Extending Task-based Programming Model beyond Shared Memory Systems

Thawan Kooburat [1], MinJae Hwang [2]

Computer Science Department, University of Wisconsin-Madison
1210 W. Dayton St., Madison, WI, USA 53706
[1] kooburat@cs.wisc.edu
[2] m@cs.wisc.edu

***Abstract*— Task-based programming model allows programmer to easily exploit the computation power of chip multiprocessing because he can express fine-grained parallelism without worrying about overhead. However, it is normally used in shared memory system which allows implicit communication and synchronization. As a result, this programming model cannot be used to efficiently utilize distributed memory system such as commodity cluster computers.**

In this paper, we propose the TaskGroup/Task programming model which extends task-based programming. TaskGroup is a collection of task and it can be sent to run on other nodes; whereas, Tasks are spawned from their parent TaskGroup and their communication within their group can be done implicitly via shared memory. We propose pre-fetch work-stealing policy to manage hierarchical work queue in our runtime system. The results show that our system has good speed up and scalability on certain classes of applications.

***Index Terms*—task-based programming, distributed memory systems, cluster computer**

I. Introduction

WITH current trends, Moore's law will be continued as to increase the number of cores with in a chip. This makes shared memory system abundant and accessible by most programmers. Nevertheless, this also means that programmers must resort multi-threaded programming in order to utilize this type of system.

Parallel programming model is increasingly important as it plays crucial role in reducing the burden of programmers. We found that task-based programming model such as those found in Cilk [1], Intel Threading Building Blocks (TBB) [2] and Java Fork/Join Framework [3] are easy to adopt and popular among programmers.

Task-based programming model allows programmer to express and divide computation into tasks rather than manually maneuver threads. Normally, tasks are created dynamically and allows program to expose high degree of parallelism as if he has infinite number of processors. Also, it prevents to have deadlocks and data races among threads. This allows the program to scale when more resources are available. As a result, the runtime system is responsible for minimizing the overhead and executes the program efficiently despite the fact that the system may have just a few processors. This problem is solved by using a work-stealing scheduler that map tasks onto a pool of threads. However, this model couple tightly with shared memory facility because both programmer and work-stealing algorithm can rely on implicit communication and synchronization via global or shared variables.

Before the rise of shared memory systems and multi-thread programming, parallel programming using message passing interface (MPI) [4] is very popular and it is very efficient in utilizing distributed memory systems. However, as its name suggests, communication between processes in MPI must be explicit and more suitable to problem which can be parallelized using static decomposition.

To harness cluster of multi-core system, it is also possible [5] to mix MPI with multi-threaded programming model such as OpenMP [6] and achieve good performance. However, it requires programmers to adopt two programming style to solve a single problems.

This leads us to the proposed TaskGroup/Task programming model. It allows dynamic creation of tasks along its execution while it can be executed over multiple computation nodes. The key idea is that TaskGroup can be considered as a collection of Task and is assumed to be abstraction of single machine. Thus, TaskGroup must contain all input necessary for carrying out it computation. However, TaskGroup can spawn Tasks and communication between Tasks within the same TaskGroup can be done implicitly via shared memory. Thus, this model can effectively utilize low memory latency and abundant memory bandwidth of shared memory system. By allowing programmer to express computation in this manner, his program will be able to utilize multi-core capability of single node and scalability over multiple nodes.

In order to realize this model, we have created a runtime system which uses work-stealing algorithm to schedule TaskGroup onto worker nodes. We use Java Fork/Join Framework extensively to handle Task at a single machine level. Then, we use MPJ Express [7], which provides MPI implementation for Java, for communication and coordinating

worker nodes.

The rest of the paper is organized as follows. Section 2 talks about related work. We state some of the design considerations in Section 3. Then, Section 4 describes our proposed programming model. In Section 5, we talk about runtime system design and follows by its implementation in Section 6. Section 7 provides runtime system evaluation result. Then, we talk about various policies that we have tried in Section 8. In Section 9 we discuss some of the behavior or our runtime system. Finally, we conclude our work in Section 10.

II. Related Works

A. *Message Passing Interface (MPI)*

MPI is the most popular programming model in the area of scientific computing. It uses explicit communication model to allow parallel programs to communicate in a distributed system. Because of the lack of shared memory facility, programmers have to manually manage data sharing via MPI. Additionally, low-latency and high-bandwidth network interface such as Myrinet or InfiniBand allows these systems to achieve higher performance.

