Implementation of Various Loop Scheduling Algorithms on Desktop Grids

Ramon P. Ayco Jr. and Jaderick P. Pabico

1 Introduction

The world has become more complexed and demanding that we are finding it difficult keeping up without relying on computers. Many of our important tasks (eg. weather forecasting, data-banking, surveillance, different simulations, etc.) are being implemented on them for the sole reason of finishing them within the shortest possible time and achieving efficiency beyond the intellectual and physical capabilities of men.

However, many of today's scientific applications perform best by using the combined computing powers of more than one computer processor. These applications easily obtain solutions to problems by simultaneously executing stages of the solving process [2] and, thus, are commonly called as parallel scientific applications. Among these applications are those which are called Embarrassingly Parallel applications 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 with multiple number of processors to work properly and efficiently.

Super-computers are capable of handling such applications due to their collection of specialized (that is, non-standard) features [5], but these super-computers are highly expensive. Fortunately, the Desktop Grid (DG) offers a remarkable computing power that also enables people to solve computationally intensive tasks in a well-organized, reliable and fast way, but with a lower cost [7].

However, efficiency is not guaranteed by simply having a powerful computing infrastructure such as the DG. Without a system that properly distributes the tasks to the computer processors, a parallel scientific application may result to a performance degradation wherein the available resources are not fully utilized.

To address 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].

In this study, the loop scheduling algorithms have been implemented using an embarrassingly parallel application on DGs with 10 varying sizes (2, 4, 6, ... upto 20 computers) as test bed for four different scenarios with varying distribution of task execution times. After four replications of the implementations, their performances have been compared.

2 Review of Literature

2.1 The Desktop Grid

Originally, the Grid research aimed to create a system where anyone can donate resources to it and dynamically claim resources from it according to their needs, e.g. in solving computationally intensive tasks. However, this twofold aim has been not yet fully achieved.

Right now, the development of DG systems follow two different trends with respect to the aforementioned aim. In the first trend, DG systems could be accessed by many of users but not anyone can bring resources into it. Conversely, in the second trend, anyone can bring resources in the DG systems, but not everyone can access them. Nonetheless, despite not attaining the original aim of the Grid, the DGs today still offer remarkable computing power that is necessary for implementing computationally intensive applications.

The DG concept was originally meant to be implemented on a world-wide scale but its advantages can also be useful 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 convenient for small-scale scientific projects done on a certain institutional, or even departmental, level. The computers are connected to a central server to form a large computing infrastructure. Their computational powers are 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 used today. The most widespread of them is the Berkeley Open Infrastructure for Network Computing (BOINC). BOINC originated from 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) which utilizes the BOINC platform. The basic building block of the SZTAKI Desktop Grid is a LDG connecting computers 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 to the public, mostly isolated from outside by firewalls and managed centrally.

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.

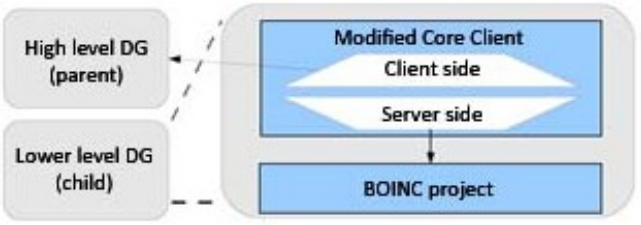

Figure 1: Hierarchy client [1]

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].

2.2 Load Balancing

A size-n problem is embarrassingly parallel if it is composed of n independent tasks, which when solved in parallel, results in the computation process achieving 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, deciding 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. Because 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 are entirely known prior to execution. These information will be used deciding 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, decide which processor will perform a task on the fly. 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 finishes early. Because there are still tasks to be computed, the processor can request for the next chunk of tasks from the scheduler, 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 of tasks and waits for the scheduler to respond 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 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.

[1] In practice, the scheduler is usually P_0.

[2] If P_0 is the scheduler, it also distributes to itself.

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].

AF and AWF are not included in the implementations in this study because they are just generalizations of FACT, WFACT.

2.3 Loop Scheduling on Desktop Grids

A study showed 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.

3 Objectives

The main objective of this study was to implement the loop scheduling algorithms in ten DGs with increasing number of computers on four different scenarios with varying distribution of task execution times (Equal, Front-Heavy, Tail-Heavy, and Random). This is to determine which of the loop scheduling algorithms best balances the load of embarrassingly parallel applications in DGs on known scenarios which causes performance changes in parallel systems.

