GLOBAL 3D SEISMIC TOMOGRAPHY USING MULTI-CORE DISTRUBUTED MEMORY PARALLEL COMPUTERS

James R. Hipp, Marcus C. Chang, Mark A. Gonzales, Benjamin J. Lawry,
Andre V. Encarnacao, and Glenn T. Barker

Sandia National Laboratories

Sponsored by the National Nuclear Security Administration

Award No. DE-AC04-94AL85000/SL09-3D-Earth-NDD02

ABSTRACT

The rapid growth of multi-core computing hardware has made it possible for scientific researchers to run complex, computationally intensive software on affordable, in-house commodity hardware. Multi-core CPU's (Central Processing Unit) and GPU's (Graphics Processing Unit) are now commonplace in desktops and servers. Researchers today have access to extremely powerful hardware that enables the execution of software that could previously only be run on expensive, massively-parallel systems. It is no longer cost-prohibitive for an institution to build a parallel computing cluster consisting of commodity multi-core servers. With each server offering 24 or more processing cores, researchers can easily have access to hundreds of parallel processing threads to facilitate complex calculations. One research area where a distributed multi-core computing system has proved to be critical is in the development of global 3D earth models. Traditionally, computational limitations have forced certain assumptions and shortcuts in the calculation of tomographic models; however, with the recent rapid growth in computational hardware including faster CPU's, increased RAM, and the development of multi-core computers, it is now possible to perform seismic tomography and location using distributed parallel algorithms running on commodity hardware, thereby eliminating the need for these shortcuts. In this paper we will describe the multi-core distributed computing system that we have designed to support our in-house geophysical algorithm research including, in particular, the task-based parallel framework used for calculational processing along with the resource management system developed for sharing computing resources among multiple simultaneous developers.

OBJECTIVE

The overall objective for this project is to improve the pace at which computationally intensive Ground-based Nuclear Explosion Monitoring Research & Development (GNEMRD) research can be performed by developing scalable software that will enable GNEMRD researchers to easily assemble a platform-portable, cost-effective, and practical distributed parallel system comprised of any number of modern commodity, multi-core computers. The availability of such a system to the Sandia National Laboratories (SNL) GNEMRD research group has dramatically accelerated the pace at which our computationally intensive 3D tomographic research is conducted, and we believe it can prove equally valuable to other GNEMRD researchers. Our design emphasis targets large scientific computing solutions, specifically geophysics problems, which commonly require a large amount of in-core memory as well as significant file and database resources. Additionally, we require an environment that supports many users / developers operating in a simultaneous fashion. Our initial testing of this system has been restricted to the SNL GNEMRD LAN, but our eventual goal is to develop a software package that can be easily deployed at other sites where GNEMRD research products are developed to improve U.S. monitoring capability.

RESEARCH ACCOMPLISHED

Background

Realistic and practical Earth structure tomography problems are generally solved using either iterative matrix solutions or with stochastic methods such as Monte Carlo (MC) or Markov Chain Monte Carlo (MCMC) techniques (Tarantola, 2005). Event location, while less strenuous, utilizes similar numerical frameworks (e.g., Levenberg-Marquardt). These types of solutions are inherently large and generally involve hundreds of thousands to many millions of ray prediction calculations when solving for realistic three dimensional Earth wave velocity structure and event positions. Additionally, they may involve many support calculations for determining optimum control parameter settings (e.g., damping, regularization), ray similarity, event clustering or for performing a statistical assessment of how well the resulting model approximates the data used to develop the model.

In the past, much of this type of research was executed on simple sequential processors using a single desktop workstation to perform the calculations. With this type of minimal processing power, typical tomography problems were by necessity defined with low spatial resolution or severely restricted in domain extent (Soldati, 2006). In the last 15 years or so some researchers have had access to small shared memory systems or clusters for performing this type of research, but these were generally expensive costing several hundred thousand dollars and not widely available to the typical researcher. These types of older cluster machines generally possessed 16 to 64 processors with an accompanying amount of memory that was usually less than 100 MB. The significant computational resources required for performing high-fidelity large seismic tomography investigations could only be found at major research institutions that could afford multi-million dollar massively parallel machines. For most researchers the only option was to simplify their approach if they wanted to produce models.