Nevertheless, this model is more suitable to problem that can be parallelized using static decomposition.

B. *Distributed Shared Memory (DSM)*

There are many works [9 - 11] which allows programmers to use existing multi-threaded programming models on distributed systems by introducing shared memory system. This imposes no cost on the programmer side since it has similar abstraction of shared memory system. Some of these systems [9, 14] increase the efficiency by using specialized DSM which has weak memory consistency model in order to hide network latency.

However, program has to exhibit good locality in order to achieve good performance. Additionally, optimization is tend to complex because it has many hidden cost that cannot be seen easily at first place.

C. *OpenMP/MPI*

MPI programs can be run on both distributed memory and shared memory systems. However, it has unnecessary overhead on both performance and cost of programming if target system is a shared memory. To increase the efficiency, programmer can mix OpenMP with MPI [5] by parallelizing inner loop using OpenMP. Then, MPI is used to share data and coordinate worker nodes.

However, programmers must adopt two programming style use this model and it only suitable with problem that can be parallelized using static decomposition.

D. *Task-based Programming Model*

Task-based programming [1 - 3] allows programmer to divide computation into tasks. It also allows programmer to express fine-grained parallelism without worrying that it will incur too much overhead because the runtime system will use work-stealing policy to efficiently manage these tasks. Tasks will be execute in parallel only went there is an available worker. The advantage of this model is that it allows dynamic task creation hence the program can spawn tasks along its execution. However, task-based programming is generally restricted to shared memory system.

E. *Memory Hierarchy Aware*

This idea is based on an observation that many systems have hierarchies of memories. One of the key benefits of this model is to allow programmer to efficiently manage memory even if program working set is much larger than each machine's main memory. Sequoia [12, 13] is a parallel programming model that allow program to divide data into tree and map computation onto tree nodes. This work shows that hierarchical programming model is necessary to express computation is a manner that fit the internal layout of the target system.

III. Considerations

A. *Programming Model Considerations*

In a cluster environment, the system exhibit two sets of characteristics. Within a single machine, communication can be done implicitly using shared memory. On the other hand, at a cluster level, communication must be done explicitly via network. Additionally, task granularity must be large enough to outweigh the overhead of network latency. As a result, there should be a programming model that is suitable for this hierarchical nature of a cluster system.

B. *Work-stealing Policy Considerations*

The main efficiency of work-stealing policy comes from building sophisticate concurrent task queue. However, in a cluster environment, network latency and bandwidth limitation means that we cannot adopt the same strategy as a typical work-stealing policy employed in shared memory environment. For example, checking the status of other worker is not going to take just a few instructions but a few milliseconds. As a result, work-stealing policy should be optimized to reduce network communication without reducing the overall performance or program parallelism.

IV. TaskGroup/Task Programming Model

TaskGroup is a collection of Task. It provides a logical view from programmer to divide computation into groups. The main benefit of this model is that Tasks, which are spawned from the same TaskGroup, can assume that they are operating on the same system so they can communicate implicitly via shared memory. On the other hand, TaskGroup will be scheduled across worker nodes so it must contain all inputs that are necessary for its computation. As a result, this model captures the benefit of both shared and distributed memory and allows programmer to use a single programming model to tackle the problem.

