model is designed with this execution model in mind. According to TNT thread model, once a hardware thread unit or processor starts running, it will continue until the user explicitly relinquishes control over the thread unit or the program performs an illegal operation. The TNT programming model provides an API that is very similar to PThreads. Because of these similarities, many Pthreads applications can be ported by simply changing the prefixes of function calls and removing unneeded parameters.

Prior to this paper, TNT had only been implemented on IBM's Cyclops64[4] which has a user-accessible global physical address space across all the nodes. As such, there is no need to provide a software module like TDSM (TiNy Threads Distributed Shared Memory). However BG/P is a distributed system, and TDSM is required to make the programming model applicable. Furthermore, since Pthreads is the standard C language threads programming interface used on SMP systems, it stands to reason that a majority of programmers employing SMP systems to parallelize algorithms are familiar with it. For this reason we believe that providing a similar interface across a distributed memory system such as BG/P is the logical step forward.

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 initially maps the TNT model to BG/P. However, at the lower level the kernel handles TNT operations on each Compute Node. Lastly, threads were used to manage the underlying framework across nodes.

Our system is built on the BG/P compute nodes which are used for user computation work. Each compute node has four PowerPC450 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. A number of compute nodes connect to an I/O node according to the configuration. An I/O node is responsible to transfer the communication request/reply to/from the outside where the control system, file system and user frontend reside.

The software stack was designed to allow the TNT model to be applied to BG/P. Figure 1 shows the software stack in our system. We'll explain it from high level to low level. At the top is the user application which uses the programming interfaces provided by the TNT library. Below the application layer resides the TNT library which provides the thread, synchronization, TDSM, 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). The implementation of global thread management and synchronization are built on top of DCMF. The DCMF library that TNT is built upon is the one designed and implemented by IBM and ported by ANL. This in turn is built on top of the BG/P System Programming Interface (SPI), which closely interacts with the kernel. Users can easily write applications based upon the TNT library.

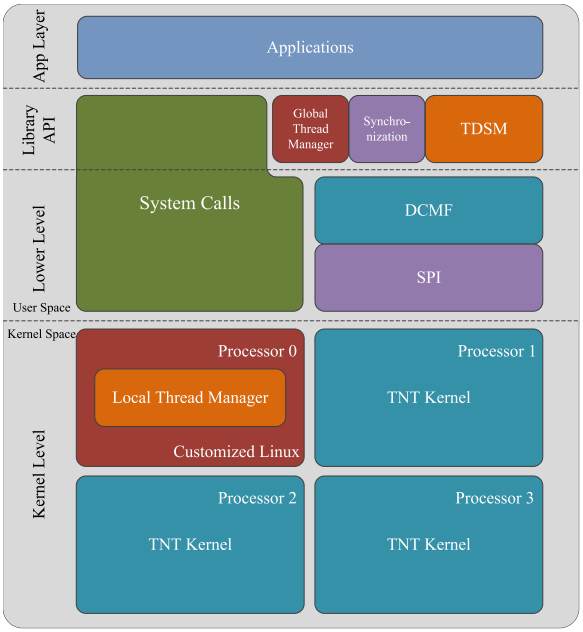

Figure 1: System Overview

The kernel handles lower level operations on a Compute Node. At the bottom of Figure 1 is the kernel on a Compute Node of BG/P. There are four processors in a Compute Node. Processor zero runs a customized version of Linux which originated from ZeptoOS [6] and was 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 in user space to communicate with the I/O nodes. It also performs management tasks which will be further explained. A TNT thread will be mapped to one of the three remaining processors and occupy it until the thread ends. A context switch is required when the processor switches from the TNT kernel to a TNT thread and another context switch is required for the switch back to the TNT Kernel.

Our underlying framework is separated into system threads and user threads. User threads are the threads that are created by the user via an explicit call to the tnt_create function. System threads on the other hand are invisible to the user. Each system thread is assigned one or more system tasks. There are two types of system tasks: global and group. Global system tasks manage across all nodes. These tasks include managing threads and synchronization across all nodes. Group tasks manage within their group. These tasks include TDSM management and load balancing the creation of synchronization primitives. Global tasks only occur on node zero. We divide nodes into groups for all group tasks and assign a leader node. These tasks are only attached to threads on the group leader node. Groups are configurable at compile time.

