Framework for Peer-to-Peer Distributed Computing in a Heterogeneous, Decentralized Environment

Jerome Verbeke, Neelakanth Nadgir, Greg Ruetsch, Ilya Sharapov

Sun Microsystems, Inc., Palo Alto, CA 94303

Abstract. This paper presents a framework for large-scale computations for problems that feature coarse-grained parallelization. The components of this framework are based on Java, which allows for a wide variety of platforms and components, and peer-to-peer communication is provided through the JXTA protocols, which allow for a dynamic and decentralized organization of computational resources.

1 Introduction

Parallel computation has been an essential component of scientific computing for decades. Traditionally, when one thinks of parallelization, one often envisions fine-grained parallelization, which requires substantial inter-node communication utilizing protocols such as MPI[1, 2] or PVM[3]. Recently, however, there is an increasing demand for efficient mechanisms of carrying out computations which exhibit coarse-grained parallelism. Examples of this class of problems include throughput computations, where numerous similar but independent tasks are performed to solve a large problem, or any solution which relies on ensemble averages, where a simulation is run under a variety of initial conditions which are then combined to form the result. This paper presents a framework for these types of computations.

The idea of achieving parallelization through performing many independent tasks is not new. One realization of this method is the Seti@Home[4] project, where data from astronomical measurements is farmed out to many PCs for processing, and when completed returned to a centralized server and postprocessed. While this example does achieve coarse-grained parallelism, there are several issues which need to be addressed when building a generalized framework for distributed computing. In particular, the ability to run a variety of simulations, and using decentralized methods of job submission and result retrieval. Furthermore, other desirable aspects which are addressed in this framework include (1) a dynamic grid, where nodes are added and removed during the lifetime of the jobs, (2) redundancy, such that the dynamic nature of the grid does not affect the results, (3) organization of computational resources into groups, such that inter-node communications does not occur in a one-to-all or all-to-all mode, thereby limiting the scalability of the system, and (4) heterogeneity, where a wide variety of computational platforms are able to participate.

The framework in this paper utilized the JXTA[5] open peer-to-peer [6] communication protocols, which allow for the dynamic aspect (point 1 above) of the grid through peer discovery, in addition to the scalability aspect (point 3) through the use of peer groups. Heterogeneity (point 4) is achieved through Java's ability to run on various platforms. The decentralized aspect of the framework in addition to redundancy are discussed in the following sections.

Project JXTA was started at Sun Microsystems in 2001. JXTA defines a set of protocols that can be implemented by peers to communicate and collaborate with other peers implementing the JXTA protocols. It tries to standardize messaging systems, specifically peer-to-peer systems, by defining protocols, rather than implementations. Currently Java and C implementations of the JXTA protocols are available.

In JXTA, every peer is a identified by an ID, unique over time and space. Peer groups are user defined collections of entities (peers) who share a common interest, (in the least case, an interest for being a part of a peer group). Peer groups are also identified by unique IDs. Peers can belong to multiple peer groups, can discover other entities (peers and peer groups) dynamically and can also publish themselves so that other peers can discover them. Three kinds of communication are supported in JXTA. The first kind is called unicast pipe and is similar to UDP as it is unreliable. The second type is called secure pipe. The secure pipe creates a secure tunnel between the sender and the receiver, thus creating a secure, reliable transport. The third type is the broadcast pipe. When using the broadcast pipe, the message is broadcast to all the peers in the peer group.

2 Framework peer groups and roles

In creating a framework for distributed computation, one needs to address the issue of reliability and scalability at the outset of defining the architecture. Because we are restricting the type of end-user applications run on the framework to those that are embarrassingly parallel, a high degree of scalability is built into the system. The issue of efficiency then turns to the administration and coordination of tasks and resources. One advantage of building the framework utilizing the JXTA protocols is that the concept of peer groups can be leveraged. By utilizing peer groups as a fundamental building block of the framework, one is able to group resources according to functionality, in the process building redundancy and restricting communication messages to relevant peers.