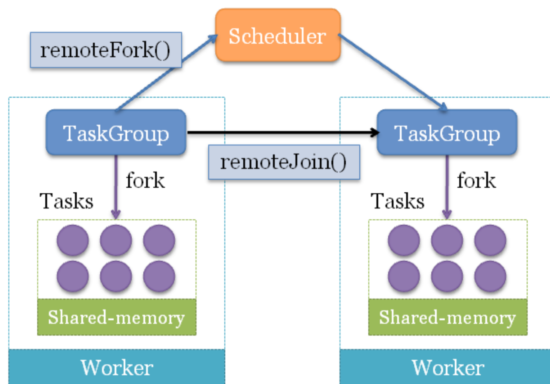

Fig. 1. TaskGroup/Task model

The main syntax extension of TaskGroup/Task model is *remoteFork()* which allow programmer to express that work can be spawned off and run on other worker nodes. Fig. 2 use the Fibonacci to show the main usage pattern of this construct in which programmer use *cutoff* value to determine the when to spawn a TaskGroup. This pattern will fit with recursive task generation because programmers can easily insert an additional condition into existing code. In this example, *cutoff* value is based on *size* because it implies the execution time of task.

```
public class FibTG extends TaskGroup<Long> {
   int size;

   protected Long compute() {
      if (size == 2 || size == 1)
   return 1L;
      FibTG first = new FibTG(size - 2);
      FibTG second = new FibTG(size - 1);
      if (size >= 35) {
         first.remoteFork();
         return second.invoke() + first.remoteJoin();
      } else {
         first.fork();
         return second.invoke() + first.join();
      }
   }
}
```

Fig. 2. Fibonacci example

Another benefit of this model is that it has shared memory accessibility available at task level. For matrix multiplication program, a programmer needs to divide and copy a portion of matrix into TaskGroup in order to execute in different machines. However, when TaskGroup spawns a task locally, the programmer only needs to pass a reference because the tasks can access same memory space that TaskGroup owns.

V. Runtime System Design

The runtime system is responsible for scheduling TaskGroups over worker nodes and scheduling inside single node. We introduce hierarchical queuing system and work-stealing over cluster. Fig. 3 describes an overview of the runtime system. It shows relationship between each queue.

A. *Hierarchical Queue*

Our runtime system relies on 3 types of queue – Global Queue, Local Queue, and Thread Queue. We introduced a single global queue to improve TaskGroup availability in the local queue. Each machine only needs to communicate with global queue rather than trying to steal from tens and hundreds of other local queues. Also, a single global queue can scheduler better because it knows status of every local queue and make decisions based on this omniscient knowledge.

Each queue works in intuitive way. Thread Queue feeds each thread in a machine. So, if a thread queue is empty, corresponding thread will be idle. Local Queue feeds thread queues in a machine. Global Queue feeds entire machines.

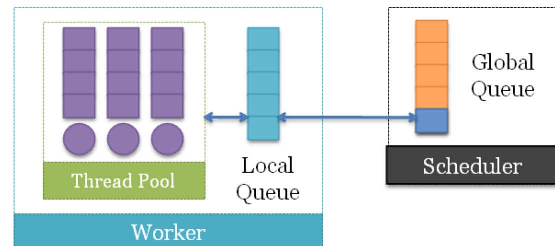

Fig. 3. Hierarchical task queues

B. *Pre-fetching Work-stealing Policy*

Pre-fetching work-stealing policy tries to hide network latency by pre-fetching some of TaskGroups. The key question here is how much TaskGroup needed to be pre-fetched. In chip-level implementation [15] of hierarchical queue, they required only one task to hide fetching delay. In our runtime system, we assumed TaskGroup is sufficiently large to be beneficial to run on other system. Since the runtime system does not have any knowledge about the execution time of TaskGroup, the ideal answer would be the runtime system dynamically adapts to the problem. As a result, this policy alternate between two behaviors which are described as follows.

1) *Pre-fetching mode*

In this mode, two variables – *low* threshold and *high* threshold are used to control the behavior of local queues. When the local queue reaches *low* threshold, it will try to steal from the global scheduler. In contrast, if it reaches *high* threshold, it will give surplus TaskGroups to the global scheduler. In this modes global scheduler behavior can be consider as a passive mode because it will never steal task from worker nodes.

2) *Work-stealing mode*

Special care must be provided if there is an idle machine in a cluster. A request from the idle machine should have high priorities than other requests because it denotes possible load-imbalance. The global scheduler should be aware of idle machine and it tries to give TaskGroup to idle machine as quickly as possible.

When global scheduler is empty under this circumstance, it should act promptly to resolve the issue. In this case, the global scheduler tries to steal from every worker as much as possible because this is emergent status. To rebalance local queue in every machine, each worker are required to send all TaskGroups except one to the global scheduler. This will allow global scheduler to service all idle workers and revert to pre-fetching mode if possible.

VI. Implementation

In this section, we will explain about general implementation of the runtime system. Also, we will describe how we implemented the runtime system in system level.