The thread model will be explained in section 4.1, the kernel design that makes this possible in section 4.2, the memory model in section 4.3, and the synchronization model in 4.4. Lastly, we will describe the interface in section 4.5.

3.1 System Overview

An efficient multi-node thread management and creation system was designed. The overall management system is divided into global and local thread management phases. The global thread management phase acts as a thread scheduler that assigns threads to an SMP node. Next, the local thread manager schedules threads across cores. We first give a brief overview of the whole system. We then go into the details of global and local thread management.

3.1.1 Thread Management Overview

From the user perspective thread management follows the traditional usage of system threads. What this means is that the user can expect our thread management interfaces to behave in the same manner as the Pthread interfaces. Internally, without the intervention of the user, we manage the threads across the underlying distributed systems.

In order to schedule threads across nodes (or globally), 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 subsequent subsection.

Within a node, the kernel handles thread management much like in normal SMP systems with the exception that threads are non-preemptive. In addition, the kernel provides some extra interfaces to allow the global thread management to operate more efficiently. 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 number of threads currently assigned to all nodes 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. All communication is achieved through IBM's DCMF.

The first scheduling model is the Workload-Unaware model. The conceptual layout of this model can be seen in Figure 2. 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

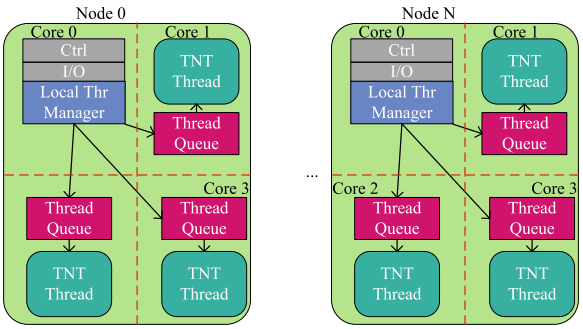

Figure 2: Workload Unaware Model

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 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.

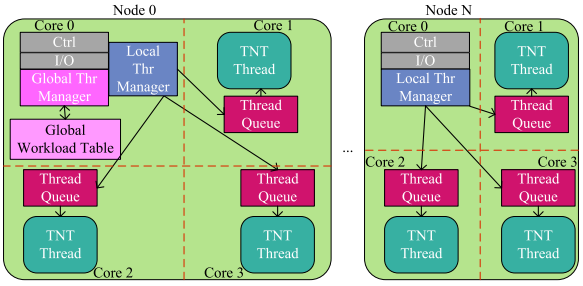

Figure 3: Workload Aware Model

The second scheduling model is the Workload-Aware model. The conceptual layout of this model can be seen in Figure 3. 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.

While the Workload-Aware approach does have a considerably greater communications overhead, approximately 50% more per thread, this model provides better performance for coarse-grained parallelism. For coarse-grained parallelism, the cost of thread creation and synchronization is small compared to the time spent performing computation. As such, the greater cost of thread creation and synchronization is outweighed by being able to evenly distribute the work across the nodes and maximize the number

of threads that are executing non-preemptively and minimizing OS noise.

Furthermore, while the Workload-Unaware models are able to evenly distribute the assignment of threads, they are unable to compensate for varying execution times. Without awareness of the number of threads currently running on each node, threads may be assigned to the weakly preemptive core of a node executing three non-preemptive threads with large execution times while other nodes suffer from starvation. The Workload-Aware model avoids this by assigning threads to the node with the fewest number of threads currently executing.

However, the Workload-Aware model does begin to lose performance when the number of nodes is large. This is because the single global manager acts as a bottleneck. Thus, with a large number of nodes, the bandwidth to the global management node will become saturated, causing further delay. Currently, we are investigating alternatives to provide the benefit of the Workload-Aware model without greatly increasing synchronization and communication overhead.

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. Local thread management provides a number of roles. First, 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. Second, after threads exits, the kernel handles cleanup. By moving the task of local thread creation and management to the kernel we are able to maximize performance by providing a scheduling algorithm with minimal OS noise.

