

Parallel Program Execution on a Heterogeneous PC Cluster Using Task Duplication

YU-KWONG KWOK

Department of Electrical and Electronic Engineering
The University of Hong Kong, Pokfulam Road, Hong Kong

Email: ykwok@eee.hku.hk

Abstract[†]—In this paper, we propose to use a duplication based approach in scheduling tasks to a heterogeneous cluster of PCs. In duplication based scheduling, critical tasks are redundantly scheduled to more than one machine in order to reduce the number of inter-task communication operations. The start times of the succeeding tasks are also reduced. The task duplication process is guided given the system heterogeneity in that the critical tasks are scheduled or replicated in faster machines. The algorithm has been implemented in our prototype program parallelization tool for generating MPI code executable on a cluster of Pentium PCs. Our experiments using three numerical applications have indicated that heterogeneity of PC cluster, being an inevitable feature, is indeed useful for optimizing the execution of parallel programs.

Keywords: Scheduling, task graphs, algorithms, parallel processing, heterogeneous systems, PC cluster computing, task duplication, resource management.

1 Introduction

Recently we have witnessed an increasing interest in employing a network of PCs connected by a high-speed network to tackle many computationally intensive parallel applications [9], [18]. Parallel processing using a cluster of machines, also commonly called *cluster computing*, enables a much larger community of users than ever before to efficiently tackle many difficult optimization problems on a readily available

platform [9], [18]. However, realizing the goal of efficient cluster computing entails handling a number of resource management chores [18]. One of the most important problems is the scheduling of tasks. Indeed, to effectively harness the aggregate computing power of such a heterogeneous cluster, it is crucial to judiciously allocate and sequence the tasks on the machines. In a broad sense, the scheduling problem exists in two forms: *dynamic* and *static*. In dynamic scheduling, few assumptions about the parallel program can be made before execution, and thus, scheduling decisions have to be made on-the-fly. The goal of a dynamic scheduling algorithm as such includes not only the minimization of the program completion time but also the minimization of scheduling overhead, which represents a significant portion of the cost paid for running the scheduler. In a cluster of PCs environment, such dynamic scheduling algorithms usually employ the so-called “idle-cycle-stealing” approach [5] which attempts to dynamically balance the work load evenly across all the machines. However, when the objective of scheduling is to minimize the execution time of a parallel application, such dynamic scheduling strategies are not suitable.

On the other hand, the approach of using static scheduling algorithms [11], [12], [22], which can afford to use longer time to generate an optimized schedule off-line, is particularly effective for many scientific applications such as the adaptive simulation of N-body problem, object recognition using iterative image processing algorithms, and

[†] This research was jointly supported by a research initiation grant from HKU CRCG under contract number 10202518, a research grant from the Hong Kong Research Grants Council under contract number HKU 7124/99E, and a seed funding grant from HKU URC under contract number 10203010.

some other numerical applications [1], [3], [4], [13], [14], [19], [25] because the characteristics of such applications can be determined at compile-time. A parallel program, therefore, can be represented by a directed acyclic task graph [3], in which the node weights represent task processing times and the edge weights represent data dependencies as well as the communication times between tasks [3], [6]. The static scheduling problem is, in general, NP-complete [5], [8] and there have been many heuristics suggested in the literature for scheduling a parallel machine. However, the problem of scheduling tasks to a cluster is a relatively less explored topic. Specifically, there are two difficult research issues to be tackled in the scheduling problem for cluster computing:

- 1) *Communication overhead*: The communication overhead in a network of PCs is still very significant relative to the processing power of the machines [9]. Thus, to avoid offsetting the gain from parallelization by excessive communication overhead, the tasks should be scheduled in such a manner that the number of communications is kept small.
- 2) *Heterogeneity*: In a PC cluster, which typically undergoes continual upgrading, heterogeneity in the hardware configuration is unavoidable. Heterogeneity can be a potential problem for some highly regular applications (e.g., some data parallel problems). However, it has been demonstrated that heterogeneity is useful for further enhancing the performance of irregularly structured parallel application [7], [21], by exploiting the affinity of different tasks to different machines.

In this study, we propose to use a *duplication* approach to scheduling the tasks to the cluster. In

duplication based scheduling, critical tasks are redundantly scheduled to more than one machines in order to reduce the number of inter-task communication operations. The start times of the succeeding tasks are also reduced. There have been many duplication approaches suggested in the literature [1], [10], [15], [16], [17], [20]. However, all these methods are designed for homogeneous parallel architectures. Furthermore, the previous approaches are all evaluated based on simulations rather than using real applications with a parallelizing compiler. In our proposed approach, the task duplication process is guided by tracking the critical path of the task graph given the system heterogeneity in that the critical tasks are scheduled or replicated in faster machines. Task duplication is indeed particularly effective for heterogeneous systems because the overall completion time of an application is usually determined by a subset of tasks (i.e., the *critical-path*, discussed in detailed in Section 2) which can be scheduled to execute efficiently on the faster machines. We have implemented this duplication based scheduling algorithm in the parallel code generator of a prototype program parallelization tool [2], which generates MPI code executable on a network of Pentium PCs. The system on which we tested our approach is shown schematically in Figure 1. Our experiments using several real applications have demonstrated that the duplication technique is very effective in reducing the completion time of the applications on a heterogeneous cluster of Pentium II PCs connected via a Fast Ethernet switch.

