JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **46,** 125–135 (1997)
ARTICLE NO. PC971378

Coordinating Parallel Processes on Networks of Workstations[1]

Xing Du and Xiaodong Zhang
Department of Computer Science, College of William and Mary, Williamsburg, Virginia 23187

The network of workstations (NOW) we consider for scheduling is heterogeneous and nondedicated, where computing power varies among the workstations and local and parallel jobs may interact with each other in execution. An effective NOW scheduling scheme needs sufficient information about system heterogeneity and job interactions. We use the measured power weight of each workstation to quantify the differences of computing capability in the system. Without a processing power usage agreement between parallel jobs and local user jobs in a workstation, job interactions are unpredictable, and performance of either type of jobs may not be guaranteed. Using the quantified and deterministic system information, we design a scheduling scheme called *self-coordinated local scheduling* on a heterogeneous NOW. Based on a power usage agreement between local and parallel jobs, this scheme coordinates parallel processes independently in each workstation based on the coscheduling principle. We discuss its implementation on Unix System V Release 4 (SVR4). Our simulation results on a heterogeneous NOW show the effectiveness of the self-coordinated local scheduling scheme. © 1997 Academic Press

1. INTRODUCTION

Networks of workstations (NOWs) have become important and cost-effective parallel platforms for scientific computations. In practice, a NOW system is heterogeneous and nondedicated. These two unique factors make scheduling policies on multiprocessor/multicomputer systems not suitable for NOWs. Since the nature of parallel processing on NOWs does not change, the coscheduling [13] principle is still an important basis for parallel process scheduling on NOWs. Thus, heterogeneity, job interactions, and coscheduling are three major concerns in our design.

Many research groups currently are using homogeneous NOWs as experimental platforms. In practice, more and more heterogeneous NOW systems are being used for parallel computing. The most common heterogeneous NOW would be a network of the same type of workstations but with differential computational capabilities.

Regarding the issue of job interactions, there are two basic approaches: avoid interactions by migrating one type of job to a dedicated environment, or go along with interactions but try to effectively schedule jobs. The studies in [5 and 12] indicate that more than 50% of workstations are idle at any time. However, in practice, it may be difficult to immediately find a set of suitable workstations to migrate the target processes to where they can stay for awhile and need not be migrated to other workstations frequently. A main reason for this is because local users' behaviors are unpredictable. Context switching and migration overheads are other factors that may offset the performance gain from process migration. In addition, the NOW utilization could be low without an interaction of multiple jobs. We focus our research on the approach to keeping both local and parallel jobs together and effectively scheduling them.

Coscheduling is another available technique for parallel process scheduling. It generally results in good parallel program performance and is widely used to schedule parallel processes involving frequent communication and synchronization [4, 8, 10]. This method is particularly effective for parallel applications partitioned into multiple processes of equal size, running on a homogeneous multiprocessor/multicomputer system. Coscheduling will ensure that no process will wait for a nonscheduled process for synchronization/communication and will minimize the waiting time at the synchronization points. However, the nondedicated feature of workstations and the low communication bandwidth make the implementation of complete coscheduling on NOWs expensive and not realistic [7].

Using quantified and deterministic system information such as a power weight, and the power preservation in each workstation, we address the three NOW scheduling issues by designing a scheduling scheme called *self-coordinated local scheduling*. This scheme coordinates parallel processes independently in each workstation based on the coscheduling principle.

Our scheduling design has the following features. First, each local scheduler adjusts its execution pace of a parallel process. The coordination of parallel processes is performed independently at each workstation and is able to obtain coscheduling performance results without a global scheduler. Second, applying power preservation at each workstation would be fair to both local user jobs and parallel jobs.

[1]This work is supported in part by the National Science Foundation under Grants CCR-9102854 and CCR-9400719, by the Air Force Office of Scientific Research under Grant AFOSR-95-1-0215, and by the Office of Naval Research under Grant ONR-95-1-1239.

0743-7315/97 $25.00
Copyright © 1997 by Academic Press
All rights of reproduction in any form reserved.

The job response times of both types would be acceptable and predictable. Third, the NOW utilization is increased by executing both local user and parallel jobs together without process migrations. Finally, it is easy to use facilities of standard Unix operating systems such as SVR4 to implement this scheduling scheme.