Normal SMP operating systems provide a concept of time-sharing for the scheduling of processes and threads. In time-sharing systems, each processor or thread on a system is given a designated time slice in which to run and then is required to yield the CPU to other threads or processors. In such systems, not only is time wasted on scheduling, but a vast amount of time is wasted context switching between processes and threads. In order to eliminate this unneeded OS noise in our system, we employ a non-preemptive run to completion thread scheduling algorithm; in which, threads that are scheduled on a core stay on that core until they finish running.

Other optimizations are used in order to reduce the synchronization cost of doing the actual scheduling on the node. For instance, the TNT queue is designed as a per CPU data structure, one for each processor, and a lock is used to protect each queue. When the Local Thread Manager decides to create a thread, it reads the status of each TNT queue without lock protection and puts the thread information, PC and arguments, into an appropriate queue with lock protection. In this way, it avoids the need for synchronization when reading the status of the queues, but it introduces the possibility that the queue status may be altered between the time of reading queue status and the time of writing into the queue. This is tolerable if we do not require an optimal balancing of the workload. The worst case occurs if the queue becomes full between the reading of the queue and the insertion of the thread in the queue. In this case, Local Thread Management will wait until the queue is available.

3.2 Kernel Design

Figure 4 shows the basic flow chart in a normal compute node. The left part shows the work flow in processor zero, where Local Thread Management, control and communication processes reside. The kernel in processor zero is a customized Linux kernel originated from ZeptoOS[6]. It's customized as if it's running on a single processor, but it has many supports for the other processors.

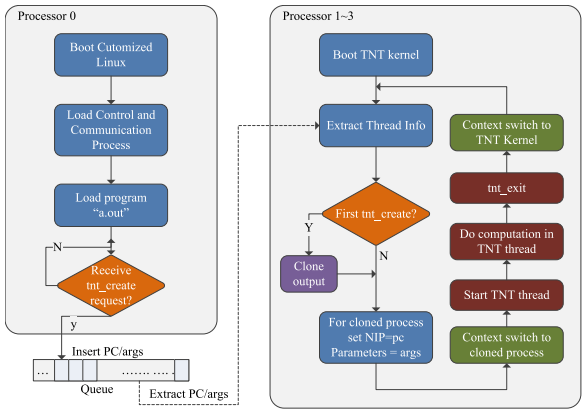

Figure 4: Kernel Flow Diagram

Each of processors 1 3 has a light weight runtime kernel, which basically takes the thread information from the queue, and sets up the process context for the new TNT thread, and switch to the thread. After the thread ends, it calls tnt_exit() and switches back to the kernel.

Currently, TNT threads are created using the expensive clone mechanism in Linux. To minimize this cost, we decided to reuse the process context by leveraging the non-preemptive feature of the TNT thread model. Non-preemption guarantees 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 fact, we only change the NIP (Next Instruction Pointer) to the PC of the second thread and the arguments so that the process context can be reused. So 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. During program execution, the extra cost would be one context switch from TNT kernel to the TNT thread and then another context switch from the TNT thread back to the TNT kernel.

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.

3.3.1 Single Logical Address Space Design

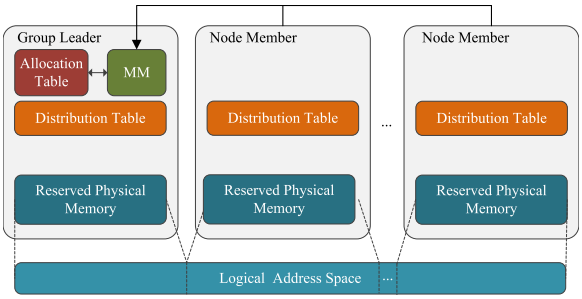

Figure 5: TDSM System

As mentioned before, TDSM also uses the group design employed in thread management. Figure 5 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 three types of addresses involved: physical addresses, 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 ***physical addresses***, ***virtual addresses***, and ***logical addresses*** respectively. Virtual addresses are translated to physical addresses through the OS and TLB. Logical addresses are translated to virtual addresses by TDSM through the Distribution Tables on each node. All the translations are transparent to the programmers.

The null pointer is reserved from the logical addressing so that programmers can handle pointers as in normal C programs. Starting at 0x1, the logical address space sequentially increments to the size specified as the desired amount of shared memory.

