Implementation of Distributed Loop Scheduling Schemes on the TeraGrid

Satish Penmatsa[1]*, Anthony T. Chronopoulos[1]†, Nicholas T. Karonis[2,3], and Brian R. Toonen[2]

[1]University of Texas at San Antonio
Dept. of Computer Science
San Antonio, TX 78249
{spenmats, atc}@cs.utsa.edu

[2]Northern Illinois University
Dept. of Computer Science
DeKalb, IL 60115
{karonis, btoonen}@niu.edu

[3]Argonne National Laboratory
Math. and CS Division
Argonne, IL 60439

Abstract

Grid computing can be used for high performance computations. However, a serious difficulty in concurrent programming of such heterogeneous systems is how to deal with scheduling and load balancing of such systems which may consist of heterogeneous computers on different sites. Distributed scheduling schemes suitable for parallel loops with independent iterations on heterogeneous computer clusters have been proposed and analyzed in the past. Here, we implement the previous schemes in MPICH-G2 and MPIg on the TeraGrid. We present performance results for three loop scheduling schemes on single and multi-site TeraGrid clusters.

1. Introduction

Modern scientific applications are computation and data intensive requiring extremely powerful computing systems. The emergence over the past decade of grid middleware has helped to spawn computational grids [19] around the world that provide unprecedented computational power and storage capacity. The US National Science Foundation's TeraGrid (TG) [7], the US Department of Energy's Earth System Grid (ESG) [5] and Open Science Grid (OSG) [6], Japan's National Research Grid Initiative (NAREGI) [4], and the European Commission project Enabling Grids for E-scienceE (EGEE) [2] are just a few examples of the many grids in existence today. While there are examples of applications that have been successfully developed or modified to run efficiently on computational grids [11, 23, 28, 26, 15], these successes have often come through customized solutions and the exploration for new or even general techniques that efficiently harness the power of computational grids like the ones we describe here is an area of active research.

So-called computational Grids enable the coupled and coordinated use of geographically distributed resources for such purposes as large-scale computation, distributed data analysis, and remote visualization. The development or adaptation of applications for Grid environments is made challenging, however, by the often heterogeneous nature of the resources involved and the fact that these resources typically reside in different administrative domains, run different software, are subject to different access control policies, and may be connected by networks with a widely varying performance characteristics.

The Globus Toolkit [17, 3] is a collection of software components designed to support the development of applications for high-performance distributed computing environments, or "Grids" [18, 19]. Core components typically define a protocol for interacting with a remote source, plus an application program interface (API) used to invoke that protocol. Higher-level libraries, services, tools, and applications use core services to implement more complex global functionality.

MPICH-G2 [24] is a complete implementation of the MPI v1.1 [8] standard that is specifically designed to run on computational grids. It uses Globus Toolkit services to provide a single interface to securely launch applications that can span many different administrative domains and to automatically convert data transferred between machines with different architectures (e.g., big endian and little endian machines). Through the use of MPI idioms MPICH-G2 enables running applications to discover process distribution to, for example, create communicators that group processes executing on the same location and also to manage network heterogeneity by configuring inter-cluster network links to use either multiple TCP streams and striped messages or to use UDP with added reliability, both for increased bandwidth utilization for large data transfers. MPICH-G2 also takes advantage of its knowledge of the performance dif-

*Student Member IEEE
†Senior Member IEEE

1-4244-0910-1/07/$20.00

ferences found in a grid's heterogeneous network by implementing grid topology-aware collective operations for improved performance [22].

The developers of MPICH-G2 have recently completed their latest MPI library called MPIg that is intended to be MPICH-G2's successor. MPIg, like MPICH-G2, is specifically designed to run on computational grids and also uses services from the Globus Toolkit to securely launch applications and to automatically convert data. However, MPIg, unlike MPICH-G2 is based upon MPICH-2, the most recent release from the MPICH group at Argonne National Laboratory [1], and is fully integrated with the web services now distributed in version 4.x of the Globus Toolkit (i.e., GT4).

MPIg has the same fundamental architecture as its predecessor MPICH-G2 using a vendor-supplied MPI for intra-cluster messaging where available and an IP-based protocol, most commonly TCP, for inter-cluster messages across the wide-area. In addition, however, there are two new low-level features embedded deep within MPIg's design.

First, while MPICH-G2 use the Globus-IO library from the Globus Toolkit for its inter-cluster messaging MPIg instead uses the more advanced Globus-XIO library [10]. This shift enables MPIg to take advantage of Globus-XIO's modular design to quickly implement and study protocols other than TCP/IP. In the past implementing alternative protocols like GridFTP [9], RBUDP [21], and UDT [20] each required significant modification to MPICH-G2. Also, Globus-XIO's so-called "transformation module" makes it easy for XIO users like MPIg to compose their own protocol stack (e.g., adding encryption or compression) for inter-cluster messaging.