2. MODELS OF NOW AND PARALLEL PROGRAMS

In comparison with multiprocessor/multicomputer systems, a NOW has two unique features: heterogeneity and nondedication. In this section, we quantify the processing capability of a workstation and discuss a representative parallel programming model.

2.1. Workstation Power Weight

A NOW can be abstracted as a connected graph *HN*(*M*, *Net*), where

- $M = \{M_1, M_2, \ldots, M_m\}$ is a set of workstations (m is the number of workstations).
- Net is a standard interconnection network, such as an Ethernet or an ATM network.

If a NOW consists of a set of identical workstations, the system is homogeneous. Otherwise, a heterogeneous NOW is formed. Different workstations may have different architectures (architectural heterogeneity) and/or different computing capability. Here we only consider the computing capability differences among workstations.

We use the power weight to refer to a workstation's computing capability relative to the fastest workstation in a NOW. The value of the power weight is less than or equal to 1. Since the power weight is a relative ratio, it can also be represented by measured execution time. If $T(App, M_i)$ gives the execution time for completing application program, App, on a dedicated workstation M_i, the power weight of workstation i can be calculated as

$$W_i(App) = \frac{\min_{i=1}^{m}\{T(App, M_i)\}}{T(App, M_i)}. \tag{2.1}$$

Equation (2.1) indicates that the power weight of a workstation is application program dependent. However, our experiments [15] show that this is mainly related to the problem size. If the memory size is large enough to hold a process and its data, the differences among measured power weights using different application programs on a workstation are insignificant. The power weight of a workstation is determined in this paper by averaging a group of power weights from different application programs which have reasonable memory allocations.

2.2. Bulk Synchronous Parallel Model

A typical message-passing NOW parallel program is coarse or medium grained and consists of one process per workstation on a fixed number of workstations throughout execution. The size of each process is generally similar. The process completes after a number of iterations. In each iteration, phases of local computation alternates with phases of communications and synchronizations. This programming model is called Bulk Synchronous Parallel model (BSP). The basic structure of the model is as follows:

```
Loop

  simultaneous tasks for local computation;
  communications for data exchange or
    synchronization for critical sections;
  barrier;

end Loop
```

We study the scheduling of this type of applications, and assume that no process migration occurs during the execution of applications.

3. SELF-COORDINATED LOCAL SCHEDULING

3.1. Rationale of Power Preservation

In order to design an effective scheduling scheme for both parallel and local jobs on NOWs, two issues must be well addressed: how to coordinate the simultaneous execution of processes of a parallel job, and how to manage the interaction between parallel jobs and local user jobs. We address these two issues together by explicitly dividing the computing power of a workstation into two parts: one preserved for running local jobs, and the other preserved for parallel ones. This voluntary power division is reasonable based on two facts:

- A networked workstation is no longer a private resource of its local user. Rather, it is often shared by other users and jobs.
- The user of a parallel job may be the local user of a workstation and needs distributed resources.

In addition, the explicit power preservation has two advantages:

- The performance of both local and parallel jobs is guaranteed. When they coexist, a certain amount of power is guaranteed to each of them. Meanwhile, parallel jobs and local jobs can use each other's free cycles if they are available.
- On a heterogeneous NOW, proper power preservation based on power weight in each workstation may ensure that parallel processes across workstations proceed at a certain pace without global coordination. If they are started simultaneously, coscheduling can be achieved as if they ran on a virtual "homogeneous" NOW.

We call this scheme *self-coordinated local scheduling*.

3.2. Power Preservation for Parallel Jobs

A key issue of the scheme is to preserve a proper portion of the power in each workstation for parallel jobs. In particular, each workstation's power is used to serve three types of jobs: the operating system kernel processes, local jobs, and parallel jobs. Since the power used for kernel processes is small compared with that used for the other two types, for simplicity of discussion, we only consider the power used for local and parallel workloads and include the kernel in the local workload.

