International Journal of Grid and Distributed Computing
Vol. 1, No.2, March, 2009

SymWorkManager: A Novel HPC Multi-thread Programming Model

Xiaohui WEI[1], Hongliang LI[1], Shaocheng XING1, Ji LI[1], Zane Zhenhua HU[2], Shutao YUAN[2]
[1] *College of Computer Science and Technology, Jilin University, ChangChun, P.R.C.*
[2] *Platform Computing Inc. Markham, Ontario, Canada*
weixh@jlu.edu.cn, simonlee@email.jlu.edu.cn, hu@platform.com, shutao@platform.com

Abstract

Multi-thread is a mature and widely-used programming mode for multitasking applications developing, especially in Java. However java.lang.Thread is not designed for HPC parallel programming. Java multi-thread program is confined within single compute host. CommonJ Work Manager is a joint specification proposed by IBM and BEA. Work Manager is a higher-level alternative to using Java Thread. Originally, Work Manager is used to solve the constraint of opening independent threads in J2EE container. Although there are some enterprise servers such as WebLogic and WebSphere which have supported Work Manager, they are not aim at large scale parallel applications. They do not have a grid-enabled and expandable architecture, and the multi-thread programs are still executed on single compute host. Currently, grid-enabled parallel programming model like MPICH-G2 is limited with C/C++ implementations. There are no official bindings for MPI with Java. In this paper, we present a grid-enabled multi-thread programming model: SymWorkManager to enable multithread programs to run on distributed environments. SymWorkManager is designed as a high performance computing (HPC) implementation of CommonJ Work Manger specification using Platform Symphony. It provides J2EE developers a simpler way to program parallel multi-thread applications under a HPC platform.

1. Introduction

In J2EE, threads are container-managed resources like CPUs and DB connections. A J2EE container can serve multiple users concurrently via the thread pool. However, opening independent threads within a single J2EE container is not recommended to the application developers. The multithread applications in J2EE is deadlock-prone and may cause high overhead of creating threads on-the-fly. J2EE developers have been harassed by such constraint. To resolve this problem, IBM and BEA came up with a joint specification, JSR 237: CommonJ Work Manager for Application Servers[2,4]. Work Manager provides a higher level of abstraction for concurrent programming than java.lang.Thread.

The current implementations of Work Manager by WebLogic and WebSphere are confined to limited computational capability. These servers are designed for web applications rather than parallel computing. In this paper, we will present a novel HPC enabled multi-thread programming model, SymWorkManager. With the new model, the developers just use the **CommonJ Work Manager** specification as the programming interfaces to submit jobs and collect results. And the multiple threads of an application will be distributed to **Platform Symphony** [5] clusters automatically. SymWorkManager bridged the gap between the multi-thread programming model and HPC. Figure1 below shows the basic structure of the new model.

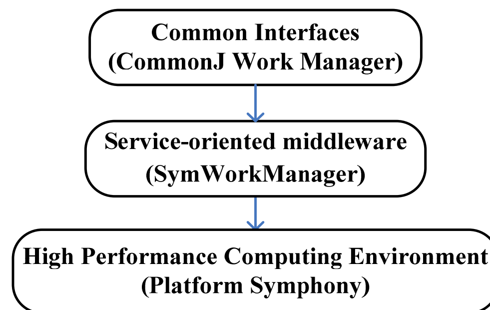


Figure 1. HPC Multi-thread Programming Model

The rest of this paper is organized as follows: section 2 gives a brief overview of the related works; section 3 presents the architecture design and implementation details of SymWorkManager. In section 4, we evaluate SymWorkManager's performance via experiments. The last section concludes the paper and discusses our future plans.

2. Related works

Currently, there are different kinds of parallel programming models to support the distributed application developing.

MPI [7] is the most widely-used message passing parallel programming model. MPICH-G2 [8, 9] is a grid-enabled implementation of MPI. It is designed based on DUROC protocol [10] of Globus toolkit. In MPICH-G2, a fixed set of processes is created at program initialization, and one process is created per processor. While powerful and feature rich, the API and programming models for MPI are relatively complex. The application developers need to design the computing pattern and the message transfer carefully. And prevailing implementations of MPI are designed in C/C++ language. There are no official bindings for MPI with Java.