Specifically, the following are done in this study:

- Design the Synthetic Embarrassingly Parallel Application (SEPA) which is a trivial case of an embarrassingly parallel application.
- Implement the loop scheduling algorithms using the SEPA as test bed over DG systems
- Compare the performance of each scheduling algorithm over an increasing number of computers

4 Methodology

4.1 Synthetic Embarrassingly Parallel Application

The SEPA is a program designed to have four modes—each having a different distribution of task execution times. These four modes are: EQUAL, FRONT-HEAVY, TAIL-HEAVY, and RANDOM.

In EQUAL mode, each of the tasks $T_i, \forall i = 0, 1, \ldots, n$ has an identical execution time x. This mode was used to simulate embarrassingly parallel applications where each of the independent iterations have identical load.

In FRONT-HEAVY mode, the beginning of the loop has a relatively large execution time x that decreases for iterations T_i as i approaches n. In this mode, the distribution of task execution times is skewed to the left. This mode was used to simulate embarrassingly parallel applications where the load is relatively heavy in the start of the program but gradually lightens as it ends.

The TAIL-HEAVY mode is the opposite of FRONT-HEAVY. The beginning of the loop has a relatively small execution time x that gradually increases for iterations T_i as i approaches n. In this mode, the distribution of the task execution times is skewed to the right. This mode was used to simulate embarrassingly parallel applications where the load is relatively light in the start of the program but gradually weighs up as it ends.

In the RANDOM mode, the execution times of the iterations does not have any pattern. This mode was used to observe which of the loop scheduling algorithms is best when dealing with embarrassingly parallel applications with independent iterations that has unpredictable load sizes.

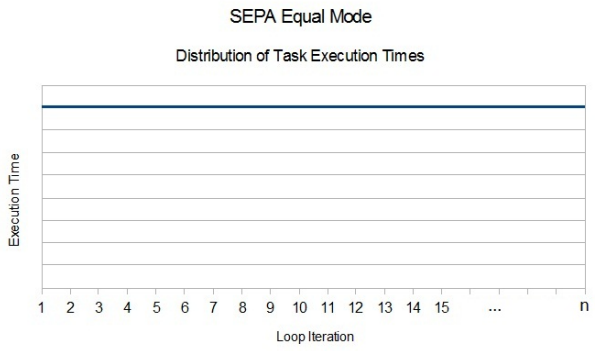

Figure 2: Distribution of the task execution times in SEPA Equal Mode. The loop iterations in this mode have identical task execution times.

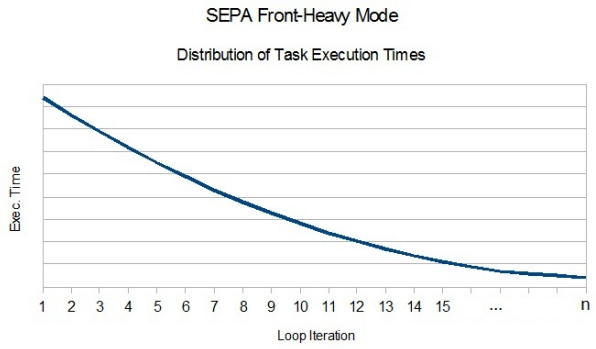

Figure 3: Distribution of the task execution times in SEPA Front-Heavy Mode. In this mode, the iterations in the beginning of the loop have relatively large execution times that decrease as program executes.

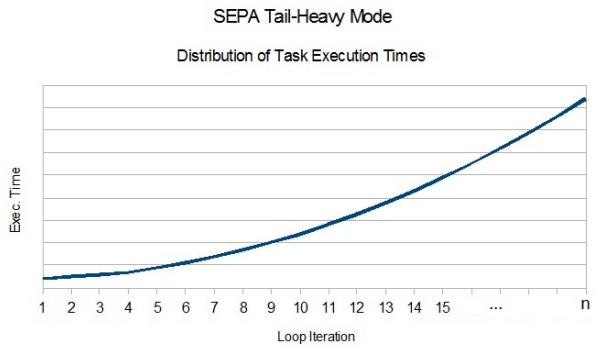

Figure 4: Distribution of the task execution times in SEPA Tail-Heavy Mode. In this mode, the iterations in the beginning of the loop have relatively small execution times that increase as the program executes.