Second, and more importantly, MPIg now uses multiple threads to transport messages. The greatest challenge in developing successful MPI grid applications is dealing with the network performance heterogeneity inherent in grids. Inter-cluster network performance, as characterized by latency and bandwidth, across the wide area often differs from faster intra-cluster networks by orders of magnitude. While the decrease in available bandwidth can play a role, particularly for applications that send large messages across the wide area, more often it is the latency, which is measured in milliseconds to tens of milliseconds across the wide area as opposed to a few microseconds within a cluster, that proves to be the most problematic. Accordingly, many message-passing applications are characterized by their "latency tolerance" and application and middleware groups spend a significant amount of their time searching for "latency-hiding techniques". When executed on multi-core systems, like the ones used to conduct the experiments described in this paper, MPIg's multi-threaded design now allows MPI applications to experience a true overlap of computation and communication by calling MPI's non-blocking point-to-point messaging functions MPI_Isend and MPI_Irecv. MPIg applications that have used MPI_Isend and MPI_Irecv to overlap computation and communication, like the one we present in this paper, have found this to be an effective latency-hiding technique.

The TeraGrid [7] is one of the largest distributed cyber-infrastructure for scientific research that provides more than 102 Tflops of computing capability and more than 15 Pbytes of online and archival data storage connected by high performance networks. TeraGrid currently uses Globus Toolkit 4.0 [17].

Scientific applications may contain large loops inside them which are one of the largest sources of parallelism. If the iterations of a loop have no interdependencies, each iteration can be considered as a task and can be scheduled independently, which is commonly known as *loop scheduling*. Loops can be scheduled statically at compile-time. This scheduling has the advantage of minimizing the scheduling time overhead, but it may cause load imbalancing when the loop style is not uniformly distributed. Dynamic scheduling adapts the assigned number of iterations whenever it is unknown in advance how large the loop tasks are. On distributed systems these schemes can be implemented using a master-slave model. See [12, 16] and references there-in.

Heterogeneous systems are characterized by heterogeneity and large number of processors. Such a system can be part of a computational grid. Some significant distributed schemes that take into account the characteristics of the different components of the heterogeneous system were studied in the past [13, 12, 25, 16, 14]. In this paper, we review three important distributed schemes (DTSS, HDTSS and TREES) which were the most efficient in previous studies [12, 13]. We implemented these schemes on the TeraGrid using MPICH-G2. DTSS is also implemented using MPIg to study the communication cost for bulk data transfers. We made performance tests using the Mandelbrot test problem. As expected, the hierarchical DTSS (HDTSS), which is a more scalable version of DTSS yields the best performance.

The rest of the paper is organized as follows. In Section 2, we review the distributed dynamic scheduling schemes that are implemented in this paper. In Section 3, we provide the implementation details and discuss the simulation results. Conclusions are drawn in Section 4.

2. Distributed Dynamic Loop Scheduling Schemes

In this section, we review three distributed loop self-scheduling schemes [13] that we implement using MPICH-G2 on the TeraGrid. We use the following notations:

- PE : Processor in the distributed system
- I : Total number of iterations in a parallel loop

- p : Number of slave PEs in the distributed system which execute the computational tasks
- P_1, P_2, ..., P_p : p slave PEs in the system
- *Chunk* : A few consecutive iterations
- C_i : The chunk size at the i-th scheduling step ($i = 1, 2 \ldots$).

Self-scheduling is an automatic loop scheduling method in which idle PEs request new loop iterations to be assigned to them. These self-scheduling schemes are based on the Master-Slave architecture model. Idle slave PEs send a request to the master for new loop iterations. The number of iterations (chunk) a slave PE should be assigned is an important issue. Due to PEs heterogeneity and communication overhead, assigning the wrong PE a large number of iterations at the wrong time, may cause load imbalancing. Also, assigning a small number of iterations may cause too much communication and scheduling overhead.

2.1. Distributed Trapezoid Self-Scheduling (DTSS)

The DTSS [13] takes into account the processing speeds of the slave PEs in assigning tasks to them. The virtual power V_i ($i = 1, \ldots, p$) of a slave PE P_i ($i = 1, \ldots, p$) is computed by the master PE as: V_i = $Speed(P_i)/min_{1 \leq i \leq p}\{Speed(P_i)\}$, where Speed($P_i$) is the CPU-Speed of P_i. Thus, the total virtual computing power of the system, V, is given by: $V = \sum_{i=1}^{p} V_i$.