Recently, Ruby [11] attracts attention as an object oriented scripting language for agile development. Distributed Ruby (dRuby) [12] is a library that allows you to send and receive messages from remote Ruby objects via TCP/IP. Using dRuby, developers can possibly build new parallel programming models.

Work Manager for Application Servers provides a high-level programming model that enables applications to logically execute multiple work items concurrently under the control of the container. These work items execute out of a thread pool managed by a container.

Work Manager is a quit simple service that only handle three things: work submissions, executing the work and returning results. As a result of the diversity of user-defined works, Work Manager specification become a powerful universal service interface, which means that a single implementation of Work Manager can deal with all kinds of user-works.

The Work Manager API is comprised of five primary interfaces: **WorkManager**, **Work**, **WorkItem**, **WorkListener**, and **WorkEvent** [3]. The WorkManager interface provides a set of schedule() methods whereby Work can be scheduled for execution. The WorkManager then returns a WorkItem, which can be used to get the status of the in-flight Work. And the WorkManager enables returning execution results synchronously and asynchronously.

The structure of Work Manager is shown as figure2.

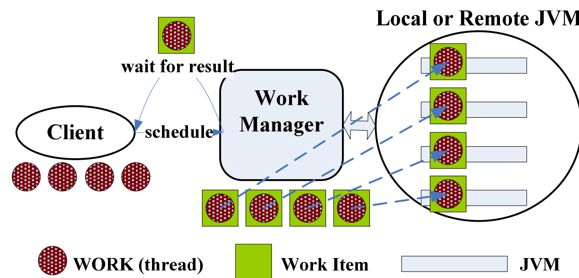


Figure 2. Work Manager Proposal

A user-defined Work contains the main logic functions of an application. Developers could program multi-thread program using these Works. Each Work represents a thread.

There are already implementations of Work Manager under WebSphere and WebLogic platform already [4, 13]. But these platforms provide limited computational capabilities with single server or cluster.

Different WorkManagers vary in the diverse fashion of implementations. According to the specification the underlying platform of Work Manager can be multiform which makes it more flexible.

In this paper, in order to fit the parallel programming model, we provide an implementation of the specification with a grid-enable platform, Platform Symphony.

3. Design and implement

3.1. Platform Symphony

3.1.1. Overview: Platform Symphony is a distributed computing framework that makes it practical to develop and manage distributed computing services in a coherent way. Symphony is a high performance, grid-enabled, service-oriented architecture, which can provide high performance parallel computing (HPC) based on its clusters and compute hosts. By distributing computations in parallel to nodes on a grid and using a well designed scheduling strategy, Symphony accelerates applications. Symphony provides a simpler way to perform parallel distributed calculations that:

a) Determines appropriate execution hosts and adjusts application processes at runtime
b) Tolerates node failures and automates recovery
c) Presents a simple, guided model that does not bother programmers with low-level details
d) Is implementation and language agnostic
e) Exhibits low latency [6]

3.1.2. Develop distributed program using Symphony: A standard Symphony applications program consists of three parts: service program, client program and communication messages.

The **service program** provides an independent function. It is where the commonly-used business logic is encapsulated. Symphony administrators must upload the service program onto the platform beforehand. After that, service programs wait for client programs to access. Once a service program is in-flight, Symphony brings up a session instance (SI) on one of the compute host.

The **client program** is the client-specific applications logic of a user application, which can use several service programs at the same time. Each service program has its unique service name as the identifier. A client program use service names to trigger service programs.

While executing, there must be some certain ways to maintain the communications between client and service programs, which is the **communication messages**. Every service program can have two custom classes as the input and output messages according to its own function. As a result, a client program must understand the "cryptogram" in order to use certain service programs.

The figure 3 shows how a typical Symphony application works.

