Implementation of Various Loop Scheduling Algorithms on Desktop Grids

Ramon P. Ayco Jr. and Jaderick P. Pabico

1 Introduction

Today, the world has become complexed and more demanding that we are finding it more and more difficult keeping up without relying on computers. Many important tasks are being depended on our computers. Weather forecasting, data-banking, surveillance, and different kinds of simulations are only some of the important applications done on computers. And these are all for the sole reason of finishing jobs within the shortest possible time and achieving efficiency beyond the intellectual and physical capabilities of men.

However, many of today's scientific applications performs best requiring the combined powers of more than one computer processor. These applications easily obtain solutions to problems by simultaneously executing stages of the solving process [2]. Most of these applications are considered embarrassingly parallel because they contain loops with large numbers of independent iterations that are computationally intensive [6]. This means that they require the use of extremely fast and powerful computers to work properly and efficiently. Supercomputers are capable of handling such applications due to their collection of specialized (that is, non-standard) features [5]. But these super computers are expensive; and not all users or developers have the resources to avail of such costly computers. Fortunately, the desktop grid (DG) offers cheap, but remarkable computing power that will still enable people to solve computationally intensive tasks in a well-organized, reliable and fast way [7].

But efficiency is not guaranteed just by having a powerful computing infrastructure such as the DG. Without properly distributing the tasks to the processors, a parallel scientific application may result to a performance degradation where the resources available are not fully utilized by the program.

To solve this problem, various loop scheduling algorithms have been designed by researchers. These algorithms aim to optimize embarrassingly parallel scientific applications by effectively scheduling n independent iterations to a set of p processors [8]. These algorithms have been implemented and compared on three different sizes n =(20K, 50K, 100K), using five different number of processors and were proven to effectively reduce application execution times (2, 4, 8, 16, and 32) [6].

The aim of this study is to implement and compare the loop scheduling algorithms using an embarrassingly parallel application executed on DGs also having five different number of PCs (2, 4, 8, 16, and 32). The results of the experiments in this study will show which of the algorithms will perform best as the number of working PCs in the DG increases.

Therefore, with the implementations of loop scheduling algorithms done in this study, a programmer will know which algorithm is best when making and/or optimizing applications for DGs.

2 Review of Literature

2.1 The Grid

Originally, the aim of the Grid research was to realize the vision that anyone could donate resources for the Grid, and anyone could claim resources dynamically according to their needs, e.g. in order to solve computationally intensive tasks. This twofold aim has been, however, not fully achieved yet.

Two trends are observable in the development of DG systems today. Either it could be accessed by lots of users but not anyone can bring resources into it, or vice versa. Anyhow, despite not attaining the two original aim of the Grid, the DGs today still offer the remarkable computing power that is required to perform computationally intensive applications.

The DG concept is originally meant to be implemented on a world-wide scale. But its advantages can also be used for smaller scale computations, combining the power of idle computers at an organizational level. This type of DG is known as the local desktop grid (LDG) [1].

The LDG is perfect for small-scale scientific projects done on a certain institutional, or even departmental, level. The members PCs are connected to a central server to form a large computing infrastructure. Their computational powers will be combined and put to good use; considering that most of them are only used for office applications, e.g. text editing and web browsing [7].

There are several DG systems being put to use today. The most widespread of them is the Berkeley Open Infrastructure for Network Computing (BOINC). BOINC originated form the SETI@home project and is currently the most popular DG system. BOINC can run several different distributed applications and yet, enables PC owners with access to the Internet to join easily by installing a single software package (the BOINC Core Client) and then decide what projects they want to support with the empty cycles of their computers. It has now the aggregated computational power of more than 250,000 participants with about 475 Teraflops, thus, providing the most powerful "supercomputer"of the world.

Another DG system currently being used is the Computer and Automation Research Institute Desktop Grid (SZTAKI DG) of the Hungarian Academy of Sciences (MTA). The SZTAKI DG utilizes BOINC because it is a well-established free and open source platform that has already proven its feasibility and scalability and it provides a stable base for experiments and extensions.