In the meantime microprocessor manufacturers have achieved higher performance rates by manipulating the processors' fidelity at handling Instruction Level Parallelism (ILP). Eventually, large power consumption and heat dissipation problems drove these manufactures to investigate Thread Level Parallelism (TLP) where performance is achieved using many different cores (CPU assemblies) contained on a single chip where clock rates are controlled to effectively manage power consumption and heat dissipation issues (Buttari, 2008). This latest direction of using multi-core processors has defined a new paradigm in affordable multi-processor parallel computation available to almost any researcher. Without exploiting this new multi-core paradigm, even the most advanced single-threaded software algorithms can only run on a single core, leaving the remaining cores idle, and thus wasting a significant amount of available processor resources. On the other hand, algorithms designed to run on multiple cores concurrently can theoretically approach an n-fold increase in performance given a multi-core processor offering n cores.

Today we are sitting on the threshold of harnessing computational resources in a manner not seen in the past. The multi-core CPU and, to a similar extent, the GPU turned numerical processor, are paving the way for a new paradigm in scientific computing. To fully realize this potential, researchers need methodologies and frameworks to integrate disparate platforms and systems into a single common computing environment.

In this report we examine such a system that we have assembled to support our in-house 3D geophysical algorithms research. In particular, we will describe a multi-core parallel processing framework that includes a processing task distribution system and a resource management system. The task distribution system handles computational work unit assignment for individual clients (applications) while the resource management system allocates physical processing nodes, memory, and IO capabilities to each application. These systems provide a crucial capability that enables many researchers and developers to simultaneously take advantage of parallel computation using an easily configurable set of disparate resources.

Before beginning with the overall framework discussion, however, we will first provide some basic background on the main types of modern multi-core hardware and the two primary types of processing parallelism (i.e., task- and data-parallelism). Understanding these terms and definitions is critical towards understanding how the entire system can provide the necessary set of capabilities to ensure that all types of numerically intensive solutions can be solved in an efficient manner.

Definition (Multi-core CPU versus Multi-core GPU)

Multi-core or "many-core" computing is a form of parallel processing that takes advantage of the presence of multiple processing cores within a single chip. For the purposes of our discussion, we will be focusing on two types of multi-core hardware: multi-core CPUs and multi-core GPUs. A single machine can contain multiple CPUs and GPUs, so the number of processing cores available in a given machine can be significant (number of processors * number of cores per processor).

The CPU is the traditional computer processor that is used in everything from laptops to desktops to servers. Today, it is nearly impossible to purchase a computer that does not include at least one multi-core processor. For desktops, quad-core (or higher) processors are becoming the norm. However, most applications do not take advantage of the presence of multiple CPU cores; instead, most off the shelf software today runs sequentially on a single core.

The GPU has traditionally been viewed as part of the video card, used by computers to render graphics. Historically, the drive for advanced GPUs came from the video gaming market, where realistic 3D visualizations have become the standard. This forced the development of GPU processors containing large numbers of pipelined cores for performing various graphical operations such as coordinate transformation, polygon rendering, texture mapping, and shading. Recently, however, the General Purpose GPU (GPGPU) concept has emerged where the massive floating point power provided by GPUs is being harnessed to perform scientific mathematical computations which can yield several orders of magnitude increase in performance over standard CPUs. In 2009, Nvidia offered a Tesla GPU with 240 cores per video card and up to four cards per machine at a cost of $1500 per card.

The determination of the best type of multi-core hardware to use will typically depend on the problem that is being solved. Some problems have a very small ratio of floating point operations per memory access, while others have large ratios. Problems spanning small to large ratios execute better on certain multi-core platforms. In the next section we will describe the two primary classes of parallelism and the types of multi-core platforms best suited for solving problems belonging to either of those categories.

