AN EXTENDED MASTER WORKER MODEL FOR A DESKTOP GRID COMPUTING PLATFORM (QADPZ)

Monica Vlădoiu
Departament of Informatics, PG University of Ploieşti Romania, Bd. Bucureşti, Nr. 39, Ploieşti, Romania

Zoran Constantinescu
ZealSoft Ltd., Str. Tg. Neamţ, Nr. 60, Bucureşti, Romania

Keywords: Grid computing, desktop grid computing, distributed and parallel computing, master worker paradigm.

Abstract: In this paper we first present briefly QADPZ, an open source platform for heterogeneous desktop grid computing, which enables users from a local network (organization-wide) or Internet (volunteer computing) to share their resources. Users of the system can submit compute-intensive applications to the system, which are then automatically scheduled for execution. The scheduling is made based on the hardware and software requirements of the application. Users can later monitor and control the execution of the applications. Each application consists of one or more tasks. Applications can be independent, when the composing tasks do not require any interaction, or parallel, when the tasks communicate with each other during the computation. QADPZ uses a master worker-model that is improved with some refined capabilities: push of work units, pipelining, sending more work-units at a time, adaptive number of workers, adaptive timeout interval for work units, and use of multithreading, to be presented further in this paper. These improvements are meant to increase the performance and efficiency of such applications.

1 INTRODUCTION

Usually, complex computational and visualization algorithms require large amounts of computational power. The computing power of a single desktop computer is insufficient for running such complex algorithms, and, usually, large parallel super-computers or dedicated clusters are used for this job. However, very high initial investments and maintenance costs limit their availability. A more convenient solution is based on the use of non-dedicated desktop PCs in a Desktop Grid (DG) computing environment. Harnessing idle CPU cycles, storage space and other resources of networked computers to work together on a particularly computational intensive application, and increasing power and communication bandwidth of desktop computers provides for this solution.

In a DG system, the execution of an application is orchestrated by a central scheduler node, which distributes the tasks amongst the worker nodes and awaits workers' results. It is important to note that an application only finishes when all tasks have been completed. The attractiveness of exploiting DGs is further reinforced by the fact that costs are highly distributed: every volunteer supports her resources (hardware, power costs and internet connections) while the benefited entity provides management infrastructures, namely network bandwidth, servers and management services, receiving in exchange a massive and otherwise unaffordable computing power. The usefulness of DG computing is not limited to major high throughput public computing projects. Many institutions, ranging from academics to enterprises, hold vast number of desktop machines and could benefit from exploiting the idle cycles of their local machines.

We present briefly here QADPZ, an open source platform for heterogeneous desktop grid computing, which enables users from a local network (enterprise DG) or Internet (volunteer computing) to share their resources. The system allows a centralized management and use of the computational power of idle computers from a network of desktop computers. Users of the system can submit compute-intensive applications to the system, which are then

Vlădoiu M. and Constantinescu Z. (2008).
AN EXTENDED MASTER WORKER MODEL FOR A DESKTOP GRID COMPUTING PLATFORM (QADPZ).
In *Proceedings of the Third International Conference on Software and Data Technologies - PL/DPS/KE*, pages 169-174
DOI: 10.5220/0001882301690174

automatically scheduled for execution. The scheduling is made based on the application's hardware and software requirements. Users can later monitor and control the execution of the applications, each of them consisting of one or more tasks. Applications can be independent, when the composing tasks do not require any interaction, or parallel, when the tasks communicate with each other during the computation.

QADPZ has pioneered autonomic computing for desktop grids and presents specific self-management features: self-knowledge, self-configuration, self-optimization and self-healing (Constantinescu, 2003). It is worth to mention that to the present the QADPZ has over a thousand users who have download it, and many of them use it for their daily tasks (SOURCEFORGE, 2008).

2 MASTER WORKER PARADIGM

The QADPZ framework is based on an extended version of the master-worker paradigm. Some of the disadvantages in using this model in a heterogeneous and dynamic environment are briefly described here. The traditional master-worker has the worker nodes downloading small tasks from a central master node, executing them, and then sending back the results to the master. The master-worker computing paradigm is built on the observation that many computational problems can be broken into smaller pieces that can be computed by one or more processes in parallel. That is, the computations are fairly simple consisting of a loop over a common, usually compute-intensive, region of code. The size of this loop is usually considered to be long.