We assume that initially each local user of a workstation specifies a ratio at which the power is divided between local and parallel workloads. Assume $R_{\text{user}}(i)$ is the percentage of power used for local user jobs in the ith workstation, and W_i is the power weight of the ith workstation; thus, the available power weight for parallel jobs in the ith workstation, denoted by ρ_i, is

$$\rho_i = W_i(1 - R_{\text{user}}(i)). \tag{3.2}$$

In practice, because the task size of a parallel process on each workstation is normally similar, the bottleneck of a parallel job execution is the relatively slowest workstation which offers the smallest power weight in the NOW. Thus, the preserved power weight for parallel jobs in each workstation is determined by the minimum available power weight for parallel jobs among all the workstations

$$\rho = \min_{i=1}^{m} \rho_i, \tag{3.3}$$

where m is the number of workstations in the system. The self-coordinated local scheduling scheme on a heterogeneous NOW can be outlined as follows:

1. Determine ρ, the available power weight for parallel jobs in a NOW.
2. Start the execution of parallel processes on workstations simultaneously.
3. The local scheduler in workstation i allocates ρ/W_i of its power to its parallel process.

Using this scheme, the same amount of processing power would be used by each workstation for executing its parallel process. The equivalent preserved power in each workstation for a parallel job simulates the execution of that job in a dedicated virtual homogeneous system. On the other hand, because the processing power in ith workstation used for local workloads is $W_i - \rho$, which is equal to or greater than what the local user expects ($W_i \times R_{\text{user}}(i)$), the performance of local jobs is guaranteed.

We show how a coordination is achieved as follows. Recall the BSP model discussed in Section 2: if the local scheduler can ensure that all computations finish within a reasonable time, all possible communication and synchronization activities following the computation will be coordinated. Here we temporarily use an application program dependent parameter for the local scheduler, the size of computations, denoted by *Size*, which is measured by the number of floating point operations. For a given power of a workstation measured by the number of floating point operations per second, the time to finish a computation on the relatively slowest workstation s is

$$t_s = \frac{Size}{Pow(s) \times (1 - R_{\text{user}}(s))}, \tag{3.4}$$

where $Pow(s)$ is the power of the slowest workstation; and the time to finish a computation on workstation i is

$$t_i = \frac{Size}{Pow(i) \times (1 - R_{\text{user}}(i))}, \tag{3.5}$$

where $Pow(i)$ is the power of workstation i ($i = 1, \ldots, m$, and $i \neq s$). If a time slice based scheme is used in the local scheduler of each workstation, the number of time slices needed to finish the computation on the slowest workstation is

$$N_{\text{slice}}(s) = \frac{t_s}{\delta_s}, \tag{3.6}$$

where δ_s is the length of a time slice in the slowest workstation; and the number of time slices used to finish a computation of the same size on workstation i is

$$N_{\text{slice}}(i) = \frac{t_i}{\delta_i}, \tag{3.7}$$

where δ_i is the length of a time slice in workstation i, and $i = 1, \ldots, m$. Since the slowest workstation is the bottleneck of parallel jobs, if all the computations in other workstations finish within the time t_s in each loop, the performance would be optimal and equivalent to that in a dedicated NOW using coscheduling. However, within t_s, there are t_s/δ_i time slices available in workstation i, which is larger than $N_{\text{slice}}(i)$. This means that all workstations except the slowest one have extra time slices, which may be used by local processes. Because workstation s is the bottleneck, it finishes the parallel process at last by t_s. It is not necessary for the process on workstation i to be finished before t_s. We may evenly distribute these $N_{\text{slice}}(i)$ time slices for the computation of its parallel process over t_s/δ_i. Therefore, if one time slice is guaranteed to be given to the parallel process in workstation i within $t_s/(\delta_i N_{\text{slice}}(i)) = t_s/t_i$ time slices periodically, it is ensured that by the end of t_s, the computation of its parallel process will finish. Consequently, the computation of each parallel process across the NOW is finished neither too fast nor too slow, just before t_s. Coordination is achieved.

By (3.4) and (3.5), we further obtain the execution pace in each workstation for parallel processes:

$$\frac{t_s}{t_i} = \frac{Pow(i) \times (1 - R_{\text{user}}(i))}{Pow(s) \times (1 - R_{\text{user}}(s))} = \frac{\rho_i}{\rho}. \tag{3.8}$$

The execution pace in (3.8) can be self-determined, and is workstation computing power dependent rather than application program dependent.

TABLE I
Process Priority Distributions

Class	Ranks	Management
RT	100—159	fixed
SYS	60—99	fixed
TS	0—59	dynamic

4. POWER PRESERVATION ON SVR4

Unix System V Release 4 (SVR4) [9] is a powerful and open operating system. We select SVR4 as the target operating system to discuss how to preserve power and implement the scheme in a commodity operating system.