Task-Parallelism and Data-Parallelism

Most problems can be grouped into one of two parallel classes distinquished by the frequency with which they access hardware memory resources. The first, task parallelism, is the simultaneous execution of a specific computation on many CPUs/GPUs using the same or different data. The work done in the computation is executed in a separate thread and in the simplest case there is no need for communication between threads. Task parallelism is also known as coarse-grained parallelism. Coarse-grained parallelism implies that each separate task requires very little communication with other tasks. It is known as "embarrassingly" parallel if the tasks require no communication with each other. We use task-parallelism to solve ray bending solutions through a 3D Earth model (Ballard et al., 2008; 2009). Each ray bending calculation is entirely independent from all other ray bending calculations. That is, you do not need any information from a ray to be able to solve for another ray.

We also use task-parallelism to solve many Least Squares QR (LSQR) solutions simultaneously in Earth model tomography. Here we are interested in determining an optimal regularization parameter to avoid over- or under-smoothing while calculating a new Earth model (Aster, 2005). Again each LSQR computation is independent and

does not require information from any other LSQR computation. We also utilize an out-of-core blocked Cholesky solver which can solve an Earth tomography problem setup as a rectangular matrix,

$$Gs = r \qquad (1)$$

where G is the tomography path length weight matrix and s is the incremental slowness perturbation for which we are solving. The travel time residual is the column vector r. Multiplying each side of the matrix by G^T forms a positive definite system that can be solved using the out-of-core blocked Cholesky solver. Each Cholesky block is updated with other blocks and scaled with its associated diagonal block on separated processors. Although some communication is required to avoid scaling a block before it is completely updated, it remains largely a task-parallel solution technique.

Other examples of our current research that use task-parallelism include determining ray-similarity and performing ray clustering from the similarity results (Young et al., 2009). Ray clustering is used to reduce the total number of rays, many of which are slight perturbations of other rays, for performing Earth tomography calculations. Having many essentially equal rays in a tomography calculation can overweight the solution in the neighborhood of the rays' path, often leading to very locally biased results. Reducing a family of nearly identical rays to a single representative helps to alleviate the biasing. Determining ray similarity is a comparison evaluation of one ray with all other rays in a dataset. Since each ray must be compared to all other rays, the total computation involves $N(N+1)/2$ comparisons where N is the number of rays in the dataset. In practice a significant number of pairs can be excluded by applying a few obvious constraints (e.g., only comparing rays to the same station), but the total number of pairs can still be very large. We can solve this problem in a parallel fashion by dividing up the rays into a number of groups such that each group is compared with other groups on a set of multi-core processors. Once the ray similarity has been determined, a clustering algorithm is used to form the ray representatives. Here again we can utilize multi-processors to form clusters from disjoint groups of rays that are only linked to a subset of other rays by way of the similarity measure. Each subset can be processed on an individual processor simultaneously thereby speeding up the entire calculation.

Residing at the other extreme of parallel problem classes are those that present what is known as data parallelism. Data-parallelism is the simultaneous execution of the same computation across the same data, and is often referred to as fine-grained parallelism. It is also known as loop-level parallelism as it focuses on the inherent parallelism in a program loop. The work done by a data-parallel application executes in separate threads and usually shares information between threads. An example of data parallelism is matrix inversion and matrix decomposition algorithms, where each processing thread is working on a subset of the matrix (performing the exact same operation on each piece of data). We have several data-parallel problems within our research umbrella that require solution in a parallel manner. These include individual LSQR calculations and Cholesky decompositions. In the case of the LSQR solution we are not talking about solving several independent LSQR calculations simultaneously, as we described above, but instead we mean solving a single large LSQR problem in parallel (Liu, 2006). In this mode the entire sparse matrix (G in the tomography problem above) is dispersed among many processors (typically GPUs) and the solution is performed on each piece of the matrix simultaneously.

The distinction between task- and data-parallelism is important as all geophysics problems, and scientific problems in general, will exhibit one or the other or some combination of both types of parallelism. Our design ensures that both types of parallelism are supported including support for task-parallel applications which utilize multi-core CPUs and data-parallel applications which use predominantly GPUs.