The basic building block of the SZTAKI Desktop Grid is a LDG connecting PCs at the given organizational level. SZTAKI LDG is built on BOINC technology but is oriented for businesses and institutes. In this context these DGs are normally not open for the public, mostly isolated from the outside by firewalls and managed centrally. SZTAKI LDG focuses on making the installation and central administration of the LDG infrastructure easier by providing tools to help the creation and administration of projects and the management of applications. SZTAKI LDG also aims to address the security concerns and special needs arising in a corporate environment by providing a default configuration that is tailored for corporate use and configuration options to allow faster turn around times for computations instead of the long term projects BOINC is intended for. SZTAKI LDG is distributed prepackaged, so it can be easily installed using the apt tool on Debian GNU/Linux systems. After installation the `boinc create project` command can be used to create a new project. This creates everything needed for the project: a working directory, a database, an administrative user account and default configuration files for the web server and BOINC to make the project accessible. Administering is done using the administrative user of the project but for security reasons not by directly logging into it rather, acquiring the rights of this user when needed authenticating with their own password. The system administrator can grant or revoke project administrative rights to/from users via the BOINC admin tool. Project administrators are allowed to install application executables (master, client, validator), start/stop the project and access the database and administrative pages of the project. The `boinc appmgr` tool can be used for automatic installation and configuration of packaged application binaries that come in an archive containing an XML description (provided by the application developer).

The SZTAKI DG also provides the possibility of building a hierarchy of LDGs. This means that an organization could directly access the DG systems of its lower institutions and/or departments to perform computations for higher organizational level projects.

In a hierarchy, DGs on the lower level (child) can ask for work from a higher level DG (parent). When the child node has less work than resources available, the server will contact a parent node in the hierarchical tree and request work from it.

Hierarchical mode is implemented by a hierarchy client, which is run on the child LDG server. This way, the parent does not have to be aware of the hierarchy; it sees the child as one powerful client. The hierarchy client has two sides (see Figure 1): a master side which puts retrieved work-units in the database of the LDG and gets the computed results, and a client side which retrieves work-units from the parent and uploads results [1].

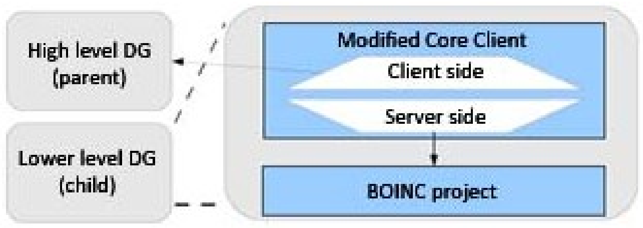

Figure 1: Hierarchy client [1]

2.2 Load Balancing

A size-n problem is embarrassingly parallel if it is composed of n independent tasks, which when solved in parallel, the computation process achieves a speed-up of O(n) with very little interprocess communication. One example of this is a loop with n independent iterations. The ith loop iterate is independent if it doesn't require the result of the computation of the $(i-1)$th iterate, and it does not affect the other iterates [6].

To ensure efficiency of embarrassingly parallel applications, various loop scheduling algorithms have been designed and evaluated by several researchers. These algorithms find an efficient policy to effectively distribute the tasks to the processors such that few, if not none, will be idle. There are two types of scheduling algorithms: the static or offline scheduling and the dynamic or online scheduling.

In static scheduling, the decision of which processor will work on a task is done before program execution by the programmer. The most trivial static scheduling decision for a given problem with n number of tasks on a computational infrastructure with p number of processors, where $p << n$, is to assign n/p tasks to each processor. With this scheduling decision, performance degradation of the parallel system will happen if the actual distribution of the task execution times is skewed. And since the assignment happened at compile time, the idle processors cannot help in computing the load of the busy ones, increasing the realized execution time and effectively reducing speed-up [6].

For this reason, using the static scheduling, also referred to as deterministic scheduling, is only advisable when all information about tasks to be scheduled (i.e. execution times) and their relation to one another is entirely known prior to execution time. These information will be used to make a decision as to how each task is to have a static assignment of which particular processor it is assigned to when it is submitted for execution [3].

Dynamic scheduling algorithms, on the other hand, make the decision of which processor will perform a task on the fly as the program executes. These algorithms are fixed-size chunking (FSC), guided self scheduling (GSS), factoring (FACT), weighted factoring (WFACT), adaptive weighted factoring (AWF), and adaptive factoring (AF).