The distributed computing framework contains the following peer groups: the monitor group, the worker group, the task dispatcher group, and the repository group. The monitor group is a top-level group which coordinates the overall activity of the framework, including handling requests for peers to join the framework and their subsequent assignment of the node to peer groups, and high-level aspects of the job submission process. The worker group is the peer group responsible for performing the computations of a particular job, while the task dispatcher group distributes individual tasks to workers. The repository group serves as a cache for code and data.

A single node can belong to several peer groups in the framework, and likewise there can be many instances of each peer group within the framework. These interconnectivity and redundancy features are critical in handling the dynamic nature of the environment, where resources are added and removed on a regular basis. In the following sections we discuss the interconnectivity of various peer groups in detail.

2.1 Code repository and the job submission process

There are two parts to the submission of a job: the code used by the worker nodes which is common for all tasks within the global job, and the data used by the code which generally varies for each task within a global job. The data segment of the job submission can range from being simple parameters which vary from task to task, to large data sets required for computations. As with other aspects of this framework, the storage of the two elements for a job are distributed throughout the network in a decentralized fashion. The management of these components falls under the repository peer group, and example of which is given in Figure 1.

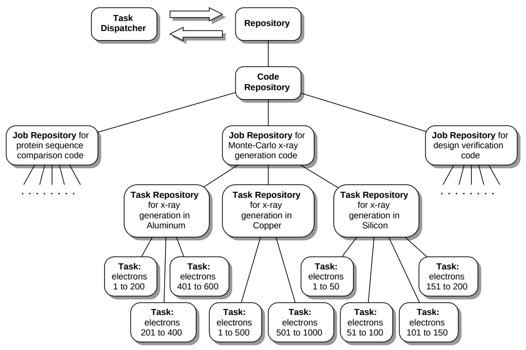
Task Dispatcher
Repository
Code Repository
Job Repository for protein sequence comparison code
Job Repository for Monte-Carlo x-ray generation code
Job Repository for design verification code
Task Repository for x-ray generation in Aluminum
Task Repository for x-ray generation in Copper
Task Repository for x-ray generation in Silicon
Task: electrons 1 to 200
Task: electrons 401 to 600
Task: electrons 201 to 400
Task: electrons 1 to 500
Task: electrons 501 to 1000
Task: electrons 1 to 50
Task: electrons 151 to 200
Task: electrons 51 to 100
Task: electrons 101 to 150

Fig. 1. Example of a Code Repository that contains three codes, each having its own job repository. The Monte-Carlo job repository has currently three jobs, each composed of a different number of tasks.

The interaction of the code repository group with the rest of the framework is through the task dispatcher group. Upon receiving the job submission, the task dispatcher polls the repository to determine the status of the code within the code repository. If the repository is current, then the code is retrieved, and otherwise uploaded and stored in the code repository. For each job, a job repository is created, which is a tree containing a repository for tasks within the job, which are submitted by the end-user.

2.2 Distribution of tasks amongst workers

In this section we discuss the interaction of the task dispatcher and the worker groups. Within each worker group there is one task dispatcher. Idle workers regularly poll the task dispatcher relaying information regarding resources available, including codes the worker has cached. Based on this information, the task dispatcher polls the repository for tasks to be performed on available codes, or for codes to be downloaded to the workers. Upon distribution of code and tasks, the worker performs the task and returns the result to the task dispatcher. It is important to note that the task dispatcher does not keep track of which workers are performing which tasks.

No handshaking is being performed between the worker and the task dispatcher. Both are working in such a manner that lost messages do not affect the final completion of a job. As such, a worker could become inaccessible during execution, which would not affect the overall completion of a job. The task dispatcher updates the repository with information about task completion, and redundant tasks are performed to account for node failure.

2.3 Result retrieval

Once a job has completed, that is, all the tasks in its task repository have completed, the tasks are ready to be sent back to the job submitter. However, the task dispatcher does not keep track of the job submitters. It is therefore up to the job submitter to initiate the result retrieval process.

The job submitter has a method that polls the task dispatcher to see whether the job that it submitted has completed. Each job consists of a task repository, which has a unique ID. This ID is sent to the job submitter when the task repository is created, and is used to request the results.