The scheduling policy of SVR4 is time-sharing and priority-based. The processes are classified into four classes: system (SYS), real-time (RT), time-sharing (TS), and interactive (IA). Under each class, a scheduling policy is defined. Because TS and IA share the same policy, there are actually three policies/ classes in SVR4. Associated with each class is a contiguous set of priority ranks of integer numbers, which is shown in Table I.

The priority ranks of SYS and RT processes are fixed during their lifetimes, while those of TS are changed dynamically for the sake of fairness and efficiency. Another table in SVR4, called the *time-sharing dispatcher parameter table* (ts_dptbl) describes TS priority changes. Table II gives an example of it. There are five columns in ts_dptbl: `quantum`, `tqexp`, `slpret`, `maxwait`, and `lwait`. Column `rank` is the comment column and lists the priority ranks. Each line describes how the priority of a process at this rank changes with time and events. `Quantum` specifies the length of the time slice allocated to processes at this rank. If it is used up, `tqexp` usually gives a lower priority rank and a larger time quantum to this process. `Slpret` is the priority rank of a process after it returns from sleeping. Usually, a process returned from "sleeping" is assigned a higher priority rank than its original rank. If a process has waited for the CPU awhile (longer than `maxwait`), a higher priority rank, given in `lwait`, is assigned to this process to prevent it from starvation.

Processes of RT and SYS types should be executed more urgently than TS processes and parallel jobs, and they are usually given higher fixed priorities. To preserve power in each workstation using SVR4, we focus on how to allocate computing power among traditional TS processes and parallel jobs.

TABLE II
Dispatcher Parameter Table

Quantum	tqexp	slpret	maxwait	lwait	rank
200	0	50	0	50	0
.	.	.	.	.	.
20	49	59	32000	59	59

TABLE III
Priority Rank Redistribution

Class	Ranks	Management
RT	100 + N—159 + N	fixed
SYS	60 + N—99 + N	fixed
PJ1	60—59 + N	dynamic
TS	0—59	dynamic
PJ0	$-N$— -1	dynamic

We first extend the process classification of SVR4 by adding a new class called parallel job (PJ). All processes of parallel jobs are of this class. The priority ranks as numbers are extended to $\pm N$, where N is a given positive integer number ($N \geq 1$). The priority ranks of PJ have two separate sets of contiguous numbers. The lower set, defined as PJ0, ranges from $-N$ to -1, and the higher set, defined as PJ1, ranges from 60 to $59 + N$. The new priority distribution is shown in Table III. As listed in Table III, PJ has two types of processes: those within PJ1 and those within PJ0. The priority ranks of PJ1 processes are higher than that of any TS and PJ0 processes, while the priority ranks of PJ0 processes are lower than that of any TS and PJ1 processes.

The priority ranks of PJ processes could be changed dynamically based on a revised ts_dptbl table, called the *time-sharing and parallel job dispatcher parameter table* (tspj_dptbl). This table is a direct extension to the original SVR4's ts_dptbl, where new lines are added to represent new priority ranks introduced by class PJ. Another difference between ts_dptbl and tspj_dptbl is the time unit for column `maxwait`. In SVR4, the unit is one second. In contrast, we use a smaller unit (ms) as the unit of waiting time. Table IV gives an example of tspj_dptbl with $N = 1$.

The changes of priority ranks for PJ processes deployed in tspj_dptbl follow the two rules:

- If the priority rank of a process, defined as a quantitative value, pri, is within class PJ0, and the process has waited CPU for awhile (`maxwait`), its priority rank is increased to class PJ1, by setting the priority to $pri + 61$;

TABLE IV
New Dispatcher Parameter Table

Quantum	tqexp	slpret	maxwait	lwait	rank
500	-1	-1	150	60	-1 (PJ0)
200	0	50	1000	50	0
.	.	.	.	.	.
20	49	59	1000	59	59
50	-1	60	200	60	60 (PJ1)

- If the priority, pri, of a process is within class PJ1 and has finished its time quantum (`quantum`), its priority rank is decreased to class PJ0, by setting the priority to $pri - 61$.