In FSC, the tasks are grouped into chunks of size c, usually chosen to be $c << n/p$ and, as much as possible, is divisible by p (i.e. p modulo c is 0). The first p chunks of tasks are initially distributed by the scheduler[1] to the processors[2]. When a processor happens to be assigned with tasks that have short execution times, the processor will finish early. And since there are still tasks to be computed, it can request the scheduler for the next chunk of tasks, incurring a cost tc brought about by communication latency. The communication latency is incurred by both the scheduler and the requesting processors, and happens when the processor requests for the next chunk until the scheduler answers with the next assignment. For scientific applications with independent loops, only the indexes of the iterates are communicated by the scheduler, incurring a smaller tc. At the end of the computation, the processors finishing times will vary at a maximum of c, improving the realized execution time of the scientific application.

The GSS scheduling is simply implemented as FSC with $c = 1$. The communication latency is incurred by both the scheduler and the requesting processors, and happens when the processor requests for the next chunk until the scheduler answers with the next assignment. For scientific applications with independent loops, only the indexes of the iterates are communicated by the scheduler, incurring a smaller tc. At the end of the computation, the processors finishing times will also vary at a maximum of c, improving the realized execution time of the scientific application.

FACT is a scheduling algorithm that implements a variable sized chunk c using some factoring rules. The idea of this scheme is to accommodate load imbalances caused by predictable phenomena, such as data-access latency due to I/O bursts and OS interference. One

[1]In practice, the scheduler is usually P_0.
[2]If P_0 is the scheduler, it also distributes to itself.

example of a factoring rule is to schedule chunks of tasks such that c is a fixed factor of those remaining. For example, setting the factor to be $0 < \gamma < 1$, then P_0 is assigned $\gamma \times n$ tasks and P_1 is assigned $\gamma^2 \times n$ tasks. In general, P_j is assigned $\gamma^{j+1}n$ tasks. The selection of c requires that the chunks have high probability of being completed by the processors before the optimal time. The chunk sizes are dynamically computed by the scheduler at runtime. When computing for the larger chunks, the processors incur a relatively low communication overhead. The unevenness of the respective finishing times of the larger chunks can be smoothed over by the smaller chunks made available towards the end of the computation.

WFACT was proposed to take into consideration the effect of processor heterogeneity of the underlying runtime system. This method, derived from FACT, computes c by taking into consideration the relative computing speed of the processor as a weight for during the time of computation. At the start of the computation, the relative processing speeds of the processors will be profiled resulting in an array $S = \{s_0, s_1, \ldots, s_{p1}\}$ of relative processor speeds corresponding to processors $P_0, P_1, \ldots, P_{p1}$. A vector of chunk sizes $C = \{c_0, c_i, \ldots, c_{p1}\}$ will be computed from $c_i = si/smax \times \gamma$, where $smax = max(s_0, s_1, \ldots, s_{p1})$. These chunk sizes are statically assigned to processors and are considered to remain unchanged throughout the entire lifespan of the application. Experiments involving network of workstations, where relative processor speeds are extremely heterogeneous, have shown that WFACT significantly outperformed FACT.

In computational environments where processor workloads vary during the computation, chunk sizes must be assigned to processors dynamically. Many applications whose solutions require a number of iterations over the computation space are expected to benefit from a dynamic adjustment of weights after finishing each chunk of tasks. This aspect is addressed by the AWF, wherein the relative processor speeds are profiled after every computation of a chunk.

AFACT was developed as a general model for FACT, WFACT, and AWF schemes. AFACTs flexibility is suited for highly irregular applications, where even within an iteration over the computation space, the load becomes unpredictably imbalanced. Because of its generality, AFACT reduces into FACT, WFACT, or AWF under specific conditions of processor speeds and task workloads [6].

2.3 Loop Scheduling on Desktop Grids

A study shows that the loop scheduling algorithms are effective in balancing the load of embarrassingly parallel applications [6].

However, the loop scheduling algorithms' performances on DGs have not yet been presented and analyzed as of now. A comparison of each of the loop scheduling algorithms using a Synthetic Embarrassingly Parallel Application[3] (SEPA) will show us which of the algorithms will be best for DGs.