The remainder of this paper is organized as follows. In the next section, we describe in detail the model used and the design considerations of the duplication algorithm. Section 3 includes the results of our performance study. The last section concludes the paper.

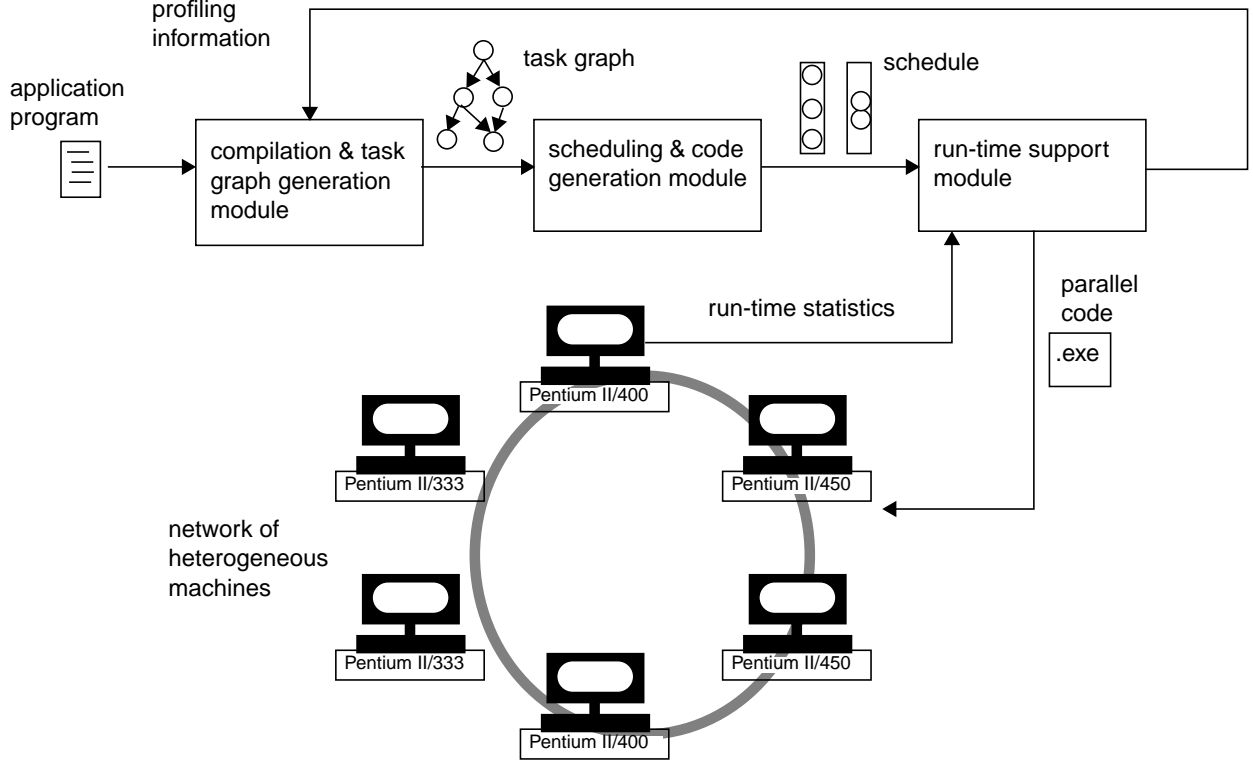


Figure 1: System support for high-performance computing on a heterogeneous cluster.

2 Scheduling for a Heterogeneous PC Cluster

In this section, we first describe our scheduling model, followed by a discussion of the duplication techniques employed in our scheduling module of the parallel code generator.

2.1 The Model

A parallel program is composed of n tasks $\{T_1, T_2, \dots, T_n\}$ in which there is a partial order: $T_i < T_j$ implies that T_j cannot start execution until T_i finishes due to the data dependency between them. Thus, a parallel program can be represented by a directed acyclic *task graph* [3]. Parallelism exists among independent tasks— T_i and T_j are said to be independent if neither $T_i < T_j$ nor $T_j < T_i$. Each task T_i is associated with a nominal execution cost τ_i which is the execution time required by T_i on a reference machine in the

heterogeneous system. Similarly, a nominal communication cost c_{ij} is associated with the message M_{ij} from T_i to T_j . Assume there are e messages where $(n-1) \leq e < n^2$ so that the task graph is a connected graph.