If workstation i preserves ρ / W_i power weight for a parallel process, where ρ is the preserved power in the system, and W_i is the power weight of workstation i, this preservation can be done by setting up the following parameters in table tspj_dptbl. We denote the minimum time quantum for processes of all priority ranks as Min_q. The power preservation for that process can be achieved by assigning the process's priority rank initially to −1, and assigning rank 60's (PJ1) `quantum`:

$$quantum = Min_q; \tag{4.9}$$

and assigning rank −1's (PJ0) `maxwait`:

$$maxwait = (W_i/\rho - 1) \times Min_q. \tag{4.10}$$

By loading the tspj_dptbl table (replacing the original ts_dptbl) at boot time, we may implement the scheme easily in SVR4.

5. SIMULATION AND PERFORMANCE EVALUATION

5.1. Simulator

To evaluate and analyze the performance of parallel and local jobs under the self-coordinated scheduling scheme, we designed and developed a simulator to perform event-driven simulation, where NOW parameters, local job events, parallel job events, and scheduling policies are input, and simulated execution times and overhead times, such as context switch, are output.

We selected 7 types of Sun SPARCstations with different computing powers. The 7 types cover a large range of processing capabilities of Sun workstations with a single processor. We measured the power weight of each type by running 4 NAS benchmark applications which will be discussed later. All the measurements were repeated 10 times in a dedicated environment. The power weight for one workstation was finally calculated by averaging all the power weights measured by the four applications. Table V gives the average power weights of the 7 types of workstations, which were used in the simulator to simulate the heterogeneity of workstations. We give each type a letter such as A or B for simple reference. A network was used to connect workstations. The context switch cost was assumed as 200 μs.

We selected four programs from the NAS parallel benchmarks [3]: EP (Embarrassing Parallel), MG (Multigrid), IS (Integer Sort), and LU (LU Decomposition) as the example applications. All of them followed the BSP model. However, the four applications were different in the computation size at each iteration and the communication patterns. The parallel job events were characterized by their computation patterns, the sizes of each computation, communication patterns and amount, the number of iterations, process arrival times, and the number of processes. Our simulator was driven by the events characterized from the four parallel applications.

In the experiments, two kinds of local workloads were taken into account. The first type only consumed CPU cycles for computations. This type of local job event was simply characterized by starting times and execution lengths. The second was a more realistic type where computations and system calls were interleaved. The priority of a process may be changed when it invokes system calls; thus, system calls affect the performance of the whole system. For this type of local job event, the distribution of the system calls in the execution times of the process was an additional parameter that was considered. We assumed the system calls were exponentially distributed in the lifetime of each local process.

The scheduling policies in the simulator included the SVR4 local scheduling, coscheduling using the matrix scheme in [13], and the self-coordinated local scheduling scheme based on SVR4 which is discussed in Section 4.

5.2. Precision of Power Preservation

Using the simulator, we first studied the effects of the `quantum` in PJ1 and `maxwait` in PJ0 on the precision of power preservation of a workstation.

We first give the definition of the precision of power preservation. Let T be the execution time of a parallel process on a dedicated workstation. If δ portion of the power of a workstation is preserved for the process, and the process finishes in T' time, then the power preservation precision is defined by

$$\left(1 - \left|1 - \frac{\delta T'}{T}\right|\right) \times 100\%.$$

In these experiments, three local processes, L_1, L_2, L_3, and one parallel process P were executed in a workstation. We preserved 1/3 power for process P. The size of P was set to 2000, 1000, and 200 ms, while the sizes of three local processes were set long enough to be finished after P terminated.

TABLE V
The Average Power Weights of 7 Types of Sun Workstations

S20-HS21	S20-HS11	S5-85	S20-50	S5-70	S10-30	Classics
A = 1.0	B = 0.790	C = 0.562	D = 0.461	E = 0.436	F = 0.374	G = 0.239