The task dispatcher relays this request to the repository which returns with the tasks if the job has completed. These results are sent back to the job submitter, and the job submitter retrieves the array of tasks and then postprocesses them.

3 Reliability

Reliability is an important requirement for distributed computing. Simulations can take days to complete and an outage can result in days of lost time. If the job is amenable to partitioning, it can benefit from the reliability features our framework implements. One of them is illustrated in Figure 2. If there was only a single task dispatcher and it was interrupted, all the results from the tasks executed by the workers who sent their results to that task dispatcher would be lost. Therefore, it is necessary to have redundant task dispatchers in task dispatcher peer groups. With two task dispatchers keeping each other up to date with the latest results they have received, the information is not lost if one of them incurs an outage.

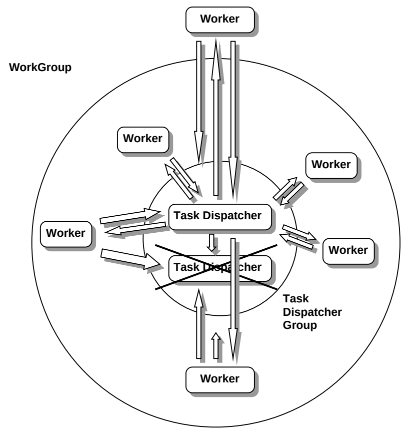

Fig. 2. Worker node assumes the role of task dispatcher when the latter is disrupted.

A new worker joining a work group does not contact a particular task dispatcher, but the task dispatcher peer group. A task dispatcher replies to the incoming message. The question of which task dispatcher replies is discussed in the following section on scalability. The worker then establishes communication with the task dispatcher. This communication establishment protocol is illustrated by the worker at the top of Figure 2. In this model, if a task dispatcher fails to respond to a worker, the worker backs out a level and contacts the task dispatcher peer group again. At this time, a different task dispatcher responds to his request. This protocol in case of a task dispatcher failure is illustrated by the worker at the bottom of Figure 2.

Task dispatchers in a peer group communicate by sending each other messages at regular time intervals. This regular message exchange will be referred as the task dispatcher heartbeat. When task dispatchers receive new results from a worker, they send them to the other task dispatcher to keep a redundant copy of these results. In order to reduce the communication between task dispatchers, the implementation of the model could be such that they update each other with newest results only during heartbeats.

A few comments should be made to the sequence of events that happen if a member of a given task dispatcher peer group is interrupted. As soon as the other task dispatcher in the same peer group realizes that his redundant colleague is missing, it will invite a worker requesting a task to execute the task dispatcher code in his peer group, transforming a regular worker into a task dispatcher. This role interchange is simple to implement, because both the worker and task dispatcher codes implement a common interface, making them equally schedulable in this model. This role interchange is illustrated in Figure 2 by the worker on the left side of the figure.

The number of task dispatchers in the task dispatcher peer group does not necessarily have to be limited to two. We could easily have triple or higher redundancy in the task dispatcher peer group. Also, because the communication protocols used do not limit us to working with peers in a small network, one can easily take advantage of the higher reliability offered by having redundant task dispatchers in various geographical locations. By having redundant task dispatchers in different states for instance, a power outage in one state would not result in any loss of information.

4 Scalability

As workers are added to a work group, the communication bandwidth between workers and task dispatchers may become a bottleneck. To prevent this, another role is introduced, the monitor. The main function of the monitor is to intercept requests from peers which do not belong to any peer group yet. Monitors act as middle men between work groups and joining peers. Job submitters who want to submit a job and workers who want to join a work group to work on a task will need to contact a monitor. Monitors free task dispatchers from direct communication with the outside world. Work groups communicate with their monitor and do not see the rest of the communication outside of the work group.

A monitor can have several work groups to monitor and can redirect requests from peers from the outside to any of the work groups it monitors. This redirection will depend on the workload of these subgroups. Just as we have task dispatcher peer groups, there are also monitor peer groups, with several monitors updating each other within a monitor peer group to provide redundancy.