3 Objectives

The main objective of this study is to implement the different loop scheduling algorithms on a DG using the SEPA as a test bed to show which of the algorithms will be best for DGs. Specifically, the following will be done in this study:

- Design and implement the SEPA
- Implement the loop scheduling algorithms using the SEPA as test bed over DG systems
- Determine the performance of each scheduling algorithm over an increasing number of processors in verying SEPA modes

4 Methodology

4.1 Synthetic Embarrassingly Parallel Application

The SEPA will be able to have different distribution of task execution times; it will have four different modes: EQUAL, FRONT-HEAVY, TAIL-HEAVY, and RANDOM. The mode of the execution times distribution will be declared manually by the programmer before the program execution.

In EQUAL mode, each of the tasks $T_i, \forall i = 0, 1, \ldots, n$ will have an identical execution time x.

pseudo-code:

```
x = any integer;
for(i=0 to n-1){
     fibonacci(x);
}
```

[3]The Synthetic Embarrassingly Parallel Application will be a program containing a loop with n independent iterations.

In FRONT-HEAVY mode, the beginning of the loop will have relatively large execution time x that will be gradually decreasing for iterations T_i as i approaches n.

pseudo-code:

```
x = large integer;
for(i=0 to n-1){
    fibonacci(x=decrease(x));
}
```

The TAIL-HEAVY mode will be the opposite of FRONT-HEVY; the beginning of the loop will have a relatively small execution time x that will be gradually increasing for iterations T_i as i approacher n.

pseudo-code:

```
x = small integer;
for(i=0 to n-1){
     fibonacci(x=increase(x));
}
```

In the RANDOM mode, the execution times of the iterations will not have any pattern.

pseudo-code:

```
for(i=0 to n-1){
    fibonacci(x=random());
}
```

4.2 Loop Scheduling Implementations

The implementation of the loop scheduling algorithms will be a single Master-Worker program that will have two modes: MASTER and WORKER. The program will be installed in the individual PCs of DGs varying with five different numbers of PCs (2, 4, 8, 16, and 32). A single Master PC in each of the DGs will run in the MASTER mode of the program while the rest will run on the WORKER mode of the program. Running in the MASTER mode, the Master PC will be able to distribute tasks to all the PCs, including the Master PC itself, by sending the lower and upper bounds of the portion of the loop that needs to be processed. The Worker PCs, on the other hand, will be able to receive the bounds and call SEPA with the received lower and upper bounds as parameters. After processing the assigned portion of the loop, the Worker PCs will request for another set of tasks from the Master PC. The Master PC, in return, will send the next portion of the loop that needs processing—that is if there are still any left.

The MASTER mode of the program will, yet again, have six different scheduling modes, each implementing one of the loop scheduling algorithms—one static scheduling (STATIC) and five dynamic scheduling (FSC, GSS, FACT, WFACT, and AWF). The scheduling mode will be decided and manually chosen by the programmer before program execution.

As you may have noticed, AF scheduling is not included in the implementation because it is just the generalization of FACT, WFACT, and AWF.

The following are the pseudo-codes for the different scheduling algorithms:

p = number of PCs
n = number of loop iterations
c = chunk size

STATIC

```
c = n/p;
for(i=0 to p-1){
     send chunk[i] to processor P[i]
        lowerbound l = i*c
        and
        upperbound u = ((i+1)*c)-1
}
```

FSC

```
c = any integer such that
    c << n/p, and if possible
    p modulo c = 0;

/*send first p chunks to all PCs*/
for(i=0 to p-1){
     send chunk[i] to processor P[i]
        lowerbound l = i*c
        and
       upperbound u = ((i+1)*c)-1
}

until all tasks are done:
    respond to a requesting processor p
    send next chunk of tasks
```

GSS This will be the same as FSC except that the chunk size c will be set to 1.

FACT

```
/*c is a fixed factor of remaining
  tasks*/
f = any number such that 0 < f < 1;

/*send first p chunks to all PCs*/
for(i=0 to p-1){
     c = (f^i*n);
     send chunk[i] to processor P[i]
        lowerbound l = i*c
       and
       upperbound u = ((i+1)*c)-1
}

do until all tasks are done:
respond to a requesting processor p
     compute c
     send next chunk of tasks
```

WFACT

```
/*c is a fixed factor of remaining
    tasks*/
/*f[i] is the the relative speeds
    of processor P[i]*/

/*initally all of the relative speeds
    of all the processors are equal*/
for(i=0 to p-1){
     f[i] = any number
             such that 0<f[i]<1;
}
/*send first p chunks to all PCs*/
for(i=0 to p-1){
    c = (f[i]*n);
    /*update f[i]*/
    f[i] = relative speed of P[i];
     send chunk[i] to processor P[i]
        lowerbound l = i*c
        and
        upperbound u = ((i+1)*c)-1
}

do until all tasks are done:
respond to a requesting processor p
     compute c
     send next chunk of tasks
```