In this model, a number of worker processes are available, each can perform any one of the steps in a particular computation. The computation is divided into a set of mutually independent *work units* by a master node. Worker nodes then execute these work units in parallel. A worker repeatedly gets a work unit from its master, carries it out and sends back the result. The master keeps a record of all the work units of the computation it is designed to perform. As each work unit is completed by one of the workers, the master records the result. Finally, when all the work units have been completed, the master produces the complete result. The program works in the same way irrespective of the number of workers available - the master just gives out a new work unit to any worker who has completed the previous one.

Whereas the master worker model is easily programmed to run on a single parallel platform, running such a model for a single application across distributed machines presents interesting challenges. On a parallel platform, the processors are always considered identical in performance. In a distributed environment, and especially in a heterogeneous one, processors usually have different types and performance. This raises the problem of load balancing of work-units between the workers in such a way to minimize the total computing time of the application. The ideal application is coarse-grain and embarrassingly parallel. Granularity is defined as the computation-to-communication ration, with coarse grain applications involving small communication time compared to computation time, and fine grained application requiring much more time for communication than computation.

Coarse-grain applications are ideal for DG computing because most DG systems employ commodity network links, which have limited bandwidth and high latencies. Embarrassingly parallel applications are those problems that easily decompose into a collection of completely independent tasks. Examples of such scientific problems are: genetic and evolutionary algorithms, Monte Carlo simulations, distributed web crawling, image processing, image rendering.

In a heterogeneous environment, scheduling, which includes both problem decomposition and work-unit distribution (placement to workers), has a dramatic effect on the program's performance. An inappropriate decision regarding decomposition or distribution can result in poor performance, due to load imbalance. Effective scheduling in such environments is a difficult problem.

This problem can be overcome by using relatively simple heuristics, if appropriate mechanisms are provided to the scheduler to determine the complexity of the problem wrt. computation and communication. This information is then used to decompose the problem and schedule the work-units in a manner that provides good load balance, and thus good performance. The idea of using dynamic creation of subtasks is also presented. Subtasks are generated according to the requirements of the problem, by taking into consideration the available performance parameters of the system (network bandwidth, latency, CPU availability and performance).

3 IMPROVED MASTER WORKER MODEL

We present in this section an improved version of the master-worker model, which is based on an algorithm with dynamic decomposition of the problem and dynamic number of workers. The improvements regard the performance of the original model, by increasing the time workers are doing computations, and decreasing the time used for communication delays. This is achieved by using different techniques, such as pipelining of the work-units at the worker, redundant computation of the last work-units to decrease the time to finish, overlapped communication and computation at the workers and the master, use of compression to reduce the size of messages. We will describe in the following subsections each of these techniques.

3.1 Pull vs. Push for Work-Units

In the original master-worker model, each time a worker finishes a work-unit, it has to wait until it receives the next work-unit for processing. If this communication time is comparable with the time needed for executing a work-unit, the efficiency of the worker is reduced very much. The time intervals used for communication and computation (processing) are described in Figure 1.

The master-worker model uses the pull technology, which is based on the request/response paradigm. This is typically used to perform data polling. The user (in our case the worker) is requesting data from the publisher (in our case the master). The user is the initiator of the transaction. In contrast, a push technology relies on the *publish/subscribe/distribute* paradigm. The user subscribes once to the publisher, and the publisher will initiate all further data transfers to the user. This is better suited in certain situations.

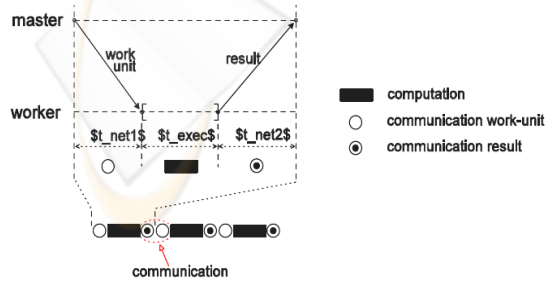

Figure 1: Worker timeline in execution.

We first extend the master-worker model by replacing the pull technology with the push technology, as it is illustrated in Figure 2.

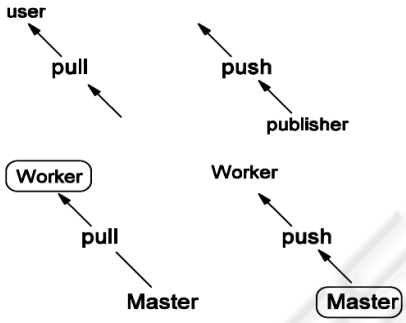

Figure 2: Pull vs. Push technology.