With the addition of monitors, the way jobs are submitted to the framework is now slightly different. Job submitters make requests to the monitor peer group. Monitors within that peer group redirect these requests to a work group. The choice of this group depends on what code these work groups are already working on, on their workloads, etc. The work group replies directly to the job submitter, who establishes a working relationship with the work group.

The redirection by the top monitor group happens only once at the initial request by the job submitter to submit a job. Afterwards, messages are sent directly from the job submitter to the correct work group. A similar protocol is followed when a new worker wants to join the framework.

The role of the monitor is not only to redirect newcomers to the right work groups, but also to monitor the work groups, because it is up to the monitor to decide to which work group a job should be submitted. It will therefore keep track of work group loads, codes, and information about the loss of task dispatchers in a work group.

Monitors keep each other up to date with the status of the work groups under them with the monitor group heartbeat. Monitors can also request a worker to become a monitor in case of a monitor failure.

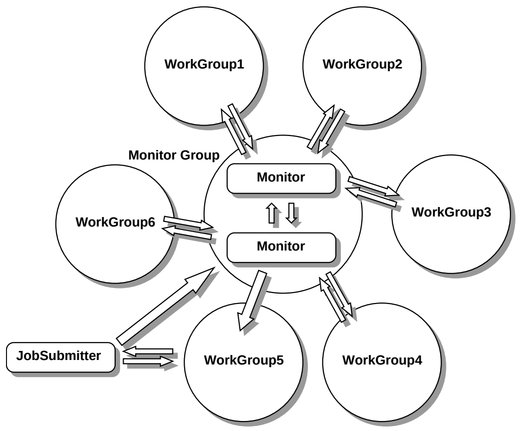

Fig. 3. Communications between monitor and other nodes.

As mentioned earlier, if too many peers are present in a work group, the communication bandwidth within that group may become a bottleneck. This would also happen if too many work groups are associated with the same monitor peer group. Therefore, the model also enables one to have a hierarchy of monitor peer groups, with each monitor peer group monitoring a combination of work groups and monitor groups. Whenever a monitor group becomes overloaded, it takes the decision of splitting off a separate monitor group, which takes some of the load off the original monitor group. The mechanism used to submit a job (job submitter) or to request a task (worker) from the computing grid is illustrated in Figure 3. The job submitter or worker contacts the top level monitor group. Based on the information passed with the message, one of the peers in the top monitor group decides which subgroup to hand on the request to, it forwards the request to the chosen subgroup. If this subgroup is a monitor group, the message is forwarded until it reaches a work group. Once the message is in a work group, a task dispatcher in the work group sends a reply to the job submitter/worker. This message contains the peer ID of the task dispatcher to contact, the ID of the task dispatcher peer group, as well as the peer group IDs of the intermediate peer groups involved in passing down the message. The job submitter/worker at this stage has a point of contact in a new work group. If it fails to contact the task dispatcher, it will successively contact the task dispatcher peer group, its parent, grand-parent, etc. until it succeeds in contacting someone in the chain. The last level of the hierarchy is the top level monitor group.

Because all the new peers joining the computing grid have to go through the top level monitor group, the communication at that level might become a bottleneck in the model. Numerous solutions exist to this problem. An easy one to implement is the following. When a new peer contacts the top-level monitor group, all the monitors within this peer group receive the message. Each monitor in the monitor peer group has a subset of requests to which it replies. These subsets do not overlap and put together compose the entire possible set of requests that exist. Based on a request feature, a single monitor takes the request of the new peer and redirects it to a subgroup.

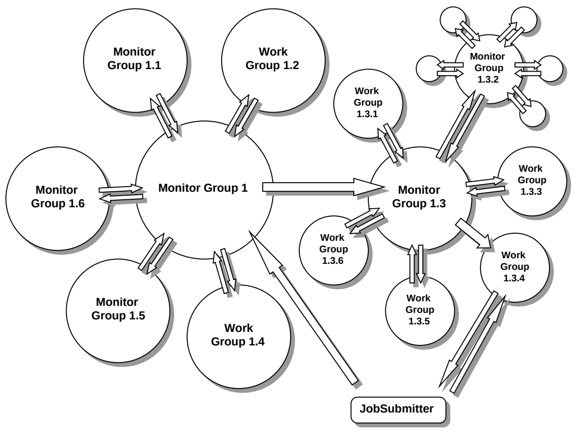

Fig. 4. Scalable network of work groups and associated monitor groups.