AWF This is basically the same as WF except that, here in AWF, the scheduling process will stop to update the relative speeds of the processors several times, say for example every approximately $n/8$ tasks are done, during execution. This will be done with the following pseudo-code:

```
/*m is the marker of the portion
    of tasks done, initially set to 1*/
m = 1;

do until (all tasks are done){
    respond to a requesting processor
         compute c
         send next chunk of tasks

    if(tasks done => (m/8)*n){

        wait for all the processors to finish

        /*increment marker*/
        m++;

   /*send next p chunks to all PCs*/
   /*start= lowerbound of the next chunk*/
        start = last chunk upperbound+1
        for(i=0 to p-1){
            c = (f[i]*n);
            /*update f[i]*/
            f[i] = relative speed of P[i];
            send chunk[i] bounds to P[i]
               l = i*c + start
               and
               u = ((i+1)*c)-1 + start
        }
    }
}
```

4.3 Comparison of Performances

Since the program in the Master PC will be the one to send tasks and wait for requests of the other PCs, it will be the one to terminate last. Therefore, the execution time of the progam in the Master PC is the system parallel time t_p. The parallel time t_p is sum total of the time spent by the system for computation, which is called the system computing time t_{comp} and the time the system spent for communicating or passing data from the Master PC to the Worker PCs, which is called the system communication time t_c.

Furthermore, the parallel cost C_p can be computed by multiplying the parallel time t_p by the number of processors p.

These performance metrics will be recorded for each of the implementations of the loop scheduling algorithms over the desktop grids with increasing number of processors and then compared with each other.

5 Expected Results

The SEPA will be a trivial case of an embarrassingly parallel problem. It will be a program containing a single loop with n independent iterations or tasks with mock execution times. The SEPA will run when called, and will be accepting two parameters: `sepa(`i, j`)`, where i and j are respectively the lower and upper bounds of the portion of the loop to be executed. Having a sufficiently large n is expected to show the differences of the execution times of the implementations of the different loop scheduling algorithms.

The implementations of the loop scheduling algorithms using SEPA is expected to effectively balance the load to the DGs. This will reduce the execution time relative to when SEPA is executed sequentially.

The comparisons of the execution times of each of the implementations of different loop scheduling algorithms on DGs with different number of PCs (2, 4, 8, 16, and 32) will show which of the algorithms is best to use when making and/or optimizing applications for DGs.

References

[1] Z. Balaton, G. Gombás, P. Kacsuk, Á. Kornafeld, J. Kovcs, A.C. Marosi, G. Vida, N. Podhorszki, and T. Kiss. *SZTAKI Desktop Grid: a Modular and Scalable Way of Building Large Computing Grids.* IEEE, 2007.

[2] B. Codenotti and M. Leoncini.*Introduction to Parallel Processing.* Addison-Wesley Publishing Inc., 1993.

[3] H. El-Rewini, T.G. Lewis, and H.H. Ali. *Task Scheduling in Parallel and Distributed Systems.* Prentice-Hall Inc., 1994.

[4] A.C. Marosi, G. Gombas, and Z. Balaton. *Secure application deployment in the hierarchical local desktop grid.* In Proceedings of DAPSYS 2006 6th Austrian-Hungarian Workshop on Distributed and Parallel Systems, Innsbruck, Austria, 2006.

[5] H.S. Morse. *Practical Parallel Computing.* Academic Press, Inc., 1994.

[6] J.P. Pabico. *Dynamic Load Balancing Algorithms for Embarrassingly Parallel Tasks.* Computing Society of the Philippines, 2009.

[7] Computer Automation Research Institute (SZTAKI) of the Hungarian Academy of Sciences (MTA), (2006). Available: http://www.desktopgrid.hu

[8] T. Tabirca, S. Tabirca, and L.T. Yang. *An O(log(p)) Algorithm for the Discrete Feedback Guided Dynamic Loop Scheduling.* In Proceedings of the 20th International Conference on Advanced Information Networking and Applications (AINA06), 2006.