Let Q_i denote the number of processes in the run-queue of P_i reflecting the total load of P_i. Then the available computing power (ACP), A_i ($i = 1, \ldots, p$) of a slave PE is given by: $A_i = \left\lfloor \frac{V_i}{Q_i} \right\rfloor$. Thus, the total available computing power of the system, A, is given by: $A = \sum_{i=1}^{p} A_i$.

The chunk sizes (C_i) are calculated using a chunk decrement. The decrement, D, is given by $D = \left\lfloor \frac{(F-L)}{(N-1)} \right\rfloor$, where F is the first chunk given by $\lfloor \frac{I}{2A} \rfloor$, L is the last chunk which can be set to a *threshold* and N is a value given by $\left\lceil \frac{2*I}{(F+L)} \right\rceil$.

DTSS algorithm:

Master:

1. **(a)** Receive all $Speed(P_i)$.
   (b) Compute all V_i.
   (c) Send all V_i.
2. **(a)** Receive A_i; sort A_i in decreasing order and store them in a temporary ACP Status Array (ACPSA). For each A_i, place a request in a queue in the sorted order. Calculate A.
   (b) Calculate F and D.
3. **(a)** While there are unassigned iterations, if a request arrives, put it in the queue and store the newly received A_i if it is different from the ACPSA entry.
   (b) Pick a request from the queue, assign the next chunk $C_i = A_i * (F - D * (S_{i-1} + (A_i - 1)/2))$, where: $S_{i-1} = A_1 + .. + A_{i-1}$.
   (c) If more than half of the A_i changed since the last time, update the ACPSA and go to step 2, with total number of iterations I set to the number of remaining iterations.

Slave :

1. **(a)** Send $Speed(P_i)$;
   (b) Receive V_i .
2. Obtain the number of processes in the run-queue Q_i and calculate A_i. If ($A_i \leq Threshold$), repeat step 2.
3. Send a request (containing its A_i).
4. Wait for a reply.
   - If more tasks arrive, compute the new tasks, go to step 2.
   - Terminate.

2.2. Hierarchical DTSS (HDTSS)

In a centralized scheme (DTSS), where a single node (the master) is in charge with the load distribution, performance degradation may occur when the problem size increases. This means that for a large problem (and a large number of processors) the master becomes a bottleneck. The access to the synchronized resources (variables) will take a long time, during which many processors will idle waiting for service, instead of doing useful work.

In HDTSS [13], instead of making one master process responsible for all the workload distribution, new master processes are introduced. Thus, the hierarchical structure contains a lower level, consisting of slave processes, and several superior levels, of master processes. On top, the hierarchy has an overall *super-master*. The level of slaves use the DTSS for load balancing.

Figure 1 shows this design for two levels of master processes. The slaves are using DTSS when communicating with their master. The $super\text{-}master \leftrightarrow master$ communication applies the DTSS algorithm with master replaced by super-master and slaves replaced by masters. The Masters do not perform any computation. They only assign tasks to slaves from the pool of tasks that they obtain periodically from the super-master. They communicate with the super-master only when they run out of tasks for their cluster. The dotted lines surround processes that can be assigned to the same physical machine, for improved performance.

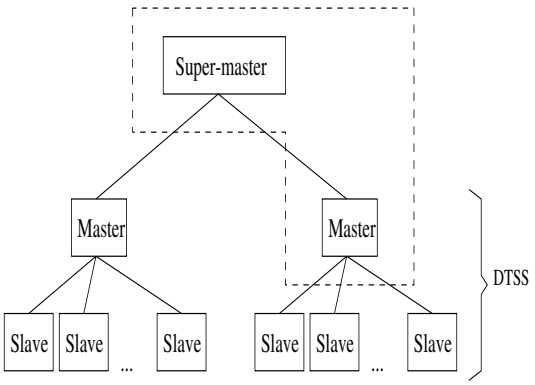

Figure 1. Hierarchical DTSS (two levels of masters)

HDTSS algorithm:

SuperMaster: Perform the DTSS-Master steps.
Master: Perform the DTSS-Master and DTSS-Slave steps.
Slave: Perform the DTSS-Slave steps.

2.3. Tree Scheduling (TREES)

TREES [14, 25] is a distributed load balancing scheme that statically arranges the processors in a logical communication topology based on the computing powers of the processors involved. When a processor becomes idle, it asks for work from a single, pre-defined partner (its neighbor on the left). Half of the work of this processor will then migrate to the idling processor. Figure 2 shows the communication topology created by TREES for a cluster of 4 processors. Note that P_0 is needed for the initial task allocation and the final I/O.