To model heterogeneity of the target system which consists of m processors $\{P_1, P_2, \dots, P_m\}$, *heterogeneity factors* are used. For example, if a task T_i is scheduled to a processor P_x , then its actual execution cost is given by $h_{ix}\tau_i$ where h_{ix} is the heterogeneity factor which is determined by measuring the difference in processing capabilities (e.g., speed) of processor P_x and the reference machine with respect to task T_i . Similarly, if a message M_{ij} is scheduled to the communication link L_{xy} between processors P_x and P_y , its actual communication cost is given by $h'_{ijxy}c_{ij}$. An example parallel program graph is shown in Figure 2.

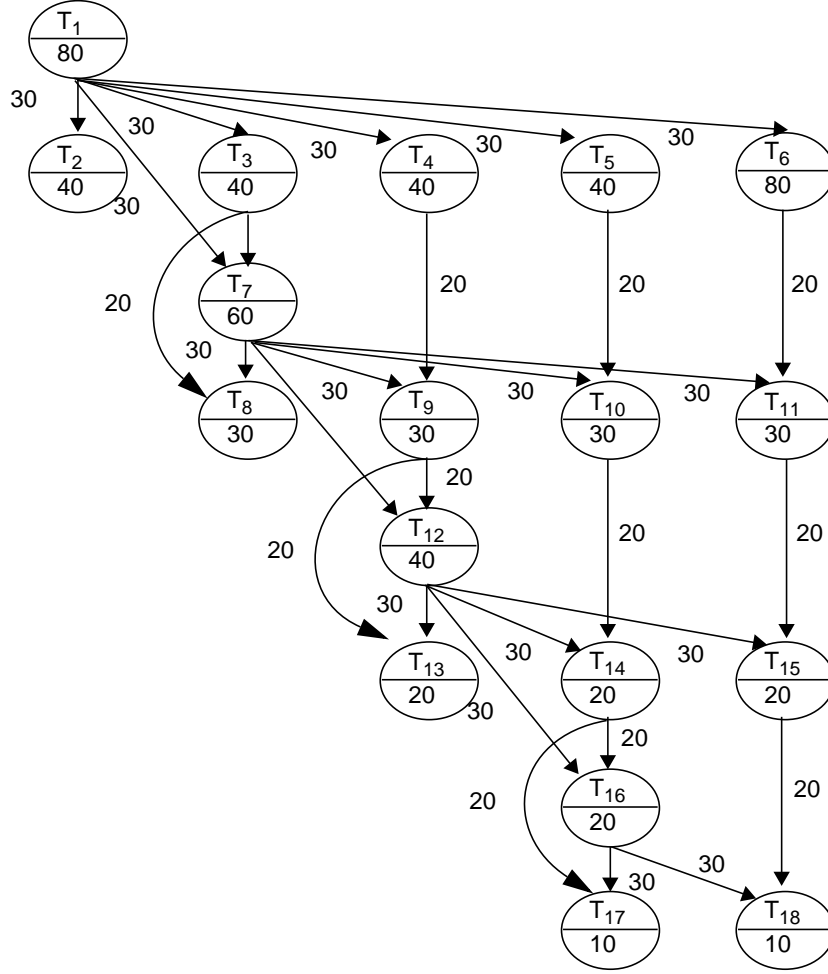


Figure 2: A Gaussian elimination task graph.

The start time and finish time of a message M_{ij} from T_i to T_j on a communication link L_{xy} are denoted by $MST(M_{ij}, L_{xy})$ and $MFT(M_{ij}, L_{xy})$, respectively. Obviously, we have:

$$MFT(M_{ij}, L_{xy}) = MST(M_{ij}, L_{xy}) + h'_{ij,xy} c_{ij}$$

The start time of a task T_i on processor P_x is denoted by $ST(T_i, P_x)$ which critically depends on the task's *data ready time* (DRT). The DRT of a task is defined as the latest arrival time of messages from its predecessors. The finish time of a task T_i is given by $FT(T_i, P_x) = ST(T_i, P_x) + h_{ix} \tau_i$. The objective of scheduling is to minimize the maximum FT , which is called the *schedule length* (SL).

2.2 Parallel Code Generation with Duplication Based Scheduling

The proposed duplication scheduler is designed as a core module in the CASCH (Computer-Aided SCHEDuling) tool [2]. The system organization of the CASCH tool is shown in Figure 3. It generates a task graph from a sequential program, uses a scheduling algorithm to perform scheduling, and then generates the parallel code in a scheduled form for a cluster of workstations. The timings for the tasks and messages are assigned through a timing database which was obtained through profiling of the basic operations [2], [6]. As soon as the task graph is generated, the duplication based scheduler is invoked.