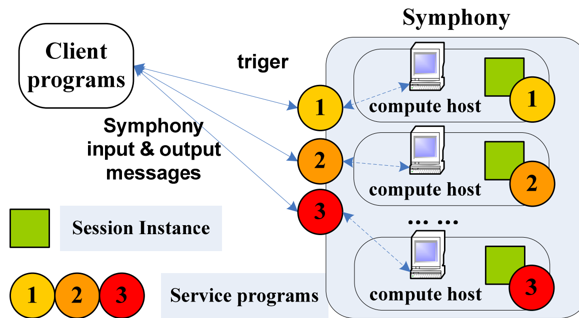


Figure 3. A Typical Symphony Application

Symphony's Service-Oriented Applications Middleware (SOAM) provides the applications development SDK and workload management necessary to develop and run reliable service-oriented applications including method invocation and binary execution tasks [6]. However, in order to take advantage of the high performance computing of Symphony, developers must learn SOAM APIs and have the knowledge of Symphony applications programming to write both service and client programs. It is quit a time consuming task.
Therefore, we implement a new middleware compliant to the Work Manager specification, SymWorkManamger, for Platform Symphony environment. SymWorkManager enables CommonJ developers to enjoy high performance and concurrent multi-thread developing without any Symphony programming knowledge. SymWorkManager is capable of supporting great number of concurrent tasks due to its extendable architecture.

3.2. SymWorkManager architecture design

In this section, we will discuss the architecture design of SymWorkManager. Briefly, our work is a bridge between the high level programming interfaces (CommonJ Work Manager) and the underlying computing platform (Platform Symphony). As introduced before, a Symphony application is consist of three parts. According to that, SymWorkManager has the similar structure, which contains: SymService, SymWorkManager and Work.

SymService is a back-end service which is a typical Symphony service application. It executes on Symphony compute hosts under SOAM's management to actually run Works and return results to SymWorkManager. Each scheduling of Work triggers a Session Instance (SI) to run SymService program on one of the compute hosts. Symphony's scheduling strategy will decide which compute host will execute SymService.

For a Symphony application, the actual class files and other relevant files must be pre-uploaded onto Symphony as part of the service program before execution by system administrators. After that, Symphony management hosts will automatically transfer these files onto compute hosts while running the service program. So administrator must deploy SymService program onto Symphony before the whole system runs.

SymService is a common service that could accommodate to any user-defined Works without being modified or re-deployed. This is the dynamic-deploy feature of SymWorkManager. We will discuss this later in our paper. In this way, we will only have to do the deployment of SymService program once.

SymWorkManager is a front-end application that implements all kernel functions that Work Manager specified. It can accept client request and collect results for client program. The SymWorkManager supports both synchronous and asynchronous applications. After an application submits work items to SymWorkManager for execution, the applications will gather the computing results. The SymWorkManager provides common "join" operations, which are block methods to wait for a specific Work Item or all of them to finish. And also SymWorkManager implements WorkListener in Work Manager specification to enable asynchronous functions.

SymWorkManager can be deployed on any J2EE server hosts, such as a JBoss server. The administrator maintain one or more instances of SymWorkManager, associate a JNDI name with each one and provide these SymWorkManagers to CommonJ consumers.

Work is a common interface that is specified in Work Manager. A Work represents a java.lang.Thread. Like programming a multi-thread application in traditional Java, developers must design their own Works. They need to put their application logic into Works, so different users will implement different Work functions. But the developers must follow the unified interfaces for Work to design their multi-thread programs. Each Work has a function: run(), as the start point of user logic like start() in java.lang.Thread. Of course, these Works could be executed on any compute host of the computational platform.

These three parts are the most significant components of our model. Basically, our project is focusing on SymWorkManager and SymService. Work is designed by developers themselves. Figure 4 illustrates how they react with each others.