A. *Structure of TaskGroup*

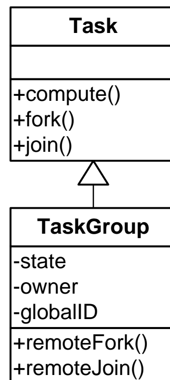

Fig. 4. TaskGroup and Task class diagrams

Task inherits most of its methods and members from *ForkJoinTask* which is a generic task class in Java Fork/Join framework. TaskGroup has three fields. These are *owner*, *globalId*, and *state*. *GlobalId* uniquely identifies given TaskGroup in a cluster. *GlobalId* used in the scheduler for various purposes. Owner denotes who originally remote-forked. So, if the TaskGroup is not locally forked, the executor who stoles the TaskGroup should send a result to the owner. We use MPI's rank number to represent node and store it in the *owner* field so that we can easily send the back using MPI message.

State can be one of *local*, *stolen*, and *completed*. Before stolen, TaskGroup is *local*. After it has been stolen, it became *stolen* state. When an executor returns a result, it becomes *completed*. This state is used when TaskGroup remote-joins on the TaskGroup.

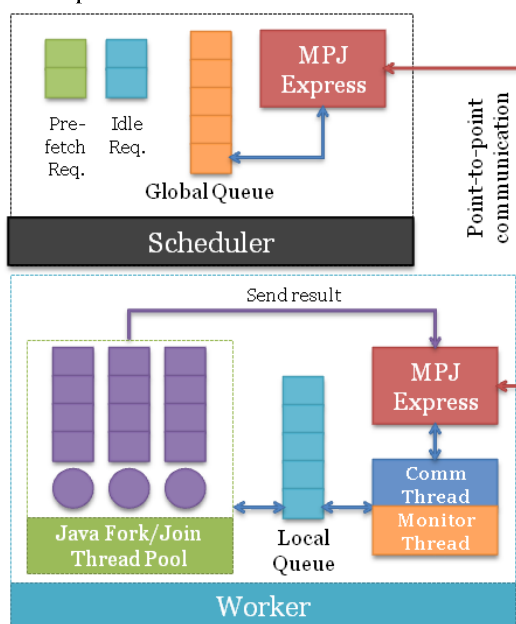

Fig. 5. Components in runtime systems

B. *Communication between Nodes*

We used MPJ Express as communication framework. So, every message is communicated with MPI-style messaging. In case of transferring TaskGroup, we utilized Java's serialization facility which allows any class that implement *Serializable* interface to be converted into bit steam. It automatically serialize and de-serialize when the runtime system send TaskGroups over network.

C. *RemoteFork*

As our runtime system works on supply and demand of each worker, remote-forked TaskGroup does not directly given to the global scheduler. Instead, it put into its local queue. If the worker later found that it has surplus TaskGroups, the communication thread will send them off to the global scheduler.

This off-loading technique imposes very little overhead on critical path. Also, it can buffer unnecessarily spawned TaskGroups because it does not transfer TaskGroups when it is not needed.

One caveat is that when every thread tries to put too many TaskGroups into the local queue, it became the bottleneck. We employed lock-free data structure to minimize the bottleneck. However, we address this concern in detail in later section.

D. *RemoteJoin*

In Java Fork/Join Framework, if a thread joins on some local task, the thread will block until joining task's state is changed to be completion. In this case, the Fork/Join Framework wakes up another spare thread to take up CPU core that this thread just releases. Even if we have only few spare threads, we can utilize our CPU fully because in most cases, joining to local tasks will not take few milliseconds.

However, the situation is a little bit different in our runtime system. We observed that in many cases, entire threads in a machine are blocked on remote-joining. As a result, we added extra number of spare threads. We carefully measured relationship between total execution time and number of pending threads. We found that number of threads in thread poo should be the triple the number of available CPU to achieve good performance.

We think that TBB's continuation-style programming completely eliminates this kind of problem. Continuation style programming suggests that each task will not join on other tasks but the programmer can specify conditions when to run tasks. So, rather than waiting on tasks to be completed, the programmer specifies the condition that if some tasks are completed, then execute "continuation" task. However, it requires major modification to Java Fork/Join Framework's scheduling algorithm.

E. *Local Scheduler Behavior*

Local Scheduler pre-fetches TaskGroups to the local queue as described in runtime system design. A separate *monitor* thread is used to periodically check a number of items in a local queue. We used the submission queue maintained by Java

Fork/Join Framework as a local queue. By default, any idling thread in thread pool will steal Task from submission queue only when they cannot steal Task from other threads in the pool. Additionally, the pooling thread tries to steal from this queue and send to the global scheduler when it sees surplus.