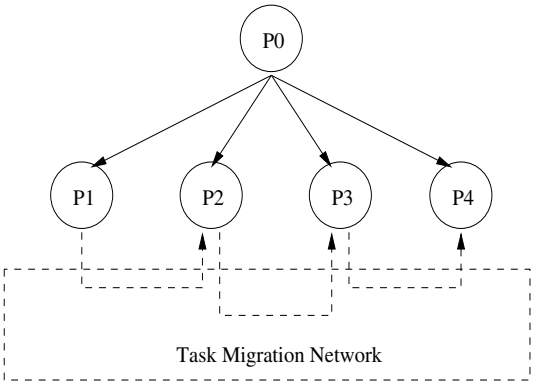

Figure 2. The Tree topology

An idle processor will always receive work from the neighbor located on its left side, and a busy processor will always send work to the processor on its right. The main success of TREES is the distributed communication, which leads to good scalability. The main disadvantage of this scheme is its sensitivity to the variation in computing power. The communication topology is statically created, and might not be valid after the algorithm starts executing. For example, if a workstation which was known to be very powerful becomes severely overloaded by other applications, its role of taking over the excess work of the slower processors is impaired. This means that the excess work has to travel more until reaching an idle processor or that more work will be done by slow processor, producing a large finish time for the problem.

3. Implementation and Test Results

3.1. Implementation

The scheduling schemes are implemented using the distributed programming framework offered by MPICH-G2 v1.2.6 [24]. In order to study the communication cost for bulk data transfers, the DTSS scheduling scheme is also implemented using MPIg and compared with its MPICH-G2 implementation. The MPIg version of the DTSS scheduling scheme was modified to replace the calls to the blocking MPI functions MPI_Send and MPI_Recv with calls to the non-blocking functions MPI_Isend and MPI_Irecv. To evaluate the performance of the scheduling schemes, we conducted single-site runs and cross-site runs on the TeraGrid. The single-site runs are conducted at the University of Chicago-Argonne National Laboratory (UC-ANL) site of the TeraGrid and the cross-site runs are conducted at UC-ANL and San Diego Supercomputer Center (SDSC) sites of the TeraGrid. The UC-ANL and SDSC TeraGrid sites have a cluster of IBM nodes, each with dual 1.5 GHz Intel Itanium-2 (IA-64) processors with 4 GB of physical memory, connected by Myrinet Gigabit interconnect network.

The test problem used is the Mandelbrot computation [27] for a matrix size (problem size) ranging from 8000×4000 to 30000×15000. The Mandelbrot computation is a doubly nested loop without any dependencies. The computation of one column of the Mandelbrot matrix is considered the smallest schedulable unit. In case of the master-slave schemes (DTSS and HDTSS), the master accepts requests from the slaves PEs and services them in the order of their arrival. For each request, it replies with a pair of numbers representing the number of columns and the interval of columns the slave PE should work on. In case of TREES, the master assigns the columns to the slave PEs in the initial allocation stage. Further requests of the slaves will be to their pre-defined partners.

We performed simulations with the number of slaves (p) ranging from 1 to 32. The size of the test problem is such

that it does not cause any memory swaps. So, the virtual powers (V_i, $i = 1, \ldots, p$) depend only on the processor speeds. To create a heterogeneous environment, we put an artificial load (one continuously running matrix multiplication process) in the background on $p/4$ slaves and three loads on another $p/4$ slaves. The slaves with three loads in the background are assumed to have V_i = 1, slaves with one load in the background are assumed to have V_i = 2 and the slaves without any load are assumed to have V_i = 4. This was verified by timing any program running on a single machine of the above three types.

In case of single-site runs (at UC-ANL) with p slaves, all the p slave PEs (slow and fast) and the master PEs (for DTSS, HDTSS and TREES) and the super-master PE (for HDTSS) are made to run at the same site. In case of cross-site runs for DTSS and TREES, the master PE is made to run at UC-ANL and $p/2$ of the slave PEs are at UC-ANL and the other $p/2$ are at SDSC. Among the $p/2$ slave PEs that are at different sites, half of them have V_i = 4, one-quarter have V_i = 2 and the other one-quarter have V_i = 1.

In the case of HDTSS cross-site runs, the slave PE distribution is similar to the slave distribution of cross-site DTSS and TREES. The super-master PE and one master PE are made to run at UC-ANL and the other master PE is made to run at SDSC. By this implementation of HDTSS, all the slave PEs at SDSC communicate with the master PE at SDSC which in turn communicates with the super-master PE at UC-ANL. This implementation of HDTSS reduces the communication between the sites which may not be possible in the case of DTSS and TREES.