The values in the default SVR4 ts_dptbl were used for local processes in table tspj_dptbl. The initial priority of P was assigned to PJ0. The ratio of `maxwait` in PJ0 to `quantum` in PJ1 was 2 based on the power preservation rules (4.9) and (4.10). We observed the precision of the power preservation changed not only as the time quantum for P (the `quantum` in PJ1) changed but also as the computation size of P changed. (The quantum of `maxwait` in PJ0 is dependent on `quantum` in PJ1). Figure 1 reports from top to bottom the precision variance as the `quantum` of PJ1 changed when the computation size of P is 2000, 1000, and 200 ms, respectively. In each curve, generally, the precision decreases with the increase of `quantum`. In the top curve, when `quantum` varies from 5 to 400 ms, the power preservation precision remains approximately at 95%, which is quite precise. However, in the middle curve, when `quantum` approaches 300 ms, the precision decreases to nearly 88%. The worst case occurs in the bottom curve where the computation size of P becomes 200 ms. In this case, only those values of `quantum` which are less than 60 ms can provide the precision within 90%. The precision reduces to 60% when `quantum` equals 400 ms. This can be explained as follows. For a given computation, the size of `quantum` determines the number of slices the computation is divided into. The more slices a computation is divided into, or the larger the ratio between the computation time to `quantum`, the more precise the power preservation.

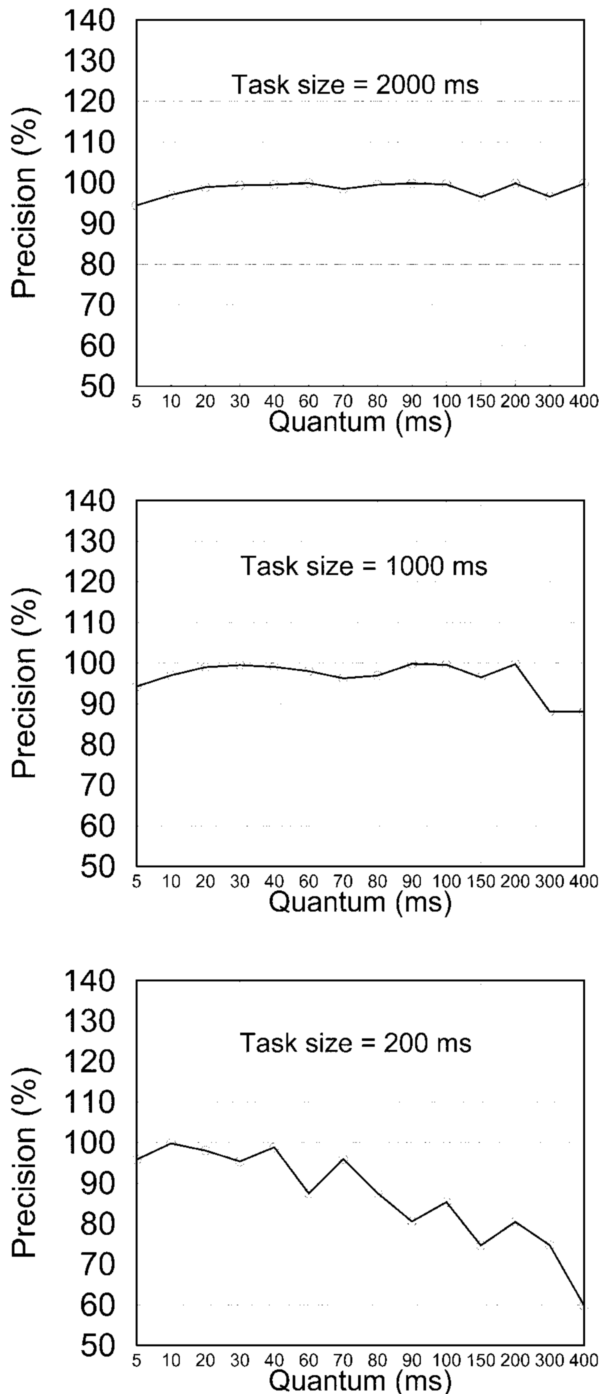

FIG. 1. The power preservation precision changes at different time quantums.

5.3. Power Preservation Effects on Local Processes

Preserving power for one process will affect the performance of other processes. The more it is preserved, the greater is the effect on other processes. We quantitatively evaluate this effect in this section.

We executed four processes L_1, L_2, L_3, and P, with the same computation size (4000 ms) in a workstation. We preserved power for the process P only and evaluated its effect on the performance of three other processes. We set `quantum` of PJ1 to 40 ms in order to preserve power varying from 1/3, 1/2, 2/3, and 3/4 for P, respectively. Figure 2 shows the slowdown factors of the four processes, which are relative to the execution time of these processes scheduled by the SVR4 scheduler.