In this model, the worker doesn't send any more requests for work-units. Instead, it first announces its availability to the master when it starts, and the master is responsible for sending further work-units. The workers just wait for work-units, and process them when received. At the end of each work-unit, it sends back the results to the master. The master will further send more work-units to the worker. This moves all decisions about initiating work-units transfers to the master, allowing a better control and monitoring of the overall computation.

3.2 Pipelining of Work-Units

The worker efficiency increases if we reduce the total time spent in waiting for communication. One way to do that is to use work-units pipelining at the worker, thus making sure that the worker has a new work-unit available when it finishes the processing of the current work-unit. Pipelining is achieved by sending more that one work-unit to the workers, as shown in Figure 3. Each worker will have at least one more work-unit in addition to the one being processed at that worker. This is done so that the worker, after finishing a work-unit, will have ready the next one for processing. In the beginning, the master sends more than one work-units to the worker, then after each received result, sends another work-unit to be queued on the worker. The worker does not need to wait again for a new work-unit from the master after sending the result, the next work-unit being already available for processing.

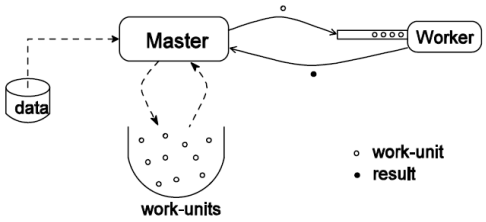

Figure 3: Pipelining of worker tasks.

The immediate advantage of pipelining is that the waiting time for a new work-unit is eliminated. This is described in Figure 4. While the worker is processing the next work-unit, a new work-unit is sent by the master and is queued in the operating system. When using non-blocking communication, the waiting time for sending the result to the master after finishing a computation can be also eliminated.

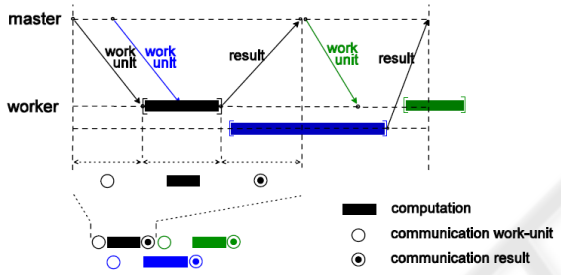

Figure 4: Worker timeline for unit pipeline.

Keeping one new work-unit available at the worker seems to be enough to reduce the waiting time for communication. However, there is a situation when this is not adequate. It can happen that the execution time of a work-unit is much shorter than the communication time (consisting of sending back the result and receiving the new work-unit). In this case, the worker finishes the current work-unit, but the new one is not yet received. Thus, a certain waiting time is involved for receiving it (see Figure 5).

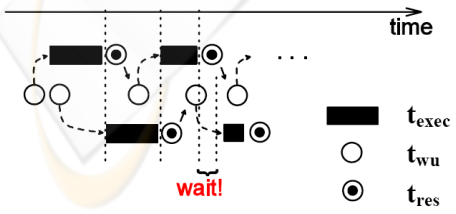

Figure 5: Unit pipeline - worst case.

If there are many work-units with short execution times, than the overall waiting time can increase significantly, reducing the efficiency of the worker. The condition for this not to happen is the following:

$$t_{wu} + t_{res} \leq t_{exec} \quad \text{for the average time values, or}$$
$$\sum_{work-units} (t_{wu} + t_{res}) \leq \sum_{work-units} t_{exec} \quad \text{for the individual time values} \qquad (1)$$

This situation can be improved by pipelining more than two work-units at the worker, thus using a larger pipeline. The master starts by sending out a number of work-units to the worker to fill the pipeline. Each time a result is received back from the worker, the master sends a new work-unit, thus keeping the pipeline full. This algorithm works as long as the average execution time for a work-unit is larger than the average communication time for sending a result and a new work-unit between the worker and the master. If the communication time is too large, the pipeline will eventually become empty and the worker will need to wait for new work-units.

