**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 (e.g. 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, 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. These applications 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 appli- cations 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 at a lower cost [7].

However, efficiency is not guaranteed by simply hav- ing 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 al- gorithms have been designed by researchers. These algorithms aim to optimize Embarrassingly Parallel scientific applications by effectively scheduling n in- dependent iterations to a set of p processors [8].

In this study, the Loop Scheduling algorithms have been implemented to an Embarrassingly Parallel application on DGs with varying number of computers (2, 4, 6, 8, 10, 12, 14, 16, 18, and 20) us- ing 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 sys- tem where anyone can donate resources to it, and dynamically claim resources from it according to their needs, e.g. in solving computationally inten- sive tasks. However, this two-fold aim has been not yet fully achieved.

Right now, the development of DG systems follows two different trends with respect to the aforementioned aim. In the first trend, DG systems could be accessed by many users but not anyone can bring resources into it. Conversely, in the second trend, anyone can bring resources in the Grid systems, but not everyone can access them.. Nonetheless, despite not attaining the two original aims of the Grid, the DGs today still offer re- markable computing power that is necessary to performing 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].

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 com- putational 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 Berke- ley 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 partici- pants 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 Desk- top Grid (SZTAKI DG) of the Hungarian Academy of Sciences (MTA). The SZTAKI DG utilizes BOINC be- cause it is a well-established free and open source plat- form that has already proven its feasibility and scal- ability and provides a stable base for experiments and extensions.

The basic building block of the SZTAKI Desktop Grid is an 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 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. SZ- TAKI 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 by 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.

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

2.2 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 com- pile time, the idle processors cannot help in computing the load of the busy ones, increasing the realized exe- cution 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 in 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 factor- ing (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 scheduler1 to the processors2 . 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 answer 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 real- ized 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 < γ < 1, then

P0 is assigned γ × n tasks and P1 is assigned γ2 × n

tasks. In general, Pj is assigned γj+1 n tasks. The

selection of c requires that the chunks have high prob-

ability 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

1 In practice, the scheduler is usually P0 .

2 If P0 is the scheduler, it also distributes to itself.

3

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 under- lying 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 = {s0 , s1 , . . . , sp1 } of relative processor speeds

corresponding to processors P0 , P1 , . . . , Pp1 . A vector of chunk sizes C = {c0 , ci , . . . , cp1 } will be computed from ci = si/smax × γ, where smax =max(s0 , s1 , . . . , sp1 ). 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 it- erations 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 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 to determine which of them are best for developing or optimizing embarrassingly parallel applications for DGs on four different scenarios with varying distribution of task execution times. The Synthetic Embarrassingly Parallel Application (SEPA), which is a trivial embarrassingly parallel problem, was used as test bed for the implementations.

Specifically, the following are done in this study:

• Design and implement the SEPA

• 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 processors in varying SEPA modes

4 Methodology

4.1 Synthetic Embarrassingly Parallel

Application

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

In EQUAL mode, each of the tasks Ti , ∀i =

0, 1, . . . , n has an identical execution time x. This

mode is used to simulate embarrassingly parallel applications where each of the independent iterations have identical load.

pseudo-code:

1 x = any integer;

2 for(i=0 to n-1){

3 fibonacci(x);

4 }

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

4

pseudo-code:

1 x = large integer;

2 for(i=0 to n-1){

3 fibonacci(x=decrease(x));

4 }

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 Ti as i approaches n. In this mode, the distribution of the task execution times is skewed to the right. This mode is 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.

pseudo-code:

1 x = small integer;

2 for(i=0 to n-1){

3 fibonacci(x=increase(x));

4 }

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

Pseudo-code:

1 for(i=0 to n-1){

2 fibonacci(x=random());

3 }

4.2 Implementation of Loop Scheduling Algorithms

The implementation of the loop scheduling algorithms is a Master-Worker program. This program is installed in the computers of ten DGs with sizes 2,

4, 6, 8, 10, 12, 14, 16, 18, and 20 computers. One computer in each of the DGs is made the Master and it is the one responsible for distributing the tasks while the rest are Workers and are the ones responsible for the computations. The Master PC distributes the tasks to all the computers by sending the lower and upper bounds of the portion of the loop that needs to be computed and the mode of the SEPA to be used. The Workers, on the other hand, receives the bounds and runs the SEPA with the received bounds. After processing the

assigned portion of the loop, the Workers sends the results to the Master PC and then requests for another set of tasks. The Master PC, in return, sends the next portion of the loop that needs processing—if there are still any left.

The MASTER program has five different scheduling modes, each implementing one of the loop scheduling algorithms—one static scheduling (STATIC) and four dynamic scheduling (FSC, GSS, FACT, WFACT). AF and AWF are not included in the implementations in this study because they are just generalizations of FACT, WFACT.

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.

5

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

AWF This is basically the same as WF except that, here in AWF, the scheduling process will stop to up- date 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:

23 /\*m is the marker of the portion

24 of tasks done, initially set to 1\*/

25 m = 1;

26

27 do until (all tasks are done){

28 respond to a requesting processor

29 compute c

30 send next chunk of tasks

31

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

33

34 wait for all the

35 processors to finish

36

37 /\*increment marker\*/

38 m++;

39

40 /\*send next p chunks to all PCs\*/

41 /\*start= lowerbound of the next chunk\*/

42 start = last chunk upperbound+1

43 for(i=0 to p-1){

44 c = (f[i]\*n);

45 /\*update f[i]\*/

46 f[i] = relative speed of P[i];

47 send chunk[i] bounds to P[i]

48 l = i\*c + start

49 and

50 u = ((i+1)\*c)-1 + start

51 }

52 }

53 }

4.3 Comparison of Performances

Since the program in the Master PC is the one which sends tasks and waits for requests, it is the one to start first and is also the one to terminate last. Therefore, the execution time of the program in the Master PC is the system parallel time τp which is the sum total of the time spent by the system for computation tcomp and the time the system spent for communication and waiting tc .

Furthermore, the parallel cost Cp can be computed by multiplying the parallel time τp by the number of processors p in the system [6].

These performance metrics are recorded for each of the implementations and then compared with each other.

6

5 Results and Discussions

The implementations of the loop scheduling algorithms on DGs (with sizes 2, 4, 6, 8, 10, 12, 14,

16, 17,8, and 20) using SEPA as a test bed on four modes (EQUAL, FRONT-HEAVY, TAIL-HEAVY, and RANDOM) shows that the scheduling algorithms effectively balances the work load to the DGs and reduces the execution times relative to when SEPA is executed sequentially in a single PC.

Figures 2, 3, 4, and 5 are comparisons of the execution times of each of the implementations. Hahahaha

:)

Figure 4: SEPA on TAIL-HEAVY mode

Figure 2: SEPA on EQUAL mode

Figure 3: SEPA on FRONT-HEAVY mode

References

[1] Z. Balaton, G. Gombas, P. Kacsuk, A´ . Kornafeld, J. Kovcs, A.C. Marosi, G. Vida, N. Podhorszki, and T. Kiss. SZTAKI Desktop Grid: a Modular

Figure 5: SEPA on RANDOM mode

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

7

(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.

8