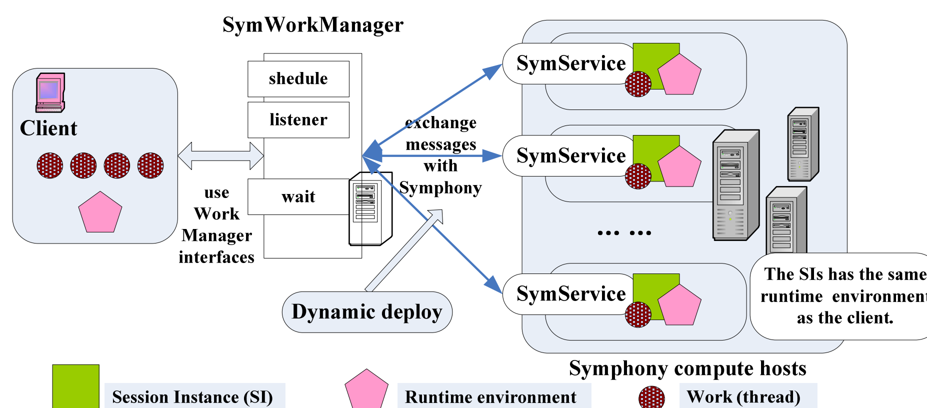


Figure 4. The Architecture of SymWorkManager

SymWorkManager enables J2EE-based applications (including Servlets and EJBs) to schedule Work Items for concurrent execution, which will improve the throughput and

response time significantly. The client developers do not have to concern about how and where their works are executed. They can just focus on their application logic.

The new model is transparent to CommonJ developers. To take advantage of our model, Java developers just need to design their applications by following CommonJ Work Manager specification. SymWorkManager is completely compatible to current CommonJ Work Manager applications. CommonJ applications can operate properly with SymWorkManager like using other existing Work Manager implementations. CommonJ developers do not have to modify their programs at all to fit SymWorkManager.

3.3. SymWorkManager implementation

In this section, we will present the implementation details of SymWorkManager. As mentioned before, SymWorkManager and SymService is the most important component of our middleware. We will discuss:

1. Dynamic-deploy in both SymWorkManager and SymService when submitting and executing Works
2. SymCommonData Sharing
3. Gathering Work results in SymWorkManager

3.3.1. Dynamic-deploy: Before actually running an application in distributed environment, the executable programs must be pre-deployed onto compute hosts in advance. When there are amounts of computing nodes in the system, such deployment is usually time consuming. SymWorkManager implements an automatic mechanism to dynamically deploy the runtime environment of every Work onto Symphony platform.

SymWorkManager is a universal service component that could work with all kinds of user-defined Works. Each Works has its own runtime environment like executable class files, input files and environment variables according to its own function. We must decide which information are significant when the Work running.

In order to bring up a user-defined Work instance, SymService on compute host must have the entire executing environment of the Work. Under the management of Symphony, a WorkItem could execute on any compute host of the platform. In this condition, we must have the whole executing environment of a user-defined work transferred from client program to Symphony during scheduling to avoiding the pre-deployment.

Correspondingly, we must unpack the runtime information and make sure that SymService has the same environment as client program.

Therefore, there are three essential tasks here: gathering runtime information, transferring runtime information from client to servers, and unpacking common files.

Gathering

To gather information, we use Java URLClassLoader to get the path of all the relative files of Work such as class files, jar packages and other ordinary files. And SymWorkManager recursively saves all these files and their directory information in SymCommonData. SymCommonData is a class designed to carry runtime information of Work. As shown is Figure 5, SymCommonData contains executable class files, input files and environment variables.

SymCommonData Transfer

We could use another feature of Symphony to accomplish the transferring, which is common data. Symphony enables developers to use a global read-only common data around Service Instances (SIs). SymCommonData is based on Symphony common data. We will only have to fill SymCommonData with Works' runtime information. And Symphony will transfer a copy of SymCommonData to the running SI to share data among tasks. SymCommonData is accessible to all the instances of SymService.

SymCommonData Unpacking

When executing a WorkItem, Session Instance needs the files shipped with SymCommonData to ensure that Work runs properly. So right after SymService gets started, we must unpack all these files from SymCommonData to SI's workaround one by one. To make sure the executing environment in the SI's workaround is as same as those on the client side, we have to rebuild the whole directory structure. Because of that, the path information must be stored in SymCommonData as well. Also you can see in figure 5, under certain conditions there could be several SIs running on same compute host simultaneously. These SIs will share the same workarounds. So we only need to unpack common data once on such compute host to reduce the time and space overheads.