3.2. Results

In this section, we present the results of the experiments performed. *Texec* denotes the total execution time measured on the master PE (super-master PE in the case of HDTSS) (Note: *seconds* is denoted by *sec* or *s*, and *minutes* by *m*).

Figure 3 shows the *Texec* of DTSS, HDTSS and TREES for problem size ranging from 8000 $\times$ 4000 to 30000 $\times$ 15000 when implemented on a single-site (UC-ANL). It can be observed that for all problem sizes, HDTSS shows better load balance and performance compared to DTSS and TREES.

In Figure 4, we present the cross-site (UC-ANL and SDSC) implementation results of DTSS, HDTSS and TREES. It can be observed that the performance of all the schemes is similar to the single-site performance. For all problem sizes, HDTSS shows superior performance compared to DTSS and TREES.

In the single-site and cross-site runs, the total communication time of the slaves with the master is around 2 and 4 seconds respectively for all the schemes for large problem sizes. This low value in communication time is because

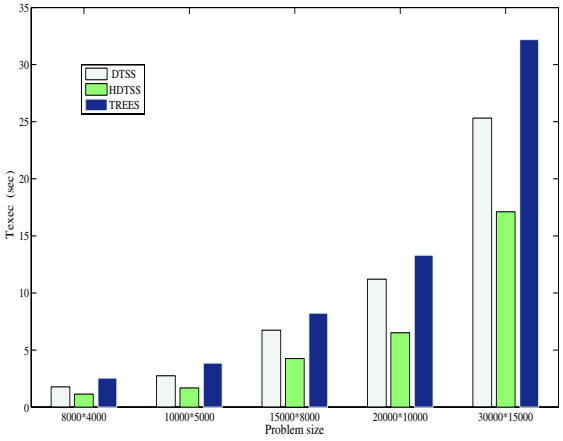

Figure 3. Single-site execution times of DTSS, HDTSS and TREES for various problem sizes

the slaves does not send the computed results back to the master. They only send a request for work and receive a reply. These messages are small in size. In case of HDTSS, the communication time between the super-master and local master is around 0.001 sec whereas the communication time between the super-master and the master on the remote site is around 0.1 sec. This low communication between the super-master and the masters is because, the chunk sizes allocated by the super-master to the masters are very large when compared to the chunk sizes allocated by the masters to the slaves. So, the masters make fewer requests to the super-master and the communication times are low.

Figures 5, 6 and 7 compare the single-site and cross-site performance of DTSS, HDTSS and TREES respectively. It can be observed that the multi-site *Texec* of all the schemes is very much comparable with their single-site *Texec*. This is because, cross-site communications are based on MPICH-G2 library. MPICH-G2 selects the most efficient communication method possible between two PEs. It automatically selects TCP for intermachine messaging and vendor-supplied MPI for intramachine messaging. These results show that the scheduling schemes are adaptable to a grid environment.

In Figure 8, we present the speedup of all the schemes with increasing number of slave PEs when implemented on a single-site (UC-ANL). We computed the speedup according to the equation: $S_p = min\{T_{p_1}, T_{p_2}, \ldots, T_{p_p}\}/T_p$ where T_{p_i} is the execution time on one PE and T_p is the execution time on p PEs. It can be observed that, as the number of PEs increase, the speedup of all the schemes improves which shows that the schemes are scalable. How-

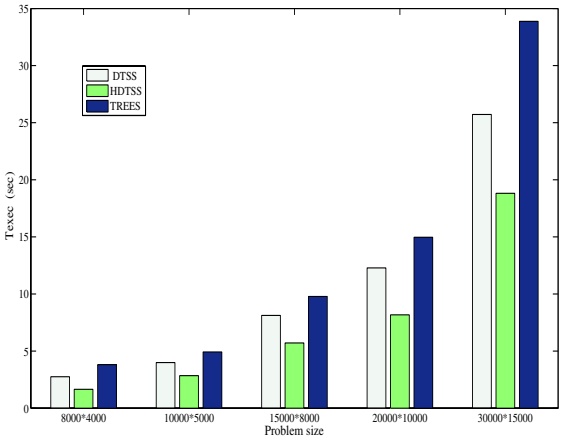

Figure 4. Cross-site execution times of DTSS, HDTSS and TREES for various problem sizes