Java Parallel Processing Framework

Given the basic definitions of multi-core CPUs and GPUs and the application divisions between task-parallel and data-parallel algorithms, we now consider the primary components in our design. The first component is the task distribution and processing system. We have integrated a 3rd party application called the Java Parallel Processing Framework (JPPF) into our design to manage the requirements of this component (Cohen, 2009). The JPPF provides a flexible and innovative way to take advantage of modern multi-core hardware. This Java-based, platform-independent framework allows research applications to submit units of work, known as JPPF tasks, to be processed on a distributed network of computers. JPPF is made up of a JPPF *task driver* and a set of *processing nodes*. Applications use JPPF by creating JPPF *tasks* that are sent to the JPPF task driver for execution via a JPPF

client that resides in the primary client application's task interface. The JPPF task driver maintains a queue of JPPF tasks and sends JPPF tasks out to the various processing nodes as they become available (see Figure 1).

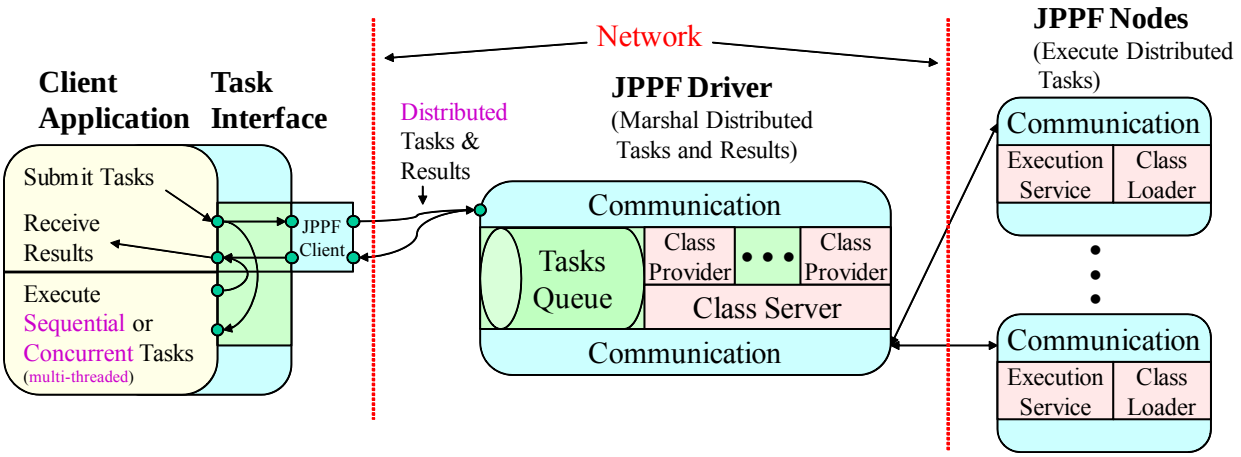

Figure 1 The JPPF computation framework and task distribution system.

We have amended the front end of the task-parallel process in the client application to include the ability to drive a solution in "sequential" mode on a single node or in "concurrent" mode (multi-threaded) using all multi-cores available on a single machine. This capability does not require JPPF but allows a user to configure their application to run on a single platform without the need of a system of distributed nodes, should they not be available. This convenience gives the user the ability to quickly switch computing modes to meet the current need, e.g., from debug mode where running on a single processor might be useful, to full "distributed" mode using all machine resources across the network when producing a real tomographic model.

All the processing nodes of the system (which can consist of servers, desktops, and even laptops as illustrated in Figure 2 below) execute the JPPF tasks in parallel, enabling algorithms to scale linearly with the number of processing nodes available. In general, one processing node is launched per CPU processor core; for example, a server with 16 cores would have 16 processing nodes. This distribution of JPPF tasks to CPU processing nodes via a JPPF driver is how JPPF achieves task-based parallelism.