F. Communication Thread

For the local scheduler, it uses a separate communication thread. The communication thread blocks on network receiving. So, it handles every income message.

However, for sending a result, we did not employ a separate *sender* thread because MPI supports a non-blocking send operation. So, each thread calls non-blocking send by itself when returning a result.

G. Global Scheduler

One of the important roles of the global scheduler is to prioritize requests from the local scheduler. It maintains two separate lists to handle idle request and pre-fetch request.

The global scheduler is single threaded server. Since every request does not involve long computation, single thread was enough to do the job.

VII. Runtime System Evaluations

We conduct test on a cluster of Intel Q9400, Quad-core 2.66GHz, Shared 6MB L2 Cache with 8GB RAM on each machine. All of them are connected using 100Mbps Ethernet network. We use 4 benchmarks for testing our runtime system.

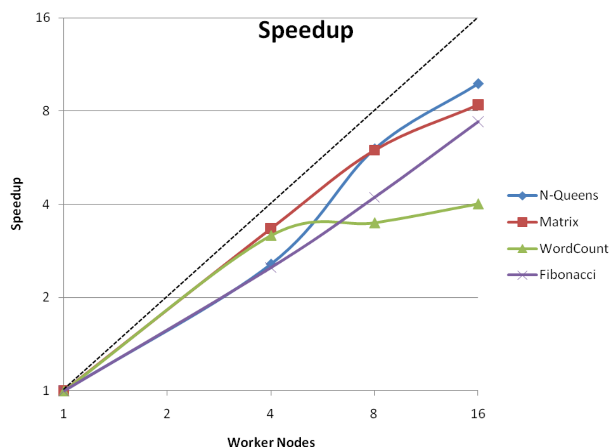

Fig. 6. Speedup of benchmark programs

1. Matrix Multiplication: This is a large (4096x4096 elements) floating-point number matrix multiplication. To reduce network traffic and serialization time, each TaskGroup initialize its own portion of matrix and only final result are returned. This program represents the class of applications with static decomposition.

2. Word Count: This problem is implemented using producer/consumer style. TaskGroup will be assigned with a list of files which can be accessed via AFS shared file system. It will process a text file and store the result into hash table which will be combined into a single one when program finished.

3. N-Queens: This is a search problem with recursive task generation style. It represents a class of application which can introduce load imbalance because computation cost of each TaskGroup may vary.

4. Fibonacci: This problem can be considered as a micro-benchmark for the runtime system because has little computation and spend most of the time spawning TaskGroups or Tasks.

Fig. 6 shows the speedup comparison between various number of worker nodes. In case of one worker nodes, we execute the original version of benchmark programs on Java Fork/Join Framework, so it does not have overhead of our runtime system.

The result shows that our system has good scalability because most of applications exhibit good speedup as number of nodes increase. Additionally, the speedup of Fibonacci also shows that the overhead of system grows linearly with the number of nodes. In case of the word count program, its performance suffers when more than four nodes are used because it does not do the reduction stage in parallel.

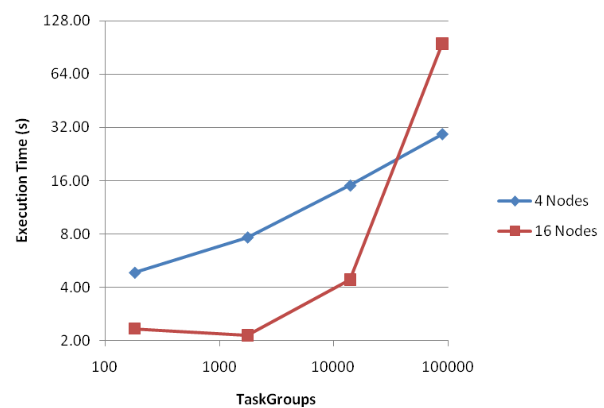

Fig. 7. Relationship between number of TaskGroups and execution time on N-Queens program

Fig. 7 shows that runtime system will execute efficiently when number of TaskGroup falls into certain range that vary according to number of worker nodes. For example, 200 TaskGroup is enough to fully utilize 4 worker nodes but not enough for 16 nodes. On the other hand, too many TaskGroups generated may cripple the system.