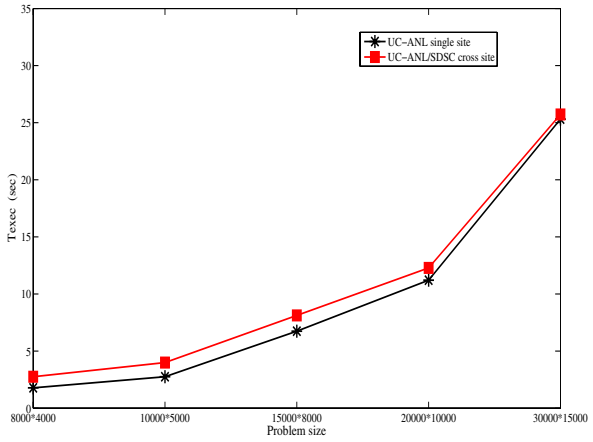

Figure 5. Comparison of single-site and cross-site execution times of DTSS for various problem sizes

ever, HDTSS is the most scalable.

Figure 9 shows the speedup of all the schemes with increasing number of slave PEs when implemented across the sites (UC-ANL and SDSC). Again, HDTSS is the most scalable scheme. The cross-site speedups of the schemes are comparable to the single-site speedups which shows their adaptability to a grid environment.

To study the communication cost for bulk data transfers, we simulated DTSS when computed data is sent back to the master by the slaves for problem sizes ranging from 50000 $\times$ 50000 to 100000 $\times$ 100000. The MPIg implementation of DTSS utilizes threads to perform wide-area communication while the application proceeds with computation.

In Table 1, we present the total execution time and communication time (*Tcomm*) of MPICH-G2 and MPIg implementation of DTSS for 32 slaves. The simulations were made across sites. From Table 1, it can be observed that there is a significant decrease in the *Tcomm* of DTSS MPIg implementation compared to DTSS MPICH-G2 implementation for various problem sizes.

On a single core system, where the application must share the processor core with the communication threads, the time to perform the computation may increase as the communication threads interrupt to send or receive more data. The performance impact depends on several factors; however, in most cases, the communication threads are not highly active and thus the performance impact to the computation is typically small. Furthermore, because communication only requires use of the processor for a small percent of the communication time, the reduction in overall execution time resulting from the overlap of communication and computation normally far exceeds the cost of the interruptions. It should also be noted that for grid applications whose computation time exceeds its communication time, which is the best case scenario, the overlap in computation and communication will typically result in virtually no observable time spent in communication. In other words, the message passing time will be completely masked by the computation.

Table 1. Execution and communication times of MPICH-G2 and MPIg implementation of DTSS

Problem size	MPI Impl.	Texec	Tcomm
50000 $\times$ 50000	MPICH-G2	1m 58s	46s
50000 $\times$ 50000	MPIg	1m 28s	14s
75000 $\times$ 75000	MPICH-G2	3m 27s	1m 14s
75000 $\times$ 75000	MPIg	3m 20s	36s
100000 $\times$ 100000	MPICH-G2	6m 36s	2m 4s
100000 $\times$ 100000	MPIg	6m 27s	1m 14s

On a multi-core system, dedicating one or more cores on which the communication threads may execute can nearly eliminate the impact of those threads. Although, reserving a core for communication may only be useful if sufficient communication exists to keep the core busy; otherwise, application performance may still peak when all cores are performing computation, even though the communication threads will occasionally interrupt that computation. The exact ratio of cores available for computation and cores dedicated to communication is an issue for further study.

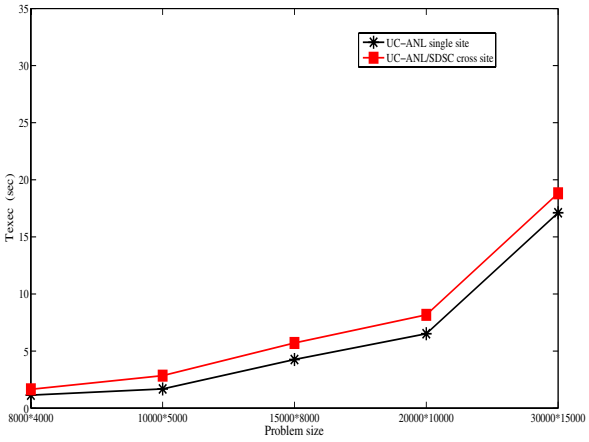

Figure 6. Comparison of single-site and cross-site execution times of HDTSS for various problem sizes