In addition, the framework will also allow a processing node to be setup as a "data parallel node" to support data parallelism. In this case a data-parallel node is configured to solve a portion of the total problem by operating on a predefined subset of the data. Using data-parallelism with JPPF requires that each node be configured with its partial data allotment after which each node executes the process and sends the subsequent results back to the driver, and eventually, back to the client. The client ensures that all results are gathered and combined to finish the process.

Precise control over JPPF in a data-parallel mode requires that a client can dictate which processor executes with what data. In a task-parallel mode this type of control is not necessary as any processor can execute with any data. Marshalling tasks in JPPF with specific node execution control during the process allows JPPF to achieve data parallelism. Possessing the ability to operate with and without node control means JPPF supports both task- and data-parallelism in a hybrid manner.

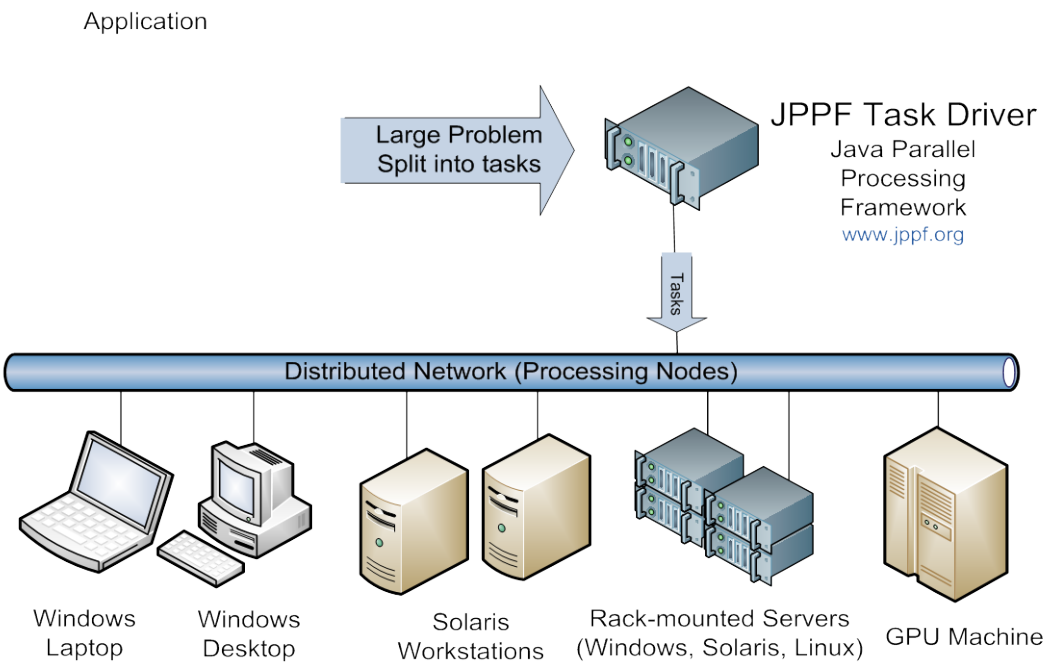

Figure 2 The Heterogeneous JPPF Distributed Network of Processing Nodes.

Node Resource Management (NRM) Framework

While the JPPF is charged with actually executing a set of tasks for one or more applications, it has no real ability to configure resources, start them, and make them available for use in the parallel environment. This is the job of the Node Resource Management (NRM) framework (see Figure 3). The NRM framework is an independent management system that works in tandem with JPPF and facilitates access to distributed computing environments. One of the main goals of the NRM framework is to make it easier to manage and share JPPF processing nodes in the distributed system.

The NRM system employs a client-server architecture and consists of three main software components: NRM server, NRM client, and NRM host. The NRM *server* is the primary component of the system and is responsible for providing applications access to the distributed computing environment and underlying JPPF processing nodes. NRM treats each system in the distributed computing environment as a resource and automatically shares these resources fairly amongst multiple applications simultaneously. The NRM *client* is responsible for connecting applications to the services and resources provided by the NRM Server. The NRM client provides an interface API to the client providing control and status information services. Lastly, the NRM *host* operates on machines that contribute JPPF nodes to NRM and is responsible for starting/stopping JPPF nodes on those machines, as governed by the NRM server.