Figure 5 illustrates how the whole system works.

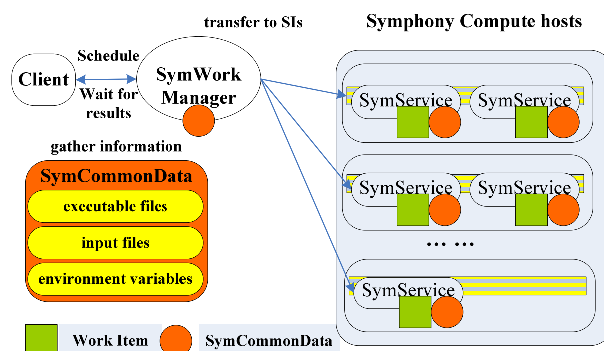


Figure 5. Using SymCommonData to Share Executing Environment

The process of gathering and transferring application runtime information could cost considerable overheads especially when the common file's size is big. And also there would be a dilemma in deciding when and whether to delete these common files from the compute hosts. Hence, we introduce the below two environment flags to make a better control of the application deployment:

DYNAMIC_PASSIN_CLASS: SymWorkManager passes common data and global environment from client site to compute hosts when the flag has the value of TRUE. Otherwise, when the flag's value is FALSE, the SymCommonData carries only global environment. Administrators have to upload their application files themselves.

WHEN_REMOVE_CLASSFILE: This flag has the value among "onSessionLeave" and "never". When the flag's value is "onSessionLeave" the common files should be removed when Session Instance is finished. And sometimes we would also want to use "never" flag to just leave these file for further execution instead of deleting them.

These two flags are fetched from system environment and carried in SymCommonData as global environment. Both SymWorkManager and SymService will use them.

3.3.2. SymCommonData sharing: SymWorkManager makes the distributed computing environment transparent to CommonJ developers. Symphony's scheduling strategies will balance the workloads of compute hosts to provide a higher throughput and shorter response times. SymWorkManager makes use of those strategies to manage the underlying computing grid. And CommonJ developers or consumers do not need to be bothered about that.

For some powerful compute hosts, SymWorkManager may startup multiple instances of a service program on them. So there could be multiple instances of one SymService running on the same machine simultaneously.

In this case, the multiple instances of a SymService running on same computing node can share one copy of the runtime information rather than maintain multiple redundant copies. However, as each instance of SymService controls the creation and elimination of the application runtime information by itself, there could be conflicts among the multiple SymService instances. Such conflicts will result in the executions of SymService unpredictable. To prevent this situation, SymService use the following mechanism to coordinate the runtime information creating and deleting operations.

First, the files to restore an application's runtime info will be named unique in global. Secondly, an "index" variable is introduced to control the file creating and deleting operations for each application on a computing node. Once a SymService is going to deploy the runtime info for an application, it will check the corresponding "index" value first. The files are created if and only if the "index" value is 0. Otherwise, the SymService just increases the "index". Similarly, if the "index" is larger than 1, the file deleting operation will be skipped. And such file deleting and creating are implemented as atomic operations by SymService. Figure 6 shows how it works.

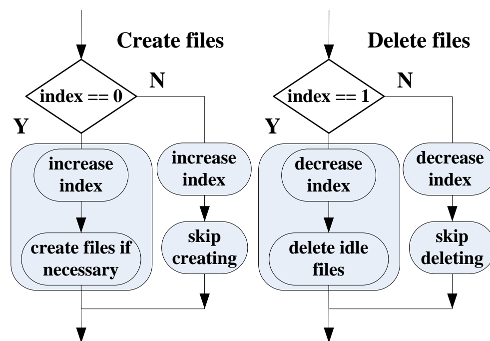


Figure 6. Use Index to Control Operations

3.3.3. Gathering work results: SymWorkManager supports both synchronous and asynchronous ways to wait for task results from Java VJMs defined in Work Manager specification. There are two synchronous functions: waitForAll() and waitFor). A Any(client application can block and poll for these results to implement the synchronous wait methods. Comparatively, the asynchronous functions are a little complicated. In the asynchronous ways, the WorkManager does not wait for results, but the results trigger WorkListener's callback functions.