VIII. Work-Stealing Policy Evaluation

Work-stealing policy plays important role in remedying load imbalance over a cluster. Dynamic task generation inherently involves load imbalance since each task cannot be identical. Network latency further complicates. Unlike task stealing in shared memory system, it involves certain amount of network latency. We will describe work-stealing policies tested during our development and provide performance comparison.

A. Static Distribution

This is most naïve approach however still shows quite good performance in some class of problem. Whenever TaskGroup spawns another TaskGroup, each thread directly send new TaskGroup to the global scheduler. Then, the global scheduler distributes new TaskGroup in round-robin fashion. We

implemented it in our prototype and could achieve decent scale-up for Fibonacci program. If an execution time of each TaskGroup is alike, this naïve approach works fine.

B. *Purely Work-stealing Policy*

Purely work-stealing policy is direct remedy to static distribution. TaskGroup is sent to the global scheduler whenever it is generated. However, instead of distributing in round-robin fashion, the global scheduler responds to each local machine's request. So, if the local machine has empty local queue, it will request to the global scheduler. Purely work-stealing works quite well if network latency is quiet small compared to the execution time of TaskGroup. However, in most cases, the runtime will suffer from inevitable network latencies.

C. *Fixed low and high threshold*

Since our assumption "TaskGroup should be reasonable size" holds, fixed *low* and *high* threshold yielded quite good performance on various benchmarks. One notable observation is that having *low* threshold to 1 is quite sufficient for most of the program. This is because most of the problem had small size of inputs except problems like matrix multiplication.

D. *Switch Policy*

Switch policy is based on an observation on specific program. Since most of the workers are idle at the initial stage, TaskGroups should be transferred to another machine as soon as possible. To transfer TaskGroup quickly, the runtime system at the initial stage should behave like static distribution. Then, after sometime passed, it switches to pre-fetch work-stealing. This strategy is quite good for divide-and-conquer style problems since they have obvious turning point.

E. *Consumption-based Policy*

Consumption-based policy means it tries to pre-fetch TaskGroup according to actual consumption rate of TaskGroup in each machine. It sounds rationale that each worker does not need more than they consume. However consumption rate is heavily fluctuating in most of applications. Its behavior is changing every 50 milliseconds or so. In order to have better scheduling, their behavior should continue same for few seconds.

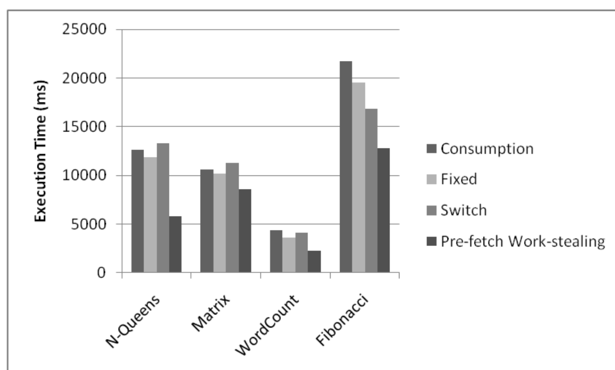

Fig. 8. Relationship between various scheduling policies and execution time

Fig. 8 shows the execution time of all benchmark programs on 4 worker nodes. Between development policies which are Consumption, Fixed, and Switch, there is no major difference in execution. However, Switch produces a noticeable execution time difference in case of Fibonacci. This is because Fibonacci computation size reduces dramatically as TaskGroup is being spawned. As a result, it is critical to distribute initial TaskGroup as soon as possible to prevent load-imbalance. This allows us to conclude that good scheduling policies need to exhibit different behaviour at different stages of computation.

Pre-fetch work-stealing performance is much better than other policies because it combines all important behaviours that other policies have. However, we believe that the main performance improvement comes from other optimizations that we incorporate into our runtime system.

IX. DISCUSSION

A. *Serialization and Returning Results*

After calling remote-join, only result or public fields of the remote TaskGroup can be accessed because runtime system cannot set non-public values. Also, programmer can reduce the size of TaskGroup by nullifying unnecessary member variables in order to reduce serialization time and network traffic

B. *Number of TaskGroups*

How many remote-forked TaskGroups are required in a program and how to decide *cutoff* value? This is critical question for this TaskGroup-based programming model. If a number of remote-fork is too small, the problem will not be able to fully utilize a cluster. However, if it is too many, unnecessary overhead will be imposed.