NRM builds upon many of the benefits provided by JPPF. The primary benefits are its user friendliness, ability to effectively and fairly share resources, and robustness. Each of these benefits is discussed in more detail below.

NRM provides developers with an easy-to-use Application Programming Interface (API) that allows access to the distributed computing environment. All services are made available to the developer as part of the NRM client via singular function calls, minimizing the amount of knowledge required to use the system. Because NRM is based on a client-server architecture, any user/developer can utilize the system from any machine on the network. Additionally, the NRM client provides the user with an embeddable GUI frontend with many features. These include the ability to start/stop JPPF drivers and associated applications, request/release JPPF processing nodes, along with various monitoring capabilities.

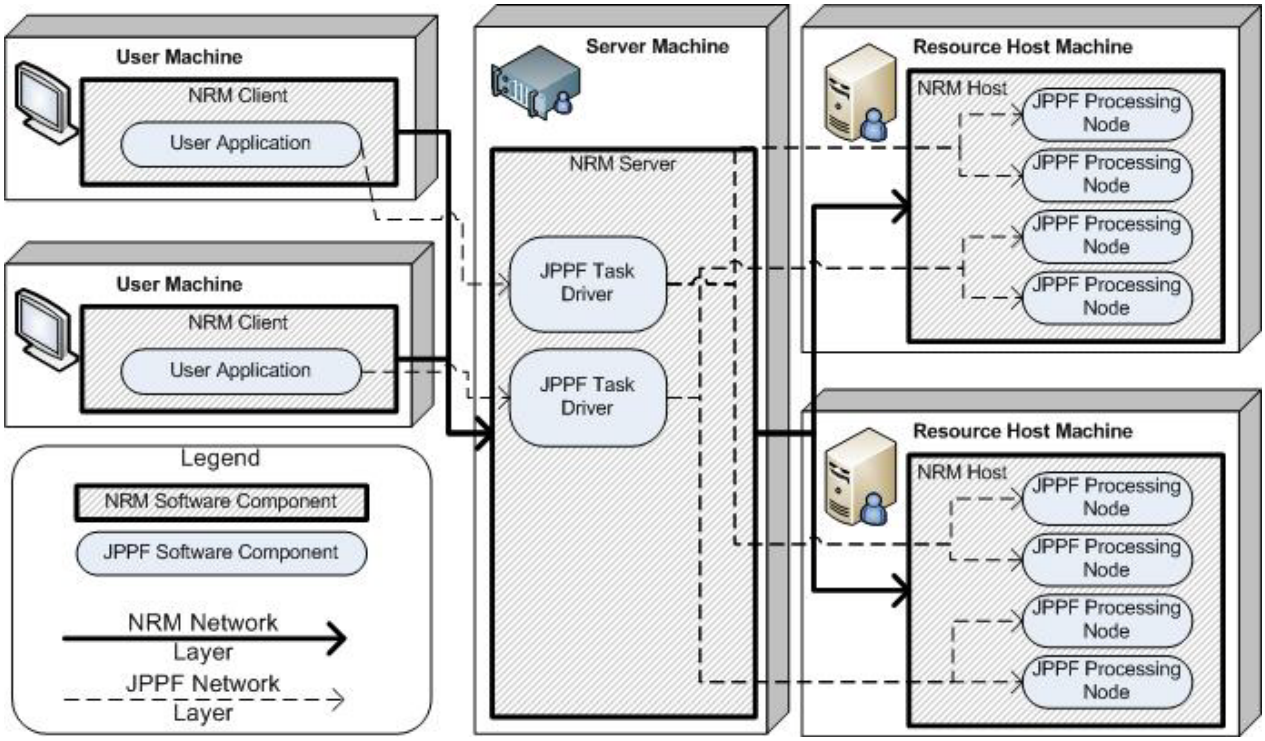

Figure 3 The Node Resource Management Framework Architecture Overview.