4.2 Implementation of Loop Scheduling Algorithms

Each of the loop scheduling algorithms implemented in this study—one static scheduling (STATIC) and four dynamic scheduling (FSC, GSS, FACT, WFACT)—was implemented as Master-Worker programs where the Master computers distribute the tasks to the Workers. Each program was installed in the ten DGs with varying sizes (2, 4, 6, ... upto 20 computers). The Master computers distributed the tasks to all the computers by sending the lower and upper bounds of the portion of the loop that needs to be computed in the SEPA. The Workers, on the other hand, received the bounds and computed the portion of the loop in the SEPA. After processing the assigned portion, the Workers sent the results to the Master PC and requested for another set of tasks. The Master PC, in response, sent the next portion of the loop that needs processing. This request-response cycles between the Master computers and the Workers went on until all the loops were processed.

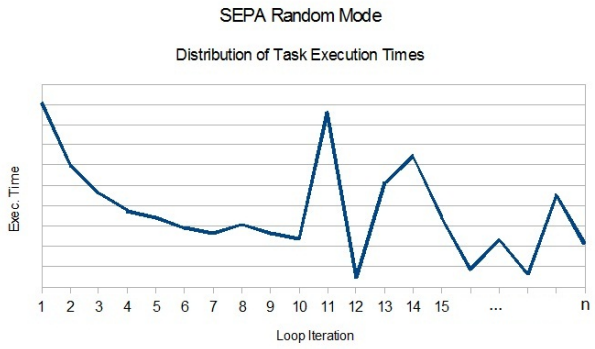

Figure 5: Distribution of the task execution times in SEPA Random Mode. The execution times of the loop iterations follow no pattern in this mode.

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

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

STATIC

```
1    c = n/p;
2    for(i=0 to p-1){
3         send chunk[i] to processor P[i]
4            lowerbound l = i*c
5            and
6            upperbound u = ((i+1)*c)-1
7    }
```

FSC

```
 1    c = any integer such that
 2        c << n/p, and if possible
 3        p modulo c = 0;
 4
 5    /*send first p chunks to all PCs*/
 6    for(i=0 to p-1){
 7         send chunk[i] to processor P[i]
 8            lowerbound l = i*c
```

```
 9            and
10           upperbound u = ((i+1)*c)-1
11    }
12
13    until all tasks are done:
14        respond to a requesting processor p
15        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

```
 1    /*c is a fixed factor of remaining
 2      tasks*/
 3    f = any number such that 0 < f < 1;
 4
 5    /*send first p chunks to all PCs*/
 6    for(i=0 to p-1){
 7         c = (f^i*n);
 8         send chunk[i] to processor P[i]
 9            lowerbound l = i*c
10           and
11           upperbound u = ((i+1)*c)-1
12    }
13
14    do until all tasks are done:
15    respond to a requesting processor p
16         compute c
17         send next chunk of tasks
```

WFACT

```
 1    /*c is a fixed factor of remaining
 2        tasks*/
 3    /*f[i] is the the relative speeds
 4        of processor P[i]*/
 5
 6    /*initially all of the relative speeds
 7        of all the processors are equal*/
 8    for(i=0 to p-1){
 9         f[i] = any number
10                 such that 0<f[i]<1;
11    }
12    /*send first p chunks to all PCs*/
13    for(i=0 to p-1){
14        c = (f[i]*n);
15        /*update f[i]*/
16        f[i] = relative speed of P[i];
17         send chunk[i] to processor P[i]
18            lowerbound l = i*c
19            and
20            upperbound u = ((i+1)*c)-1
21    }
22
23    do until all tasks are done:
24    respond to a requesting processor p
25         compute c
26         send next chunk of tasks
```

4.3 Comparison of Performances

After four replications of the implementations, several performance metrics were recorded and then compared with each other.

First is the system parallel time τ_p. This is the sum total of the time spent by a parallel system for computation t_{comp} and the time spent for communication and waiting t_c. In the implementations done in this study, this is the execution times of the progams in the Master computers since the Master computers were the ones that started first and terminated last in every implementation.

Second is the parallel cost C_p. This was computed by multiplying the parallel time τ_p to the number of processors p in each DG[6].

5 Results and Discussions

The implementations done in this study were used to analyze the performances of the loop scheduling algorithms on DGs. Results show that there are significant differences between the performances when balancing load of embarrassingly parallel applications when the task execution times of the iterations are equally distributed, front-heavy, tail-heavy, and randomly distributed.

5.1 Performance Metrics

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.