One should comment on the way monitors decide whether they will reply to a given request. This decision is made based on the request itself coming from the new peer. There is no need for communication between monitors to decide who will reply. For example, if you had two monitors in the monitor group, one monitor could reply to requests from peers having odd peer IDs, while the other monitor would reply to requests from peers having even peer IDs. The decision does not require any communication between the monitors and is therefore beneficial for our model. It reduces the communication needs and increases the bandwidth for other messages. One could also base this decision on the geographical proximity of the requestor to the monitor.

5 Example of usage of Peer-to-peer distributing computing framework

This section illustrates how to submit a job to the framework. The example used is trivial but it illustrates the features required for the framework to work.

In this example, we compute the sum of all the integers between 1 and 1000. The calculation can easily be split into several tasks. If we decide to divide the work into 10 parts. The first task would be the addition of all the integers between 1 and 100, the second task the addition of all the integers between 101 and 200, etc.

5.1 Programs for execution kernel and for job submission

To use the framework to do this calculation on several machines, we first need to write the core of the calculation in a code. In this case AddNumbers.java:

```
import java.io.Serializable;

public class AddNumbers implements Runnable, Serializable
{
    private int result, first, last;

    public AddNumbers(int first, int last)
    {
        result = 0;
        this.first = first;
        this.last = last;
    };

    public void run()
    {
        for (int i=first; i<=last; i++)
            result +=i;
    };

    public int getResult()
    {
        return result;
    };
};
```

Fig. 5. Sample computational program AddNumbers.java

The AddNumbers constructor is used to initialize the data, it takes arguments which must contain sufficient information for the code to run (in this case the first and last indexes of the numbers to be added). An AddNumbers class instance differentiates itself from another instance only through the way they are constructed. The run() method is the core of the calculation. Since it will be invoked remotely on a machine unknown to the job submitter, it should not contain anything requiring user interaction or display any graphics. Once the run() method has been executed, the AddNumbers object contains the result of the calculation (in this case, the sum of all the integers from first to last).

We will now go over some details in this code that are implementation specific. First the AddNumbers class needs to implement Serializable and Runnable. The first interface is necessary for the class instances to be sent using IO streams to the peers executing them. The second interface needs to be implemented for AddNumbers to be able to be passed to a RemoteThread object. RemoteThread described in the following section is used by the job submitter to submit a job to the computing grid. It is an implementation specific class and is similar to a regular java.lang.Thread, but it runs on a remote peer.

This job is split into ten tasks. Each task is identified by the first and last indexes of the integers to be added. These two integers are passed to the constructor of AddNumbers, and stored in two private variables. At this point, each task is well defined and ready to run.

Subsequently, a RemoteThread is created and two arguments are passed to it, the array of AddNumbers, and the directory containing the classes that are necessary to run the AddNumbers code. The bytecode transfer to remote peers uses byte arrays and XML messages. Files containing bytecode necessary to run a particular code are read by the RemoteThread class and put into XML messages. These messages are then sent to remote peers, which store the bytecode locally.

After the start() method is called, we check with the repository whether the AddNumbers code has already been submitted. If it has not been submitted, we send all the class files from the directory AddNumbers to the repository, which will create a job repository for this particular code. If it has already been submitted (for example by someone who had previously computed the sum of all the integers between 2000 and 4000), we do not create a new job repository for this code, but use the existing one. Afterwards, the array of individual tasks is sent to the repository. A task repository is created in for these tasks. Each of these task repositories has a unique ID that can be used by the job submitter to retrieve his results in the future.

Every 10 seconds, we check whether the tasks have completed. The join() method returns only when all the tasks have completed, after what it is possible to retrieve the results of the individual tasks using the getRunnable() method of RemoteThread.

The remove() method removes all tasks belonging to this job from the task repository. If this method is not called, the memory requirements of the repository increase in time.