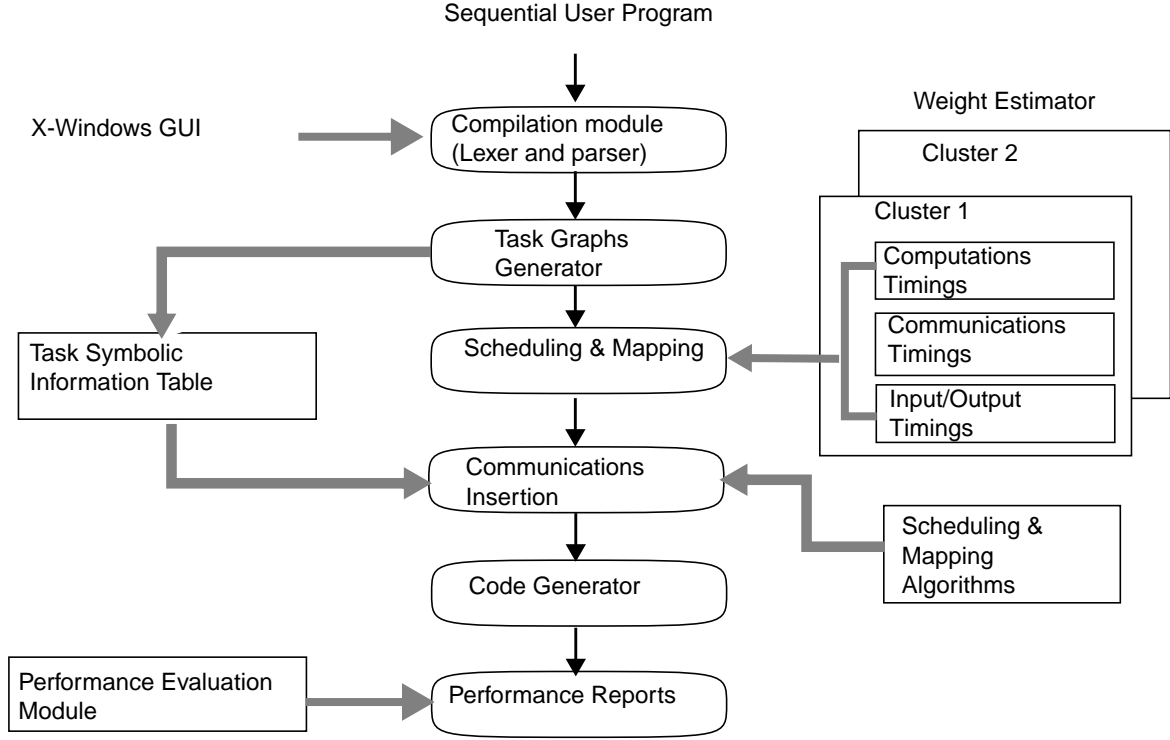


Figure 3: The organization of the CASCH tool.

To minimize the overall execution time of the application on the cluster, the scheduler first determines which tasks are more critical so that they need to be scheduled to start at earlier time slots, possibly by duplicating their ancestors. In a task graph, the *critical-path* (CP), which consists of tasks forming the longest path, is such an important structure because the tasks on the CP potentially determine the overall execution time. To determine whether a task is a CP task, we can use two attributes: *t-level* (top level) and *b-level* (bottom level) [13], [24]. The *b-level* of a task is the length of the longest path beginning with the task. The *t-level* of a task is the length of the longest path reaching the task. Thus, all tasks on the CP have the same value of $(t\text{-level} + b\text{-level})$, which is equal to the length of the CP. Based on this observation, we can easily partition the parallel program into three categories: CP (critical path), IB (in-branch), and

OB (out-branch) tasks. The IB tasks are ancestors of CP tasks but are not CP tasks themselves. The OB tasks are neither CP nor IB tasks and as such, are relatively less important. This partitioning can be performed in $O(e)$ time because the *t-level* and *b-level* of all tasks can be computed by using depth-first search. A task with a larger *b-level* implies that it is followed by a longer chain of tasks, and thus, is given a higher priority. A procedure is outlined below for constructing a scheduling list based on the partitioning.

ALGORITHM 1: CONSTRUCTION OF SCHEDULING LIST

Input: a program task graph with n tasks $\{T_1, T_2, \dots, T_n\}$

Output: a serial order of the tasks

1. compute the *t-level* and *b-level* of each task by using depth-first search;
2. identify the CP; if there are multiple CPs,

- select the one with the largest sum of execution cost and ties are broken randomly;
3. put the CP task which does not have any predecessor to the first position of the serial order;
4. $i \leftarrow 2$; $T_x \leftarrow$ the next CP task
5. while not all the CP tasks are included do
6. if T_x has all its predecessors in the serial order then
7. put T_x at position i and increment i ;
8. else let T_y be the predecessor of T_x which is not in the serial order and has the largest b -level (ties are broken by choosing the predecessor with a smaller t -level);
9. if T_y has all its predecessors in the serial order then put T_y at position i and increment i ; otherwise, recursively include all the ancestors of T_y in the serial order such that the tasks with a larger b -level are included first;
10. repeat the above step until all the predecessors of T_x are in the serial order;
11. put T_x at position i and increment i ;
12. $T_x \leftarrow$ the next CP task;
13. append all the OB tasks to the serial order in descending order of b -level;