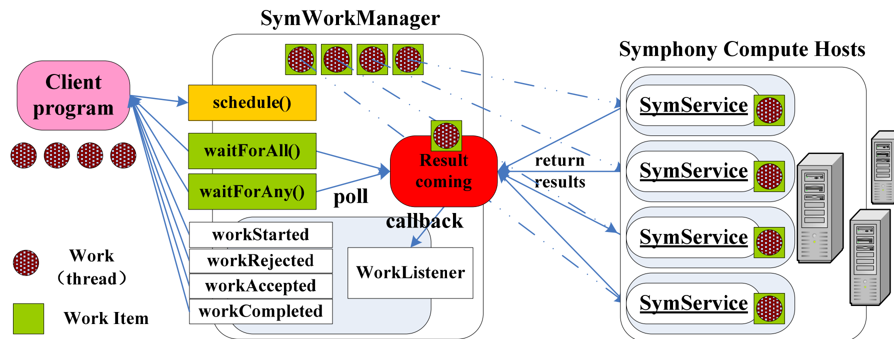


Figure 7. Gathering Task Results in SymWorkManager

Figure 7 illustrates how SymWorkManager gathers the execution results for client programs.

In Work Manager specification, WorkListener contains a series of callback functions. These methods are called when certain events arises.

1. ***workAccepted()*** : This is called when the Work is accepted for dispatching
2. ***workCompleted()*** : This is called once Work.run returns.
3. ***workRejected()*** : This is called when the Work cannot be processed prior to starting but after accept.
4. ***workStarted()*** : This is called when the Work is about to start [3].

Events like WORK_ACCEPTED, WORK_COMPLETED, WORK_REJECTED and WORK_STARTED will request for these callback methods and carry WorkItem as the parameter. Because of that, one WorkListener can watch many WorkItems at the same time. In SOAM's applications development SDK, there are synchronous and asynchronous ways to accomplish communication between service and client programs likewise. In order to make the SymWorkManager efficient, we choose to use asynchronous methods in SOAM to implement both "waitfor" functions and WorkListener's callback functions.

4. Experiment results

In this section, we will evaluate SymWorkManager by comparing its performance on a single server with that in a HPC cluster. Because SymWorkManager is designed to improve the performance of multi-thread applications, our experimental client application is a Java multi-thread program. Each thread of the program requests a number of Works to SymWorkManager. The control flags are:

DYNAMIC_PASSIN_CLASS = "TRUE"
WHEN_REMOVE_CLASSFILE = "onSessionLeave"

The first experiment will test the performance of SymWorkManager in HPC environments, and the second one will show the overheads of the dynamic-deploy feature of SymWorkManager.

Experiment 1:

In this test, the client program sends 60 to 3000 Works requests to Work Manager. Each Work carries only small size of data. SymWorkManager is deployed in two clusters with 3 and 6 compute nodes in respective. Each compute node has one processor. In Chart 1, the X axis is the number of Works and the Y axis is the total executing time in second.

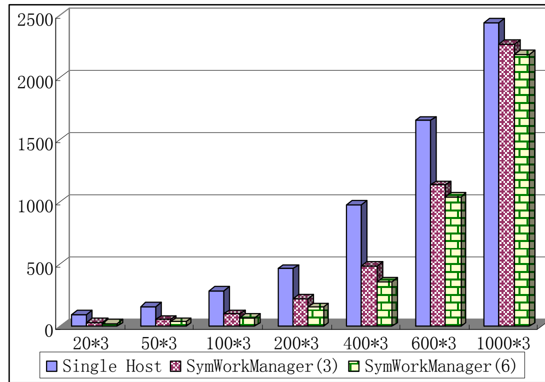


Chart 1. Multi-Thread Program Runs on Single Host and SymWorkManager

From Chart 1, when there are 60 to 1200 concurrent Works, SymWorkManager accelerates the multi-thread applications significantly. However, when the threads exceed to 1800, there is no much performance improvement by SymWorkManager.