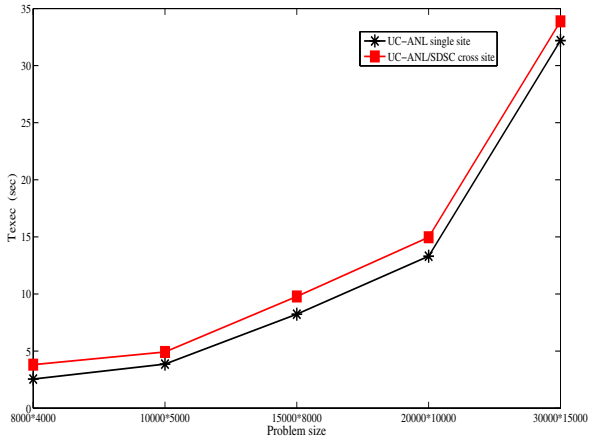

Figure 7. Comparison of single-site and cross-site execution times of TREES for various problem sizes

4. Conclusions and Future Work

In this paper, we have implemented three dynamic loop scheduling schemes on the TeraGrid using the MPICH-G2 communication platform. We also implemented the DTSS scheduling scheme in MPIg and compared it with the MPICH-G2 implementation. There we found a significant improvement in communication time when using MPIg and its multi-threaded implementation. We performed simulations on two TeraGrid sites and compared them with single-site simulations. Our results show that the cross-site performance of the schemes is very comparable to their single-site performance showing the adaptability of these schemes to a multi-site grid environment. The hierarchical scheme showed superior performance compared to the other two schemes.

In future work, we plan to use a multi-thread approach to have the master's in DTSS and HDTSS implemented on one of the slave PEs in order to make efficient use of the available PEs. We also plan to implement and compare other proposed scheduling schemes (e.g., factoring) using MPICH-G2 and MPIg and to involve more sites and more computers in the simulations.

Acknowledgements

The authors acknowledge the support by the National Science Foundation under grants CCR-0312323 and OCI-0330664. Computer time for the TeraGrid was provided by the University of Chicago-Argonne National Laboratory (UC-ANL) and the San Diego Supercomputer Center (SDSC).

References

[1] http://www.mcs.anl.gov/mpi/mpich2.

[2] European commission enabling grids for E-sciencee. http://public.eu-egee.org.

[3] The Globus alliance. http://www.globus.org.

[4] Japan national research grid initiative. http://www.naregi.org/index-e.html.

[5] United States department of energy earth system grid. http://www.earthsystemgrid.org.

[6] United States department of energy open science grid. http://www.opensciencegrid.org.

[7] United States National Science Foundation Teragrid. http://www.teragrid.org.

[8] Mpi: A message-passing interface standard. *Message Passing Interface Forum, International Journal of Supercomputer Applications*, 8(3/4):159–416, 1994.

[9] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Secure, efficient data transport and replica management for high-performance data-intensive computing. In *IEEE Mass Storage Conference*, 2001.

[10] W. Allcock, J. Bresnahan, R. Kettimuthu, and J. Link. The Globus extensible Input/Output system (XIO): A protocol independent IO system for the grid. In *Proceedings of the Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models held in conjunction with International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, CO, April 2005.

[11] G. Allen, T. Dramlitsch, I. Foster, N. T. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Supporting efficient execution in heterogeneous distributed computing environments with cactus and globus. In *SC' 2001*, Denver, CO. Gordon Bell Prize Winner, Special Category, November 10-16, 2001.

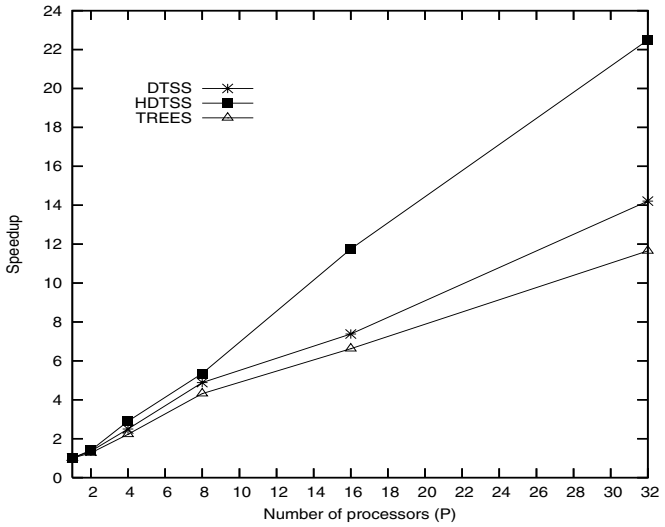

Figure 8. Speedup of DTSS, HDTSS and TREES with number of processors (Single-site)