Using the above scheduling list, we can determine which tasks have to be considered first in the duplication process. During scheduling, the CP tasks are always considered first. However, we cannot attempt to schedule the CP tasks unless all of their ancestor tasks, which need not be CP tasks themselves, are scheduled. Thus, we use a recursive approach. For each CP task, we first recursively check whether its ancestors are scheduled. If not, then the candidate for scheduling will be changed to the unscheduled ancestor which is at the earliest position on the scheduling list. To actually schedule a task, we try to minimize its finish time by attempting to schedule it to the fastest machine. Duplication is employed for the minimization of finish times in that as many ancestors as possible are inserted before the task. The duplication process

will stop when the finish time of the task starts to increase or the time slot has been used up. The order of selecting ancestors for duplication is governed by the scheduling list. The heterogeneity factors h_{ix} are also used for determining the finish times. After all the CP tasks are scheduled (and hence all the IB tasks), the OB tasks are considered for scheduling. To avoid using an excessive number of machines, we attempt to schedule the OB tasks without using duplication. This is useful because the OB tasks usually do not affect the overall completion time and, thus, need not be scheduled to finish as soon as possible. However, if such a conservative approach fails—that is, the overall completion time is increased by scheduling a certain OB task without using duplication, then the same recursive duplication process will be applied to the OB task. The whole duplication based task scheduling process is summarized in Algorithm 2 below.

ALGORITHM 2: HETEROGENEOUS DUPLICATION BASED SCHEDULING

Input: a program task graph with n tasks $\{T_1, T_2, \dots, T_n\}$, a heterogeneous system with m machines $\{P_1, P_2, \dots, P_m\}$, and the relative speeds of the machines;

Output: a duplication based schedule

1. Construct the scheduling list (use Algorithm 1);
2. For each CP task, first recursively schedule each of its unscheduled ancestor IB tasks to a machine so that they can finish as soon as possible by trying to duplicate on the machine as many ancestors as the time slot allows (use the heterogeneity factors h_{ix} for determining the finish times); the order of selecting tasks for duplication is governed by the scheduling list; finally apply the same recursive duplication process to the CP task itself;
3. Without using any duplication, schedule each of the remaining tasks (i.e., OB tasks) to the fastest machine provided that the schedule length does not increase; if this fails, employ recursive duplication

technique to schedule the OB task;

To illustrate how the heterogeneity of the machines is exploited, consider in Figure 4 the two schedules of the Gaussian elimination task graph (shown earlier in Figure 1). The schedule on the left is the best schedule without duplication using homogeneous machines. On the right is a schedule using six heterogeneous machines in which P_2 is of the same speed as the machines in the left schedule, while P_0 and P_1 are two times and 1.3 times faster than P_2 , respectively. The remaining machines are slower than P_2 . We can see that the CP of the task graph is scheduled to the fastest machine P_0 . The critical IB tasks, T_4 and T_5 , are also scheduled to finish as early as possible on fast machines P_1 and P_2 , respectively, by duplicating T_1 . The resulting schedule has an overall completion time of 182 units which is significantly smaller than that of the homogeneous schedule without duplication (330 units)[†]. Due to space limitations, detailed steps of producing the two schedules are not shown.

After a symbolic schedule is generated, the code generator is invoked to actually implement the schedule using the SPMD (Single Program Multiple Data) model [2], [23]. The program statements or procedures constituting a task T_i are allocated to the specified machine P_j for execution using conditional statements checking the ID of the machine, as shown in Figure 5. Data structures associated with a task are also replicated. The output of the code generator is a C program in which MPI communication primitives are inserted. The resulting parallel program is then compiled and executed on the cluster of workstations.

3 Performance Results

We have implemented the duplication based

scheduling algorithm in the code generator module of the CASCH tool (see Figure 3), which is executable on a Linux-based Pentium II PC in our cluster. We have parallelized several numerical applications on CASCH. Here, we present and discuss some preliminary results obtained by measuring the execution time of three applications: Gaussian elimination, Gauss-Seidel algorithm, and N-Body problem. By varying the problem sizes (i.e., the dimensions of the matrices in these applications, from 32 to 256) and the granularities (from 1-column block to 8-column block, using 1-D decomposition), we generated four task graphs for each application with roughly 100, 200, 400, and 800 tasks.

Our heterogeneous cluster consists of twelve PCs: eight Pentium II 333 MHz with 32 MB memory and four Pentium II 450 MHz with 64 MB memory. The PCs are connected by a Fast Ethernet switch. All the experiments were performed using eight PCs but with different configurations: (1) eight homogeneous machines (i.e., all are Pentium II 333 MHz); (2) five Pentium II 333 MHz plus two Pentium II 450 MHz; (3) two Pentium II 333 MHz plus four Pentium II 450 MHz. The aggregate computing power of the three configurations are approximately the same because we found that a Pentium II 450 MHz is about 1.5 times faster than a Pentium II 333 MHz. The rationale behind selecting these configurations is that we wanted to investigate the benefit of heterogeneity. These configurations are denoted as 8S (for eight slow machines), 2F+5S (two fast plus five slow machines), and 4F+2S (four fast plus two slow machines), respectively. Ten different runs for each size of the three applications were done and the average application execution times were noted.