Each node in the group contains a copy of the Distribution Table created during the initialization's reserving of memory blocks and mapping of virtual addresses to logical addresses. The table contains the information needed to translate the logical address space to virtual addresses across all nodes. 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. Logical addresses are easily translated with this table. This 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 communications 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. So far we have not considered load balancing for this due to the size of the applications that are currently being run on our system. As mentioned earlier, the number of nodes in a group is specified at compile time. A lower number of nodes in a group leads to less work for each Memory Manager, but the available memory in a group is decreased as a result. Also, Memory Management tasks can only handle one request at a time, so synchronization cost also decrease as the number of nodes increases.

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 interrupting computation on the receiving nodes.

3.4 Synchronization

The design of the synchronization constructs for use with the shared memory focused on minimizing communication costs while providing similar standard C structures: mutexes and barriers.

Mutexes are created and used exactly as in PThreads. 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, the underlying communication sends a request 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

```
#include <tnt.h>
void foo(tnt_chunk_t *p )
{
        int i;
        tnt_lock( &lock );
        tnt_shmem_read( p, &i, sizeof(int) );
        i++;
        tnt_shmem_write( p, &i, sizeof(int) );
        tnt_unlock( &lock );
}
```

(a) Thread Code

```
#include <tnt.h>
int main(int argc, char *argv[])
{
        int i;
        int *counter;
        tnt_desc_t tid[100];
        tnt_chunk_t *glb = NULL;
        tnt_init();
        glb = tnt_shmem_malloc( sizeof(int) );
        Counter = (int*)glb;
        for(int i=0 ; i<100; i++ ){
                tid[i] = tnt_create( foo, glb, NULL );
        }
        for(int i=0 ; i<100; i++ ){
                tnt_join( &tid[i], NULL );
        }
        tnt_shmem_free( glb );
}
```

(b) Main Code

Figure 6: Example TNT Program.

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. The code example in Figure 6 demonstrates the majority of our APIs additions. 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. As the provided example shows, this additional constraint has a minimal impact on the programmer's use of the library. It is hoped that, in the future, these constraints can be handled by a preprocessor as opposed to the programmer.

The most used interfaces in TDSM are listed below.

- tdsm_read A specified block of memory is read from a logical address and written to a local

virtual address.

- tdsm_write A specified block of memory is read from a local virtual address and written to a logical address.
- tdsm_malloc A specified block of memory is reserved in shared memory and the logical address is returned to the user.
- tdsm_free The block of shared memory specified by the given logical address is released.

4 Weak Non-Preemptive Thread Model

The strict non-preemptive thread model introduces deadlock problem in several cases. For example, assuming there are N TNT threads attempting to arrive at a barrier on M processors, with N larger than M. Only M threads have the opportunity to begin execution and they occupy all the processors all the time while waiting for barrier. The remaining N-M threads have no chance to start. In this way, the barrier can never be reached. Another example comes from nested thread creation. Assuming M threads are running on M processors, and each thread wants to create a child thread and waits for the child to end using tnt_join. In this case, the child threads have no chance to access a processor, so the parents must keep waiting and occupying the processors.

The solution we introduced is weak non-preemptive thread model. The key idea of the weak non-preemptive thread model is to allow the movement of TNT threads to the processor where a preemptive linux kernel is running and to make the moved TNT thread preemptable. In the system used on BG/P, processor zero implements a Linux kernel and performs all management tasks; such as thread scheduling and TDSM request management. When all the running TNT threads are at a deadlock, or the number of waiting threads reaches a threshold, the thread manager moves TNT thread requests in the queue to processor zero. This allows for deadlocks due to nonpreemptiveness to be resolved and improves performance by further balancing the workload.

5 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 focues on the thread system local to a single node, Section 5.3 is the memory system's performance, section 5.4 focuses on the multinode 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[3] with up to 1024 nodes. Each node consists of one chip, which has four PowerPC 450 processors, with 2GB off-chip DRAM per node. The PowerPC 450 processor runs at a frequency of 850 MHz. IBM's

DCMF library provides the interface to the torus network for internode communication. There is no shared memory between nodes.

5.1 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 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.