The quit() methods should be called before exiting the application to quit cleanly.

5.2 RemoteThread Class

The RemoteThread class is similar to the java.lang.Thread class and can be used by the application writer to submit an application to be run in parallel to the framework. It has the following methods:

```
public class exampleApp
{
    public exampleApp()
    {
        AddNumbers [] tasks = prepareTasks(10);           // Split the job
into 10

        RemoteThread remoteTh = new RemoteThread(tasks, "AddNumbers");
        remoteTh.start();
        remoteTh.join(10000);
        Runnable [] run = (Runnable []) remoteTh.getRunnable();

        if (run != null)
            postprocess(run);

        removeTh.remove();
        remoteTh.quit();

        System.exit(0);
    }

    private AddNumbers [] prepareTasks(int numberOfJobs)
    {
        /* Create the instances of AddNumbers class for each task */
        AddNumbers [] tasks = new AddNumbers [numberOfJobs];
        for (int i=0; i<numberOfJobs; i++)
            tasks [i] = new AddNumbers(1+(i*1000)/numberOfJobs,
((i+1)*1000)/numberOfJobs);
        return tasks;
    }

    private void postprocess(Runnable [] run)
    {
        int sum = 0;
        for (int i = 0; i<run.length; i++)
            sum += ((AddNumbers) run [i]).getResult();
        System.out.println("sum = " + sum);
    }

    public static void main (String args[])
    {
        exampleApp app = new exampleApp();
    }
}
```

Fig. 6. Example of pre- and post-processing for the application submitted to a distributed environment.

– public RemoteThread(java.lang.Runnable [] tasks, String codeDir): An instance of the RemoteThread class is created, the arguments are a set of tasks implementing the Runnable interface, and the name of the directory where the classes containing the code to be passed to the other peers is located. When the start() method is called, the classes and then the tasks are sent to the task dispatcher.
– public void start(): This method is similar to the start() method of java.lang.Thread. It submits the code and the tasks to the framework.
– public void join(int timeInterval): This method is similar to the join() method in the java.lang.Thread class, except that a time interval has to be specified to determine how often the RemoteThread should check whether the job submitted has completed.
– public java.lang.Runnable[] getRunnable(): Once the job has completed, this method allows one to retrieve the results of the computations.
– void remove(): This method removes the java.lang.Runnable objects containing the results from the code repository.
– void quit(): This method cleans up the RemoteThread and should be used before quitting the user application.

6 Conclusion

The presented model has all the features required for scalable, robust, efficient and dynamic grid computing. The peer-to-peer design of the grid limits communication to small peer groups that really require it. This enables the computing grid to scale to a very large numbers of peers. As pointed out earlier, the fact that task dispatchers can be redundant across large geographical locations, node outages in a single location will not affect the overall computational process. It also takes advantage of all the nodes willing to join the grid computing effort and is very efficient by this dynamicity. The RemoteThread class provides a set of APIs to harness the power of the framework by the application developers. One of the interesting features that has not been highlighted but which is a consequence of the features of this distributed computing model is the non-locality of the grid in the network. The grid could be composed of a few peers located in California today, and migrate to Japan tomorrow as more peers join the computing grid. This migration is directly dictated by the number of peer nodes joining the framework. Adding features to existing distributed computing models, this model pushes the boundary of grid computing to the global network.

References

1. Gropp, W., Lusk, E. & Skjellum, A. Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge, 1999.
2. Gropp, W., Lusk, E. & Thakur, R. Using MPI-2: Advanced Features of the Message Passing Interface. MIT Press, Cambridge, 2000.

3. Geist, A., Geguelin, A., Dongarra, J., Jiang, W., Manchek, R. & Sunderam, V. PVM: Parallel Virtual Machine MIT Press, Cambridge, 1995.
4. SETI@home, The Search for Extraterrestrial Intelligence (SETI), http://setiathome.ssl.berkeley.edu/
5. Project JXTA, http://www.jxta.org/
6. A. Oram, Editor. Peer-to-Peer, Harnessing the Power of Disruptive Technologies. O'Reilly & Associates, 2001.