These average execution times of the three applications are shown in Figure 6. As can be seen,

[†] In the homogeneous case, the scheduler is also given six machines. However, to arrive at the best schedule shown, it needs only three.

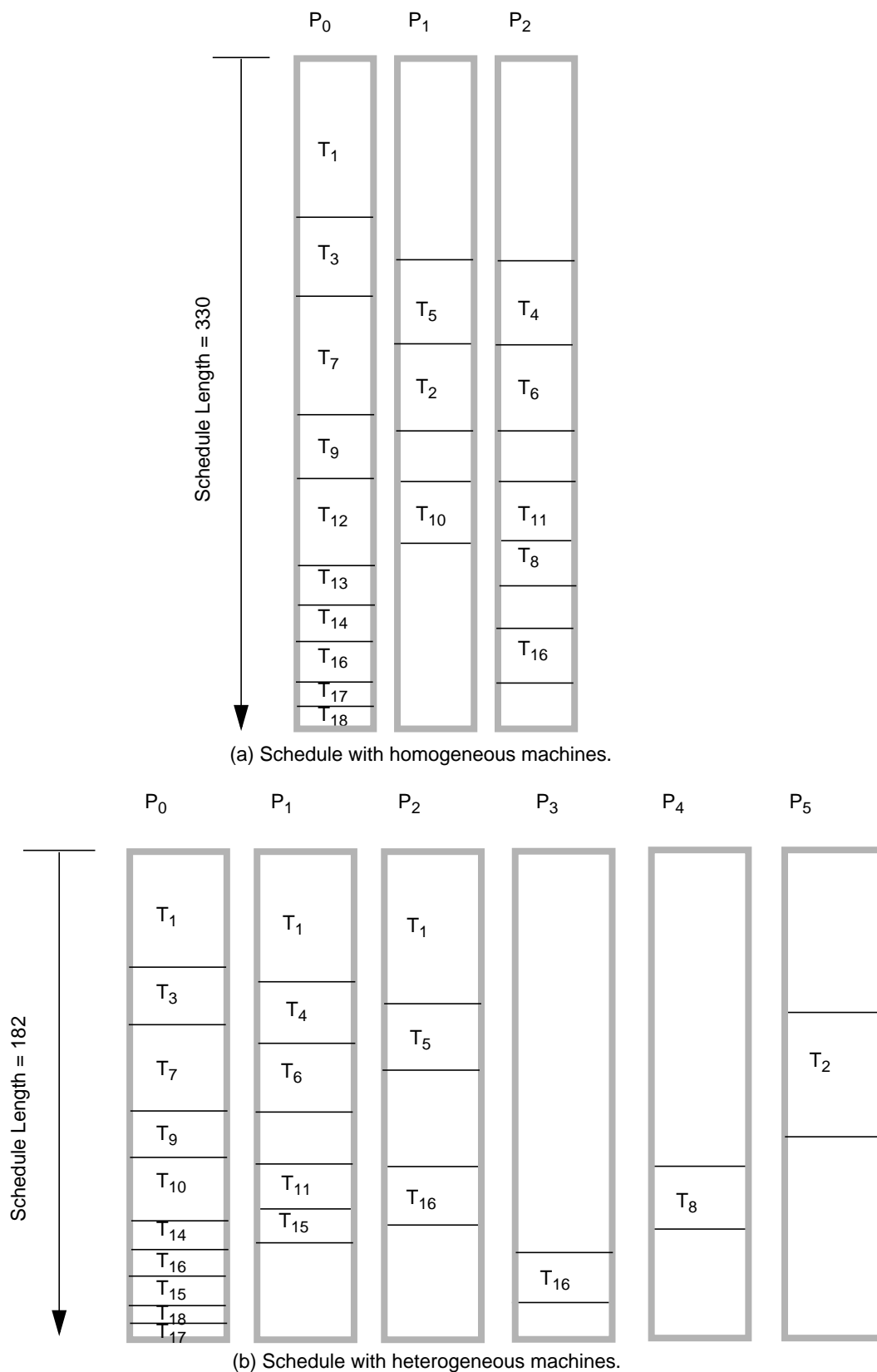
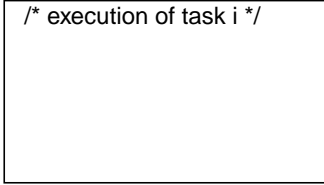


Figure 4: The effect of heterogeneity.


```

if (mynode() == j) {
    /* execution of task i */
    
}

```

Figure 5: SPMD implementation of a schedule.

heterogeneity has a significant impact on the overall execution time of an application in that using more fast machines (albeit the total number of machines is smaller), in general, can speedup the application considerably. The improvement in the Gaussian elimination application is the most remarkable. This can be explained by the fact that the Gaussian elimination graph has a distinctive critical-path (see Figure 2), the tasks on which can be scheduled to the fastest machine. On the other hand, as the Gauss-Seidel task graph has many intersecting critical-paths [23], the duplication approach is less effective in exploiting the advantage of heterogeneity. The improvement of the heterogeneous approaches for the N-Body problem, which has a slightly less regular task graph structure [23], is also considerable.