Chart 2 illustrates the speedup ratio when SymWorkManager handle Work requests using 3 or 6 processors.

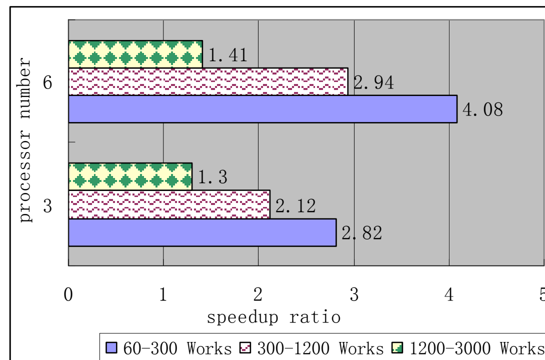


Chart 2. SymWorkManager with Different Number of Processors

The results shows that when there are a great number of concurrent job requests, the capability constrains of the host on which SymWorkManager is running will become the bottle neck of the whole system. To resolve such problem, the system administrator can set up multiple SymWorkManager services on different hosts in a cluster. And we are also working on automatic load sharing mechanism between SymWorkManager services.

Experiment 2:

In this experiment, the test client programs are similar to experiment 1 but with different data size, 15MB, 100MB and 200MB. The SymWorkManager will transfer these applications data with SymCommonData from client site to compute hosts. Chart 3 compares the run time of the same application with different sizes of SymCommonData. Compared with Figure 1, the extra overheads are used by data transfer and unpacking.

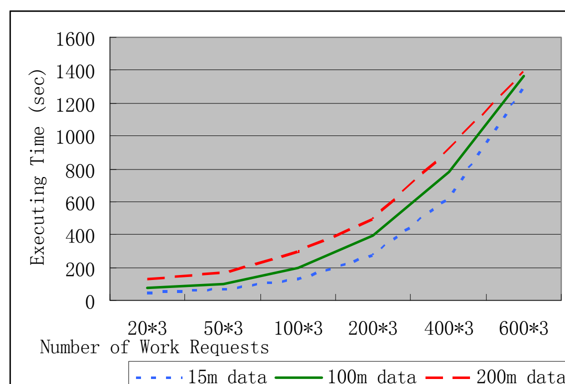


Chart 3. Programs with Different Size of Data Execute on SymWorkManager

From Chart 3, we can see the different data size does not result in significant difference in application's run time. In particular, when the number of Works is huge, each compute host will execute more workloads. Some instances of SymService may share the same common data on the host. In that case, the overheads caused by file creating and deleting operations would be reduced accordingly.

Overall, the SymWorkManager can accelerate multi-thread applications by exploiting the computing power of distributed environment with acceptable extra overheads involved.

5. Conclusion and future work

This paper has proposed a grid-enable multi-thread programming model for HPC environment. In this model, developers use CommonJ Work Manager as the parallel programming interfaces. Work Manager is a higher-level alternative of java.lang.Thread. SymWorkManager is the bridge between high level programming interfaces and underlying computing platform. SymWorkManager is a grid-enabled implementation of CommonJ Work Manager specification. Developers can build distributed multi-thread applications using SymWorkManager, which provides high throughout and quick response time. Additionally, SymWorkManager supports a dynamic-deploy mechanism to ensure that the whole system is transparent to CommonJ developers and consumers. The existing CommonJ applications do not have to change at all to fit the model.

As there is no data sharing mechanism defined in Work Manager specification, SymWorkManager only enable read-only common data among in-flight Works for now. We are planning to extend Work Manager with a sharing memory mechanism that enables running Works communicate with each other more conveniently. We also plan to support shared file system among Works with Gfarm [8] data grid.