Our runtime system does not send TaskGroups immediately when the program invokes *remoteFork()*. The runtime system sends only if there is idling machine.

Theoretically, if nothing is stolen, it should not be different whether the program called *remoteFork()* or fork(). However, there is some difference in *fork()* and *remoteFork()* for task management. If a task or TaskGroup is locally forked, it will directly push to the thread queue. However, if it is remote-forked, it will push to the local queue. Thus, it requires a thread to steal from the local queue. Therefore, it will have a small additional overhead when *remoteFork()* is called even if it is executed entirely in same machine.

From our empirical experiences, if the program runs in 10 to 20 seconds, 1,000 – 2,000 of TaskGroups can be tolerated without causing a major performance penalty. In divide-and-conquer style problem, this can be easily tuned by using *cutoff* values. So, most of cases, the programmer can decrease number of remote-fork without major problems.

The real problem is when we do not have sufficient remote-fork. Due to the pre-fetching work-stealing, each machine tends to have spare TaskGroups. If the number of TaskGroup is range of n to 4n, n denotes a number of machines in a cluster, it is better off to not to have spare TaskGroups. In this case, purely work-stealing performs better than pre-fetch

work-stealing. Programmer can tune his program by lowering *low* threshold value to zero in pre-fetching work-stealing to simulate purely work-stealing.

We suggest a simple rule of thumb base on our empirical experiences that 100 – 1,000 TaskGroups are suitable for 8 nodes and 200 – 1,500 might more suitable for 16 nodes. These guidelines are quite similar to how TBB suggests that each task should contain 10,000 – 100,000 instructions.

X. Conclusion

TaskGroup/Task is extension to task-based programming model that allows programmer to use this intuitive model in a cluster of SMP environment. The additional level of TaskGroup allows us to capture the hierarchical nature of the system which is necessary for efficient scheduling. It also allows programmer to fully utilize available resources.

We have tested various work-stealing policies and found that the pre-fetch work-stealing policy that we developed strikes a good balance between load-balancing and hiding the cost of communication.

Acknowledgment

We are thankful for Professor David Wood's advice on the direction of our project. We also gratefully acknowledge Aamir Shafi for providing documentations and suggestion about using MPJ Express. Finally, we also appreciate Professor Doug Lea for providing us with the Java Fork/Join Framework benchmark source code.

References

[1] Matteo Frigo, Charles E. Leiserson, Keith H. Randall, "The Implementation of the Cilk-5 Multithreaded Language", PLDI 1998.
[2] Thread Building Block, http://www.threadingbuildingblocks.org/
[3] Java Concurrency JSR-166, http://gee.cs.oswego.edu/dl/concurrency-interest/
[4] MPI Forum, http://www.mpi-forum.org/
[5] Rolf Rabenseifner, Georg Hager, and Gabriele Jost: Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. PDP2009.
[6] OpenMP, http://www.openmp.org/
[7] Mark Baker, Bryan Carpenter, and Aamir Shafi. MPJ Express: Towards Thread Safe Java HPC, Cluster2006.
[8] Smith, Bull, Obdrzálek, "A parallel java grande benchmark suite", SC2001.
[9] Jay P. Hoeflinger, Extending OpenMP* to Clusters, Intel Whitepaper 2006.
[10] L. Peng, W.F. Wong, M.D. Feng, C.K. Yuen, "SilkRoad: A Multithreaded Runtime System with Software Distributed Shared Memory for SMP Clusters", Cluster2000.
[11] Mark W. MacBeth, Keith A. McGuigan, Philip J. Hatcher, "Executing Java threads in parallel in a distributed-memory environment", IBM Centre for Advanced Studies Conference 1998.
[12] Kayvon Fatahalian, et al., "Sequoia: Programming the memory hierarchy". SC2006.
[13] Mike Houston, et al., "A Portable Runtime Interface For Multi-Level Memory Hierarchies", PPoPP 2008.
[14] Robert D. Blumofe, et al., "Dag-Consistent Distributed Shared Memory", International Parallel Processing Symposium, 1996.
[15] Sanjeev Kumar, et al., "Carbon: architectural support for fine-grained parallelism on chip multiprocessors", ISCA2007.