3.3 Sending More Work-Units at a Time

To overcome this situation, the master needs to send more than one work-units per each message. The master starts by sending a message containing more than one work-unit, and then keeps sending as long as the pipeline is not full. Each time it receives a result, it sends another work-unit, to compensate the decreasing number of work-units from the pipe. If the worker sends only one result per message back to the master, and the master sends only one new work-unit, then, eventually the pipeline will become empty. In order to prevent that, the worker will need to send back more results at a time.

We could consider, for example, that the number of results per message is equal to the number of work-units per message sent from the master. In this case, all results from the work-units, which came in one message, are sent back to the master the same way in one message after all of them were successfully computed.

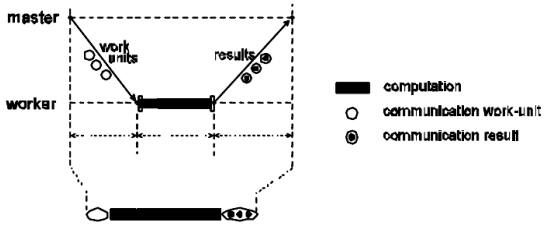

Figure 6: More work-units per message.

This would solve the previous problem if the time to send a larger message (more work-units) were much smaller than the time to send the individual messages (one work-unit). This is usually possible if the data required to describe one work-unit is small enough, so the messages are kept short. However, it could still happen that communication time is larger than the execution time, so that the worker will end up waiting for new work-units. The condition for this not to happen is the following:

$t_{wu,n} + t_{res,m} \leq t_{exec,n}$ for the average time values of multiple work-units per message and execution. (2)

3.4 Adaptive Number of Workers

As mentioned before, in a heterogeneous environment based on idle desktop computers, the total number of workers available could be changing during the computation. New workers can register to the master, and other can become temporarily unavailable. The master controls the total number of workers used for computation, since he is the one sending out work-units to the workers. If necessary, the master can choose not to use all the available workers for computation, only a few of them. This might be for different reasons, as described further.

In the master-worker model, the master can become a bottleneck, especially when there are a lot of workers, which connect to get work-units and send results back. Overloading the master could cause the bottleneck. Because the master has also to do a small amount of processing each time when it receives results from the workers, if too many workers connect to the master, the processing resource available might not be enough and the request will be delayed. There is an upper limit on the number of workers that can connect to the master without overloading it. Finding out this number is not easy and it depends on a variety of parameters from the entire system: computational capabilities of workers and master, communication delays, the amount of processing involved for each results, etc.

Another bottleneck could be caused by too much communication. Considering that there is enough computational power on the master to serve a large number of workers, it could happen that there are too many messages exchanged between the master and workers, thus communication delays can occur. This might happen either because there are too many messages per time unit, or because the amount of data transferred is too high, exceeding thus the available network bandwidth. On the contrary, if there is too few workers used, then the total time of the computation will be too large, not exploiting all the resources available. This suggests that there is some optimum for the number of workers, which can increase the overall efficiency of the whole computation, and reduce the time to complete it.

We define the overall efficiency of the computation as being the ratio between the total amount of time since the beginning of the computation (t_{total}) and the sum of execution times for all completed work-units on all workers:

$$E_{system} = \frac{t_{parallel}}{t_{serial}} = \frac{t_{total}}{\sum_{work-units} t_{exec}} \quad (3)$$

We propose an adaptive algorithm to choose the number of workers, based on performance measures and estimates of execution times for work-units and communication times for sending and receiving messages. The number of workers is automatically reduced if the efficiency of the computation decreases. We employ a heuristic-based method that uses historical data about the behavior of the application. It dynamically collects statistical data about the average execution times on each worker.

3.5 Use of Multithreading

The multithreaded programming paradigm allows the programmer to indicate to the run-time system, which portions of an application can occur concurrently. Synchronization variables control the access to shared resources and allow different threads to coordinate during execution. The paradigm has been successfully used to introduce latency hiding in distributed systems or in a single system where different components operate at different speeds.

The paradigm of multithreading can provide many benefits for the applications. In our situation, it can provide good runtime concurrency, while parallel programming techniques can be easier implemented. The most interesting and probably most important advantages are performance gains and reduced resource consumption.

Operating system kernels supporting multithreaded application perform thread switching to keep the system reactive while waiting on slow I/O services, including networks. In this way, the system continues to perform useful work while the network or other hardware is transmitting or receiving information at a relatively slow rate.

Another benefit of multithreaded programming is in the simplification of the application structure.