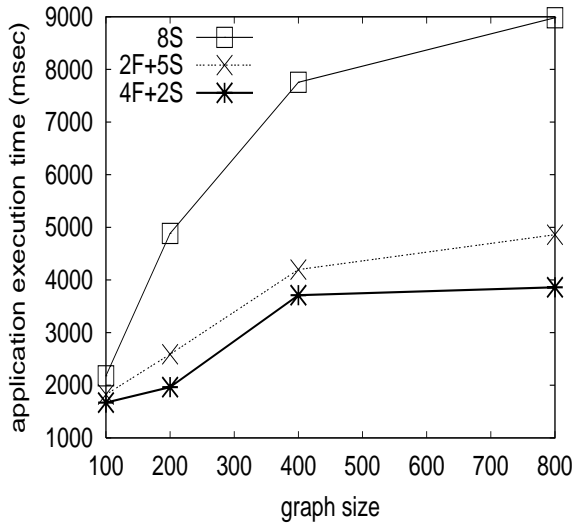
4 Conclusions and Future Work

In this paper, we have presented a duplication based approach in scheduling tasks to a heterogeneous cluster of PCs. The scheduling algorithm works by recursively duplicating critical tasks to the faster machines in order to minimize the finish times. The algorithm has been implemented in our prototype program parallelization tool for generating MPI code executable on a cluster of Pentium PCs. Our experiments using three numerical applications have indicated that heterogeneity of PCs cluster is indeed useful for optimizing the execution of parallel programs. One

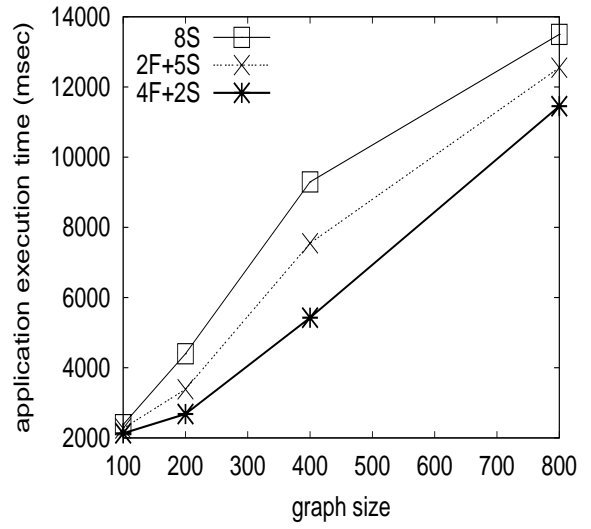
important issue related to using a PC cluster is fault-tolerance. Unlike a tightly couple parallel architecture (e.g., the IBM SP2), a PC in a cluster may experience intermittent failure, possibly due to user reboots. Thus, the task schedule has to be fault-tolerant so that the application can finish its execution even in the presence of such faults. We believe that task duplication, augmented with check-pointing and roll-back recovery techniques, is a viable approach to achieve this goal. A performance model is being developed to quantitatively analyze the merits of this approach.

References

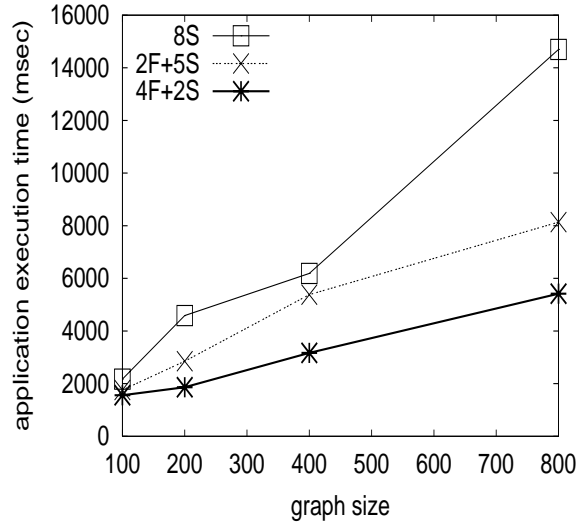
- [1] I. Ahmad and Y.-K. Kwok, "On Exploiting Task Duplication in Parallel Program Scheduling," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 9, pp. 872-892, Sept. 1998.
- [2] I. Ahmad, Y.-K. Kwok, M.-Y. Wu, and W. Shu, "CASCH: A Software Tool for Automatic Parallelization and Scheduling of Programs on Multiprocessors," *IEEE Concurrency*, accepted for publication and scheduled to appear in 2000.
- [3] M. Cosnard and M. Loi, "Automatic Task Graphs Generation Techniques," *Parallel Processing Letters*, vol. 5, no. 4, pp. 527-538, Dec. 1995.
- [4] M. Cosnard, M. Marrakchi, Y. Robert, and D. Trystam, "Parallel Gaussian Elimination on An MIMD Computer," *Parallel Computing*, vol. 6, pp. 275-296, 1988.
- [5] H. El-Rewini, T.G. Lewis, and H.H. Ali, *Task Scheduling in Parallel and Distributed Systems*, Englewood Cliffs, New Jersey: Prentice Hall, 1994.
- [6] T. Fahringer, "Compile-Time Estimation of Communication Costs for Data Parallel Programs," *J. Parallel and Distributed Computing*, vol. 39, pp. 46-65, 1996.
- [7] R.F. Freund and H.J. Siegel, "Heterogeneous Processing," *IEEE Computer*, vol. 26, no. 6, pp.