5.2 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 wrote and ran a benchmark on 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. We ran each data size with one thread and two threads, so as to demonstrate speed-up. We performed these benchmarks for TNT and the PThread library. The results can be found in Figure 7.

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. While the performance of our model under thread saturation needs fine-tuning, these results do indicate that our system is comparable to PThreads under normal circumstances.

5.3 Memory System Performance

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

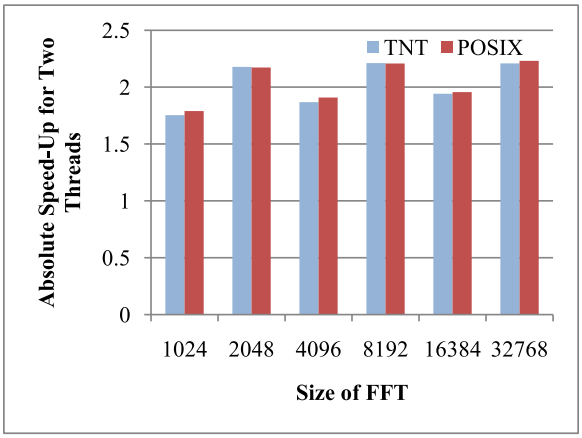

Figure 7: Speed-Up of FFT Benchmark on BG/P

5.3.1 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 one loop. This loop consists of a number of operations on shared memory. This loop is timed, and the results are 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 8

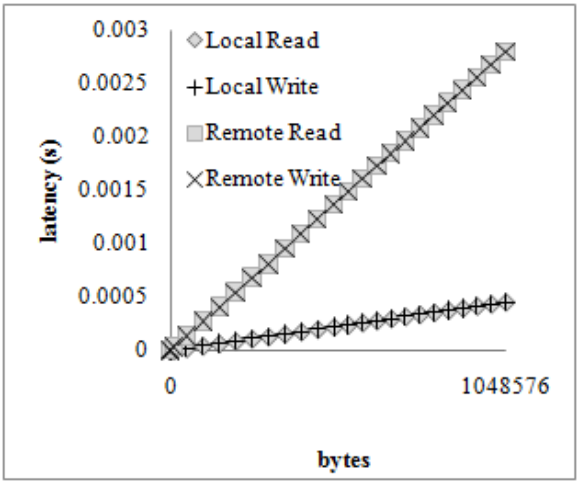

Figure 8: Latency of Read and Write Operations of Varying Sizes

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. This is important in that it indicates that TDSM performs equally for read and write heavy algorithms.

5.3.2 Scalability of Read and Write Operations

In a system like TDSM, it's 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. For example, node 0 reads/writes 1024 bytes from/to the other 1, 2 and up to 1023 nodes. If N nodes involved, the designated node reads or writes $1024 \times N$ bytes.

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 tdsm_read or tdsm_write with the size of the whole logical address space, and this ensures each remote 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 9.

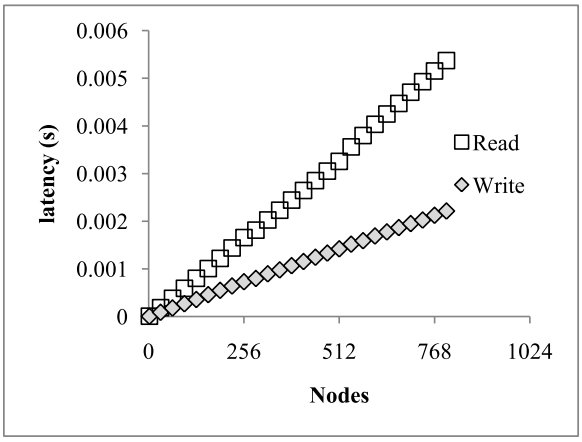

Figure 9: Latency of Read and Write Operations Across Multiple Nodes

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.

5.4 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 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.

The microbenchmark consists of a single loop. In this loop, threads are created. 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 10.