Acknowledgement

The work presented in the paper is a joint project between Platform Computing Inc. and JiLin University. The related source code and design documentations can be downloaded at website http://www.hpccommunity.org/.

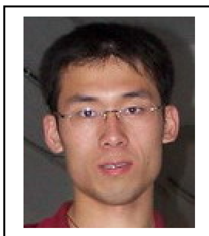
The authors would like to acknowledge the support from Platform Computing Corporation, and the support from the China NSF Grant No.60703024 and Jilin Department of Science and Technology Grant No.20070122 and 20060532.

References

[1] Rahul Tyagi. The Work Manager API: Parallel Processing Within a J2EE Container. www.devx.com .08/2005
[2] Revanuru Naresh and Billy Newport. JSR 237: Work Manager for Applications Servers 2003
[3] John Beatty, Chris D Johnson, Revanuru Naresh. Commonj-TimerAndWorkManager-Specification-v1.1. 03/2004
[4] http://dev2dev.bea.com/wlplatform/commonj/twm.html Timer and Work Manager for Applications Servers specification page.
[5] Platform. ENTERPRISE GRID: The Next Generation Architecture for Capital Markets. 2006
[6] Platform. A Developer's Guide to Building High Performance Service-Oriented Applications A technology strategy whitepater. 2007
[7] Message Passing Interface Forum, "MPI: A message-passing interface standard", Internat. J. Supercomput. Appl. 8(3/4), 165-414, 1994.
[8] N.T. Karonis, B. Toonen, I. Foster "A MPICH-G2: A Grid-enabled implementtation of the Message Passing Interface," Journal of Parallel and Distributed Computing, 2003.
[9] K. Czajkowski, I. Foster, and C. Kesselman. Resource Co-Allocation in Computational Grids. Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8), pp. 219-228, 1999.
[10] http://www-unix.mcs.anl.gov/mpi/mpich1/ MPICH web site.
[11] Dave Thomas, Chad Fowler, and Andy Hunt. Programming Ruby. PragmaticBookshelf, City, 2004
[12] Masatoshi Seki. Distributed and Web Programming with dRuby. Ohmsha, 2005.
[13] http://edocs.bea.com/wls/docs100/config_wls/self_tuned.html Using Work Manager to Optimize Scheduled Work. BEA web site.
[14] http://datafarm.apgrid.org Gfarm file system web site.

Authors

Xiaohui Wei, ACM/IEEE member, PhD of Computer Software and Theory. He had worked with Platform Computing Inc. (Canada) from 1999 to 2004. Since 2003, he is a professor of the College of Computer Science and Technology in Jilin University, China. He is also the vice dean of the College of Computer Science and Technology and the director of the Grid Computing and Information Security Lab. His research interests include distributed system, grid computing, fault tolerance and information security.



Hongliang LI was born in March 1983. He is a Master student of College of Computer Science and Technology of Jilin University, P.R.C. He received his BS degree in Computer Science from Jilin University in 2006. His research interests include grid computing, high performance computing and fault tolerance in distributed system.

Shaocheng Xing was born in 1983. He received the BS degrees in Computer Science from Jilin University in 2006.He is currently a Master student of College of Computer Science and Technology of Jilin University, P.R.C. His research fields include grid computing and resource co-allocating.

JiLi was born in 1985. She received the BS degrees in Software Engineering from Jilin University in 2006. She is currently a Master student of College of Computer Science and Technology of Jilin University, P.R.C. Her research fields include grid computing and resource predicting.

Zane Zhenhua Hu was born in January 1958 in Changchun, China. He is a Principle Product Architect at Platform Computing Inc, Canada. He received Ph.D in Computer Science, Huazhong University of Science and Technology in 1990. His research interests include grid computing, high performance computing, compute/data/service grids, SOA and parallel programming, data distribution management, cloud computing.

Shutao Yuan was born in July 1962 in Anhui, China. He is the Director of Software development at Platform Computing Inc, Canada. He received Ph.D in Computer Science Department, Peking University in 1995. His research interests include grid computing, high performance computing in large heterogeneous distributed environment.