(a) Gaussian elimination



(b) Gauss-Seidel



(c) N-body

Figure 6: The average execution time of the three applications with three different cluster configurations.

13-17, June 1993.

- [8] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 1979.
- [9] K. Hwang, X. Zu, and C.L. Wang, "Viable Approaches to Realizing Single System Image

in Multicomputer Clusters," *IEEE Concurrency*, accepted for publication and to appear.

- [10] B. Kruatrachue and T.G. Lewis, "Grain Size Determination for Parallel Processing," *IEEE Software*, vol. 5, no. 1, pp. 23-32, Jan. 1988.
- [11] Y.-K. Kwok and I. Ahmad, "Dynamic Critical

- Path Scheduling: An Effective Technique for Allocating Tasks Graphs to Multiprocessors,” *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506-521, May 1996.
- [12]—, “Benchmarking and Comparison of the Task Graph Scheduling Algorithms,” *Journal of Parallel and Distributed Computing*, accepted for publication and scheduled to appear in 1999.
- [13]—, “Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors,” *ACM Computing Surveys*, accepted for publication and to appear in 2000.
- [14]M.G. Norman and P. Thanisch, “Models of Machines and Computation for Mapping in Multicomputers,” *ACM Computing Surveys*, vol. 25, no. 3, pp. 263-302, Sept. 1993.
- [15]M.A. Palis, J.-C. Liou, and D.S.L. Wei, “Task Clustering and Scheduling for Distributed Memory Parallel Architectures,” *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 1, pp. 46-55, Jan. 1996.
- [16]C. Papadimitriou and M. Yannakakis, “Toward an Architecture Independent Analysis of Parallel Algorithms,” *SIAM J. Computing*, vol. 19, no. 2, pp. 322-328, Apr. 1990.
- [17]G.-L. Park, B. Shirazi, and J. Marquis, “DFRN: A New Approach for Duplication Based Scheduling for Distributed Memory Multiprocessor Systems,” *Proc. 11th Int’l Parallel Processing Symposium*, pp. 157-166, Apr. 1997.
- [18]G.F. Pfister, *In Search of Clusters*, second edition, Englewood Cliffs, New Jersey: Prentice Hall, 1998.
- [19]V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, Cambridge, MA, 1989.
- [20]B. Shirazi, H. Chen, and J. Marquis, “Comparative Study of Task Duplication Static Scheduling versus Clustering and Non-Clustering Techniques,” *Concurrency: Practice and Experience*, vol. 7, no. 5, pp. 371-390, Aug. 1995.
- [21]H.J. Siegel, H.G. Dietz, and J.K. Antonio, “Software Support for Heterogeneous Computing,” *ACM Computing Surveys*, vol. 28, no. 1, pp. 237-239, Mar. 1996.
- [22]G.C. Sih and E.A. Lee, “A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures,” *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 2, pp. 75-87, Feb. 1993.
- [23]M.-Y. Wu and D.D. Gajski, “Hypertool: A Programming Aid for Message-Passing Systems,” *IEEE Trans. Parallel and Distributed Systems*, vol. 1, no. 3, pp. 330-343, July 1990.
- [24]T. Yang and A. Gerasoulis, “List Scheduling with and without Communication Delays,” *Parallel Computing*, vol. 19, no. 12, pp. 1321-1344, Dec. 1993.
- [25]A.Y. Zomaya, M. Clements, and S. Olariu, “A Framework for Reinforcement-Based Scheduling in Parallel Processor Systems,” *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 3, pp. 249-260, Mar. 1998.

Author Biography:

YU-KWONG KWOK received his BSc degree in computer engineering from the University of Hong Kong in 1991, the MPhil and PhD degrees in computer science from the Hong Kong University of Science and Technology in 1994 and 1997, respectively. Currently, he is an assistant professor in the Department of Electrical and Electronic Engineering at the University of Hong Kong. Before joining the University of Hong Kong, he was a visiting scholar for one year in the parallel processing laboratory at the School of Electrical and Computer Engineering at Purdue University. His research interests include mobile computing, wireless networking, software support for parallel and distributed computing, heterogeneous cluster computing, and distributed multimedia systems. He is a member of the IEEE Computer Society and the ACM. For more information about Kwok, visit <http://www.eee.hku.hk/~ykwok>.