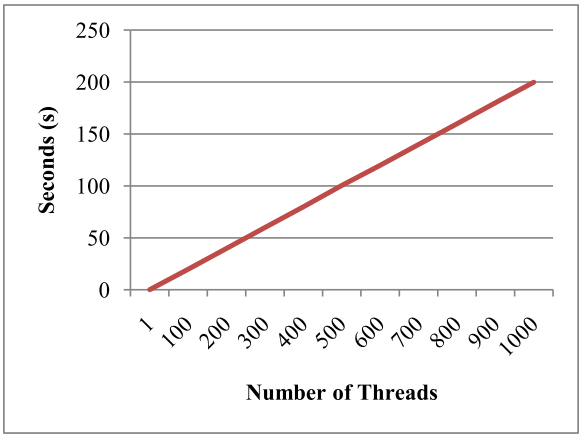

Figure 10: Cost of Thread Creation

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.

5.5 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, its synchronization constructs performance must scale as well.

A fast and efficient barrier minimizes the cost of synchronization which in turn increases performance.

The program used in this benchmark creates a barrier as well as a number of threads. Each thread, as well as the main, 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 11.

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

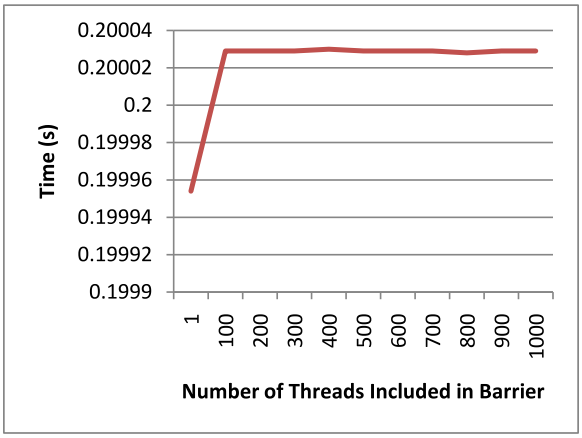

Figure 11: Latency of Barrier Operation

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.

6 Related Work

Due to the growing shift toward distributed memory systems, much work has been done on allowing programmers to utilize these systems for parallel applications. These works can be grouped based upon how distributed memory is managed and shared. Three such groups are message passing models, user-managed shared memory models, and runtime managed shared memory models. TNT on BG/P incorporates and expands upon these works by providing users with a runtime managed memory model utilizing a parallel programming API which is immediately recognizable by any programmer who has utilized threads.

Many parallel programming libraries for distributed systems rely upon underlying message passing models. As such, many message passing models have been designed, including DCMF[8], ARMCI[9][10], Adaptive MPI[11], GASNet[12], Active Messages[13], and MPI[14]. The Deep Computing Messaging Framework is a low-level user library developed by IBM to provide efficient RDMA for BlueGene/L and BG/P, and is used as the underlying communication framework for other message passing libraries such as ARMCI, GASNet, and BG/P's version of MPICH-2 [15]. MPICH-2 is the implementation of the MPI programming model on BG/P. Other message passing libraries used on BG/P include AMPI and GASNet. AMPI provides a runtime managed mapping of MPI tasks to physical processors, but still relies upon the underlying MPI programming model. On BG/P, GASNet directly maps to IBM's DCMF. ARMCI's implementation on BG/P also directly maps to DCMF, but provides further support for remote memory allocation. TNT's underlying communication framework was inspired by ARMCI but maps distributed memory to a linear address space, rather than utilizing specialized handle types.

Due to the differences between message passing models and those traditionally found on shared

memory systems, another avenue of research has been user-managed shared memory libraries, such as Global Arrays[16][17][18][19], Bigtable[20], VMMC-2[21], and Sinfonia[22]. These provide interfaces that allow users to access reserved regions of shared memory from any node on the system without the need for explicit message passing. One approach utilized by models such as Sinfonia, reserve dedicated memory nodes. This limits the maximum performance of a system and ensures that all data will always be remote to the node performing computations. This is in contrast to the distributed memory model employed by libraries such as Global Arrays. These libraries distributed shared memory across all nodes, removing the distinction between memory and compute nodes. TDSM follows a similar approach to Global Arrays, in that shared memory is distributed across every node. However, whereas Global Arrays requires users to specify the distribution of memory when making a request, TDSM instead performs all shared memory reservations during initialization and simply requires the user to specify the size of the request as though it were a local memory allocation.