Threads can be used to simplify the structure of complex, server-type applications. Simple routines can be written for each activity (thread), making complex programs easier to design and code, and more adaptive to a wide variation in user demands. This has further advantages in the maintainability of the application and future extensions.

The multithreaded paradigm can also improve server responsiveness. Complex requests or slow clients do not block other requests for service, the overall throughput of the server being increased.

4 CONCLUSIONS

We have presented here our extended master-worker model, which has been used in the QADPZ development. Our model includes some refined capabilities that are meant to increase the performance and efficiency of the computation: push of work units, pipelining, sending more work-units at a time, adaptive number of workers, adaptive timeout interval for work units, and use of multithreading. Further measurements need to be made in order to sustain our preliminary benchmark tests that suggest significant improvements regarding efficiency of the computation.

REFERNCES

Berman, F., *et al.*, 2003. *Grid computing: making the global infrastructure a reality*, J. Wiley, New York

BOINC. (2006) Open Source Software for Volunteer Computing and Grid Computing (online) Available from http://boinc.berkeley.edu, (Accessed 25 November 2007)

Constantinescu, Z., 2003 *Towards an autonomic distributed computing environment*, in Proceedings of 14th International Workshop on Autonomic Computing Systems, held in conjunction with 14th Int. Conf. on Database and Expert Systems Applications DEXA 2003, pp. 694-698, Prague, Czech Republic

Constantinescu, Z. & Petrovic, P., 2002, Q2ADPZ* an open source, multi-platform system for distributed computing. ACM Crossroads, 9, pp. 13-20.

Constantinescu, Z., 2008, *A desktop grid computing approach for Scientific Computing and Visualization,* PhD Thesis, Norwegian University of Science and Technology, Trondheim, Norway

Cummings, M. P., (2007), *Grid Computing* (online) Available from http://serine.umiacs.umd.edu/ research/grid.php (Accessed 25 March 2008)

David, P. A., et. al., 2002 *SETI@home: an experiment in public-resource computing*. Communications, ACM, 45, pp. 56-61.

Distributed.Net (2008) (online) Available from http://distributed.net, (Accessed 5 March 2008).

Distributedcomputing.Info (2008) (online) Available from http://distributedcomputing.info, (Accessed 5 March 2008)

Foster, I. & Kesselman, C., 1999. *The grid: blueprint for a new computing infrastructure*, San Francisco, Morgan Kaufmann Publishers.

Foster, I. & Kesselman, C., 2004. *The grid: blueprint for a new computing infrastructure,* Amsterdam, Boston, Morgan Kaufmann.

Garg, V. K., 1996. *Principles of distributed systems, Boston,* Kluwer Academic Publishers.

Garg, V. K., 2002. *Elements of distributed computing*, New York, Wiley-Interscience.

Globus (2007) Globus (online) Available from http:// www.globus.org, (Accessed 15 March 2008)

Juhasz Z., Kacsuk P., Kranzlmuller D., 2004, *Distributed and Parallel Systems: Cluster and Grid Computing*, New York, Springer

Karniadakis, G. & Kirby, R. M., 2003. *Parallel scientific computing in C++ and MPI: a seamless approach to parallel algorithms and their implementation,* NY, Cambridge University Press.

Leopold, C., 2001. *Parallel and distributed computing: a survey of models, paradigms, and approaches,* New York, Wiley.

Mustafee, N. & Taylor, S. J. E., 2006. *Using a desktop grid to support simulation modelling,* in Proceedings of 28th International Conference on *Information Technology Interfaces (ITI 2006)*, Dubrovnik, Croatia, pp. 557 - 562

QADPZ (2008) (online) Available from http:// qadpz.sourceforge.net. (Acc. 1 April 2008).

SETI@HOME (2003) (online) Available from http:// setiathome.ssl.berkeley.edu, (Acc5 May 2003)

SOURCEFORGE (2008) (online) Available from http:// sourceforge.net, Acc 1 April 2008

Sunderam, V. S., 1990 *PVM: a framework for parallel distributed computing*. Concurrency: Practice. Experience, Vol. 2**,** pp 315-339.

Vahid, G., Lionel, C. B., Yvan, L., 2006 *Traffic-aware stress testing of distributed systems based on UML models,* in Proceeding of the 28th International Conference on Software engineering, Shanghai, China

Zomaya, A. Y., 1996 *Parallel and distributed computing handbook,* New York, McGraw-Hill.