Most capabilities come in the form of automated periodic statistics updates. The NRM client automatically monitors every application running on the framework and displays information on CPU usage, number of nodes in use, total available resources, and more. The NRM client also tracks statistics for each running application, making it possible to determine the efficiency of distributed applications. For easier debugging, the NRM client is equipped with extensive logging capabilities and has functionality to display error notifications generated by the various JPPF components used in the system.

NRM is designed to simplify the overhead involved with setting up a distributed computing network as much as possible. Written in the latest release of Java (6.0), NRM is fully platform independent, allowing developers to run their applications simultaneously in mixed-environment networks including Windows, Linux, Unix, and Apple operating systems, without having to rewrite code for any specific platform. NRM is also architecture independent and supports computation on any combination of 32- and 64- bit architectures. In addition to the portability features mentioned above, NRM provides automated code distribution – whenever a developer wishes to run code on the system, the NRM framework automatically distributes compiled libraries to each machine on the network, simplifying the deployment process.

Besides user friendliness, NRM also has the ability to effectively and fairly share computing resources among multiple users and their applications in a transparent manner. More specifically, NRM implements a node balancing algorithm that provides all connected applications access to a shared pool of processing nodes. Given the nature of JPPF, each node can only be assigned to one application at a given time. For each application using the NRM, the node balancing algorithm maintains a weighted node count metric based on the application's priority and the processing power of the nodes that application is currently using. The basic idea behind the algorithm is to keep NRM 'balanced' by ensuring that these application-specific metrics remain roughly equal for all connected applications, at all times. Over the course of its operation, there are certain events that can place the algorithm in an un-balanced state, thus requiring nodes to be dynamically reassigned from one application to another in order to rebalance the system. The most common events are when a new application joins the system and wants to acquire its own processing nodes, or when a new host joins the system and additional processing nodes are available for use. In

these and other cases, node reassignments can occur. In order to minimize any disruptions to the applications, reassignments only take place after the completion of one or more JPPF tasks. This allows the node balancing algorithm to remain transparent to JPPF and ultimately the end-user of the application. Also, prior to making node reassignments (as well as initial assignments), the node balancing algorithm ensures that an application is only given nodes that meet certain memory and minimum processor performance constraints that it may have specified. This prevents processing nodes from being assigned to applications that are incapable of making use of it.

Resource management is handled directly by the NRM server. NRM hosts are added to a database that is read by the NRM to identify which hosts belong to the system. These NRM hosts connect with the server and specify the number of processors that will be made available for use. Furthermore, NRM facilitates the sharing of resources by giving end-users the ability to contribute their own computing resources to the NRM system. Using a policy-based and database-driven approach, users can specify the number of processors to contribute at different times of the day. For example, a user may wish to contribute only a subset of the processors on their desktop computer while present in their office and performing work, but all of their processors when away from the office. Additionally, users are allowed to override their policy at any time through the NRM client.

In addition to user friendliness and resource sharing NRM is also a robust framework. One of the major pitfalls of distributed computing is that as the system grows larger the likelihood that any one component of the system will fail increases. NRM has been designed with this problem in mind, making sure to remain in a stable state and minimize the effect on applications using the system. The most common scenarios are node and host machine failures. In the event that a single node fails, the NRM server automatically detects the failure and restarts the node. If the node cannot be restarted, or it is determined that a configuration issue caused the failure, a notification is sent to the user immediately. In the event that an entire host machine fails (during task execution), the NRM server again automatically detects the failure. If the failure causes an unfair imbalance in processing nodes between connected applications, node reassignments will occur to rebalance the system. In both failure cases, the directly affected applications will continue to function normally since the underlying JPPF components also detect these failures and are able to re-schedule unfinished tasks as necessary.

NRM also ensures that actions taken by one application do not affect other applications using the system. For example, when an application exits the system, either voluntarily after completing all of its processing tasks or as a result of a failure, other applications connected to the system are not negatively impacted. In fact, the NRM server detects such an event and ensures that the resources released by the terminated application are distributed fairly amongst the remaining applications. Similarly, when an additional application joins the system, the NRM server again performs node balancing to redistribute the computing resources in a fair manner.