To improve scalability of code across varying numbers of nodes and provide more familiar interfaces to programmers, research has been done on parallel programming models in which the runtime manages the distributed memory to provide users with a shared memory. Such works include X10[23], Fortress [24], Chapel[25], Titanium[26], Co-Array Fortran[27], Cilk[28][29], Unified Parallel C[30], and Charm++[31]. Cilk provides a C-based multithreaded programming model in which the user-level runtime systems handle memory access on different platforms including SMP and distributed machines. As far as the authors know, Cilk does not provide support for BG/P. Libraries which support BG/P include Berkeley's implementation of UPC and Charm++. UPC employs an SPMD model to parallelize programs, whereas Charm++ utilizes an object-oriented approach that maps work to physical processors. TNT also maps threads to physical processors, but does so through a PThreads inspired API. Furthermore, whereas Charm++ only provides limited support for synchronization through shared objects, UPC employs keywords and specialized memory allocation functions to allow the user to treat shared memory as though it were local for the purpose of pointer arithmetic. TNT's underlying TDSM is inspired by UPC's shared memory model, but handles addressing of shared memory in a manner to avoid intermediary forms during compilation.

7 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 [3] for access to their Surveyor machine.

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.

8 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 multichip systems.

In order to test the feasibility of our design we benchmarked a number of accept 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] "Argonne leadership computing facility." December 2009, `www.alcf.anl.gov`.

[4] 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.

[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.

[9] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda, "High performance remote memory access communication: The armci approach," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 233–253, 2006.

[10] "Armci webpage." September 2009, `http://www.emsl.pnl.gov/docs/parsoft/armci/`.

[11] C. Huang, G. Zheng, L. Kalé, and S. Kumar, "Performance evaluation of adaptive mpi," in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2006, pp. 12–21.

[12] R. N. Nishtala, P. H. Hargrove, D. O. Bonachea, and K. A. Yelick, "Scaling communication-intensive applications on bluegene/p using one-sided communication and overlap," in *IPDPS*, 2009.

[13] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," UCB, Tech. Rep., Nov. 1991, from SC'91 workshop.

[14] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA: MIT Press, 1994.

[15] 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.

[16] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: a portable "shared-memory" programming model for distributed memory computers," in *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 340–ff.

[17] J. Nieplocha, B. Palmer, V. Tipparaju *et al.*, "Advances, applications and performance of the global arrays shared memory programming toolkit," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 203–231, 2006.

[18] J. Nieplocha, M. Krishnan, B. Palmer *et al.*, *The Global Arrays User's Manual*, 2006.

[19] "Global arrays webpage." September 2009, `http://www.emsl.pnl.gov/docs/global/`.

[20] F. Chang, J. Dean, S. Ghemawat *et al.*, "Bigtable: A distributed storage system for structured data," in *In Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, 2006, pp. 205–218.

[21] C. Dubnicki, A. Bilas, Y. Chen *et al.*, "Vmmc-2: Efficient support for reliable, connection-oriented communication," in *In Proceedings of Hot Interconnects*, 1997.

[22] M. K. Aguilera, A. Merchant, M. Shah *et al.*, "Sinfonia: a new paradigm for building scalable distributed systems," in *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. New York, NY, USA: ACM, 2007, pp. 159–174.

[23] P. Charles, C. Grothoff, V. Saraswat *et al.*, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2005, pp. 519–538.

[24] J. Hallett, "Semantics and type soundness proof of a core fragment of fortress with hidden type variables," 2007.

[25] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.

[26] K. Yelick, L. Semenzato, G. Pike *et al.*, "Titanium: A high-performance java dialect," in *In ACM*, 1998, pp. 10–11.

[27] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[28] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul *et al.*, "Cilk: An efficient multithreaded runtime system," pp. 207–216.

[29] R. D. Blumofe and P. A. Lisiecki, "Adaptive and reliable parallel computing on networks of workstations," in *ATEC '97: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1997, pp. 10–10.

[30] W. Carlson, J. Draper, D. Culler *et al.*, "Introduction to upc and language specification," IDA Center for Computing Sciences, Tech. Rep. CCS-TR-99-157, 1999.

[31] L. Kale and S. Krishnan, "CHARM++: A Portable Concurrent Object-Oriented System based on C++," *ACM Sigplan Notices:Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, vol. 28, no. 10, pp. 91–108, October 1993.