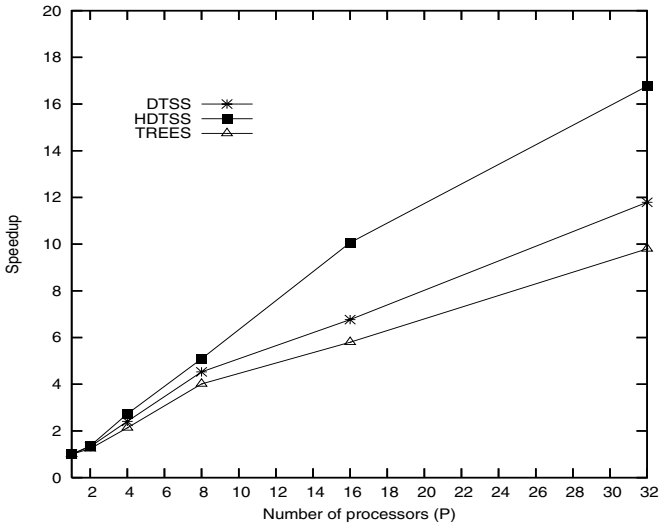

Figure 9. Speedup of DTSS, HDTSS and TREES with number of processors (Cross-site)

[12] A. T. Chronopoulos, S. Penmatsa, J. Xu, and S. Ali. Distributed loop-scheduling schemes for heterogeneous computer systems. *Concurrency and Computation: Practice and Experience*, 18(7):771–785, 2006.

[13] A. T. Chronopoulos, S. Penmatsa, N. Yu, and D. Yu. Scalable loop self-scheduling schemes for heterogeneous clusters. *Intl. Jrnl. of Computational Science and Engineering*, 1(2/3/4):110–117, 2005.

[14] S. P. Dandamudi and T. K. Thyagaraj. A hierarchical processor scheduling policy for distributed-memory multicomputer systems. In *Proc. of the 4th International Conference on High-Performance Computing*, pages 218–223, Nagoya, Japan, 1997.

[15] S. Dong, N. T. Karonis, and G. E. Karniadakis. Grid solutions for biological and physical cross-site simulations on the teragrid. In *Proc. of IEEE Intl. Parallel and Distributed Proc. Symp.*, Rhodes Island, Greece, April 25-29, 2006.

[16] Y. W. Fann, C. T. Yang, S. S. Tseng, and C. J. Tsai. An intelligent parallel loop scheduling for parallelizing compilers. *Journal of Information Science and Engineering*, pages 169–200, 2000.

[17] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP Intl. Conf. on Network and Parallel Computing*, pages 2–13, Springer-Verlag LNCS 3779, 2006.

[18] I. Foster and C. Kesselman. The globus project: A status report. In *IEEE Proc. of the Heterogenous Computing Workshop*, pages 4–18, 1998.

[19] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kauffman, 1999.

[20] Y. Gu and R. L. Grossman. Supporting configurable congestion control in data transport services. In *SC 2005 Conference*, Seattle, WA, November 12-18, 2005.

[21] E. He, J. Leigh, O. Yu, and T. DeFanti. Reliable blast udp: Predictable high performance bulk data transfer. In *IEEE Cluster Computing*, 2002.

[22] N. Karonis, B. de Supinski, I. Foster, W. Gropp, W. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computing networks to optimize collective operation performance. In *14th Intl. Parallel and Distributed Proc. Symp.*, pages 377–384, Cancun, Mexico, May 2000.

[23] N. Karonis, M. Papka, J. Binns, J. Bresnahan, J. Insley, D. Jones, and J. Link. High-resolution remote rendering of large datasets in a collaborative environment. *Future Generation of Computer Systems (FGCS)*, 19(6):909–917, Aug 2003.

[24] N. T. Karonis, B. Toonen, and I. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, May 2003.

[25] T. H. Kim and J. M. Purtilo. Load balancing for parallel loops in workstation clusters. In *Proc. of Intl. Conference on Parallel Processing*, pages 182–190, Bloomingdale, IL, USA, 1996.

[26] K. Mahinthakumar, M. Sayeed, and N. T. Karonis. Grid enabled solution of groundwater inverse problems on the teragrid network. In *High Performance Computing Symp. (HPC 2006)*, Huntsville, AL, April 2-6, 2006.

[27] B. B. Mandelbrot. *Fractal Geometry of Nature*. W.H. Freeman & Co, 1988.

[28] B. Y. Mirghani, M. E. Tryby, D. A. Baessler, N. T. Karonis, R. S. Ranhthan, and K. G. Mahinthakumar. Development and performance analysis of a simulation-optimization framework on teragrid linux clusters. In *The 6th LCI International Conference on Linux Clusters: The HPC Revolution 2005*, Chapel Hill, NC, April 26-28, 2005.