CONCLUSIONS AND RECOMMENDATIONS

We have described a multi-core parallel processing framework that can be run on affordable commodity hardware to dramatically decrease the time needed to solve large complex geophysical problems, such as 3D seismic tomography. Our framework is comprised of two major sub-components: a processing task distribution system and a resource management system. The task distribution system, JPPF, handles computational work unit assignment for individual clients (applications) in both a task- and data-parallel manner. This dual hybrid system allows us to solve all types of geophysically defined problems with great efficiency.

The resource management system allocates physical processing nodes, memory, and IO capabilities to each application. It provides an easy to use API for control and abstracts the complexity of processing node configuration and assignment from the client. Additionally, it provides a fair and automatic system for assigning nodes to client applications that works seamlessly with the JPPF task processing system. These capabilities ensure that the client need only be concerned with the scientific problem of interest and not the particulars of hardware and software interaction required to perform a parallel solution. These abilities also imply that the system is made to order for everything from simple processing tasks to multi-pipelined operational systems. A system such as this can support multiple independent, but simultaneously functioning, research, development, and operational pipelines.

Our system is easily scalable from a single node with a single processor to many nodes executing multi-core and GPU processors over a widely distributed network. Using Java as the computing language guarantees platform portability and removes communication issues associated with disparate hardware platforms (Unix, Linux,

Windows, MacOS). The hardware used in our research consists of standard over-the-counter laptops, desktops, and servers. All of these points imply that the system is simply scalable and expandable and will compliment any research budget.

We have successfully deployed this system on the SNL GNEMRD LAN and demonstrated that it allows us to generate tomographic models 2 orders of magnitude faster than is possible with standard, single-processor serial processing. Our system is general and we believe it can be applied to aid research involving any other computationally intensive algorithm that is readily parallelizable. Recently we have started using it for waveform correlation event detection with similarly impressive performance improvements. We continue to work on improving the robustness and ease of use of the system toward the eventual goal of a package that can be easily installed and configured on the LANS of other GNEMRD researcher organizations to improve their computational capability and thereby accelerate the overall pace at which computationally intensive GNEMRD research can be conducted.

REFERENCES

Aster, Richard C., B. Borchers, and C. H. Thurber (2005). *Parameter Estimation and Inverse Problems*, Elsevier Academic Press, ISBN 0-12-065604-3.

Ballard, S., J. Hipp, and C. Young (2008). Robust, extensible representation of complex earth models for use in seismological software systems, in *Proceedings of the 30th Monitoring Research Review: Ground-Based Nuclear Explosion Monitoring Technologies*, LA-UR-08-05261, Vol. 1, pp. 347–355.

Ballard, S., J. Hipp, and C. J. Young (2009). Efficient and accurate calculation of ray theory seismic travel time through variable resolution 3D earth models, *Seism. Res. Lett*. (in press).

Buttari, A., J. Langou, J. Kurzak, and J. Dongarra (2008). A class of parallel tiled linear algebra algorithms for multicore architectures, arXiv:0709.1272v3 [cs.MS].

Cohen, L. (2009). Java Parallel Processing Framework, http://www.jppf.org/.

Liu, Jing-Song., Liu, Fu-Tian, Liu, Jun, Hao, Tian-Yao (2006). Parallel LSQR Algorithms Used In Seismic Tomography; *Chinese Journal of Geophysics* 49: N 2.

Soldati, G., L. Bochi, and A. Piersanti (2006). Global Seismic Tomography and Modern Parallel Computers, *Annals of Geophysics*, 49: N 4/5.

Tarantola, Albert (2005). Inverse problem theory and methods for model parameter estimation, SIAM, ISBN-13 978-0-898715-72-9.

Young, C., S. Ballard, J. Hipp, M. Chang, J. Lewis, C. Rowe, and M. Begnaud (2009). A global 3D *P*-velocity model of the Earth's crust and mantle for improved event location, these Proceedings.