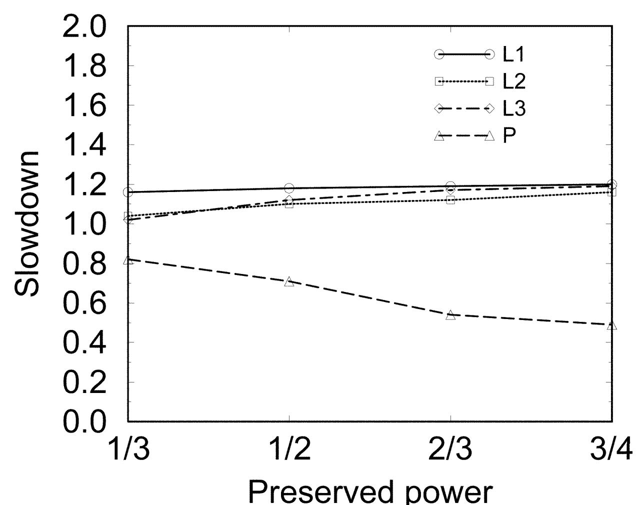

FIG. 2. The power preservation effects on local processes.

When 1/3 power was preserved for P, L_1 increased its execution time by 10%, L_2 and L_3 had only about 2% increase (L_1 had twice more system calls than L_2 and L_3). The execution time of P decreased by 18%. When more power was preserved, the execution times of L_1, L_2, and L_3 increased slightly, but P decreased quickly. For example, the 3/4 power preservation makes the execution time of P decrease by 51%, and that of L_1, L_2, and L_3 increase by 20, 16, and 19%, respectively.

These experiments show that the power preservation slightly affects the performance of the other processes, but significantly improves the performance of the process for which the power is preserved. Thus, if we preserve some power for parallel processes, the performance of local processes may decrease as the local user expects (since the user allows the power to be used by parallel processes), but not too much. Their response time will not increase greatly.

In the experiments, we also evaluated the context switch overhead. Because we fixed `quantum` to be 40 ms, and increased the preserved power, the number of context switch increased. Compared with the SVR4 scheduler, the context switch cost increased by 47% to 88% when the preserved power changed from 1/3 to 3/4. However, the actual context switch times were only 50.8 and 65.2 ms for 1/3 and 3/4 power preservation, respectively, which accounted for about 1.3 and 1.6% of the computation time (4000 ms).

5.4. Self-Coordinated Local Scheduling for Parallel Jobs

We next studied the performance of parallel jobs scheduled by the self-coordinated scheme on NOWs. We evaluated three factors affecting the performance of parallel jobs:

1. the number of local jobs,
2. different NOW systems, and
3. local jobs' system calls.

We did extensive simulation to evaluate the above effects. Because of space limit, we only report some major results in the paper. We used 8 workstations of type A, B, C, D, E, F and G to form two different NOW systems

NOW1: A+B+2C+2D+2E,
NOW2: A+B+C+2D+E+F+G.

We preserved $\rho = 0.2$ power weight for both NOW1 and NOW2 to run parallel jobs. Since the fastest workstations in both systems were identical (both are A), the preserved powers were also identical (this will be verified by the following experiments on NOW1 and NOW2). Workstations A, B, C, D, E, F and G preserved ρ / W_i of its total power for parallel jobs, that was, 20, 25, 36, 43, 46, 54, and 84%, respectively. We set 40 ms to `quantum` of PJ1.

The four parallel program events were used. The computation sizes in the following discussions are represented by the execution times on the fastest workstations. The execution times on other workstations are determined correspondingly based on their power weights. Two kinds of local processes were considered: those with system calls and those without system calls. The system calls were assumed to be exponentially distributed and occurred at the rate of about 40 times in 4000 ms. In the experiments, we also changed the number of local processes from 1 to 8 to see its effect on the performance of parallel jobs.

Figures 3, 4, 5, and 6 report the simulated execution times of EP, MG, IS, and LU on NOW1 (left) and NOW2 (right), respectively. The execution times are broken down into computation, communication, synchronization, and context switch, which are calculated by averaging their timing values across all workstations. For a given number of local processes, we also evaluated those with system calls (left bar) and those without system calls (right bar).

The EP program was composed of one computation followed by all-to-one communication. Figure 3 gives the execution times of an EP with its computation size of 4000 ms

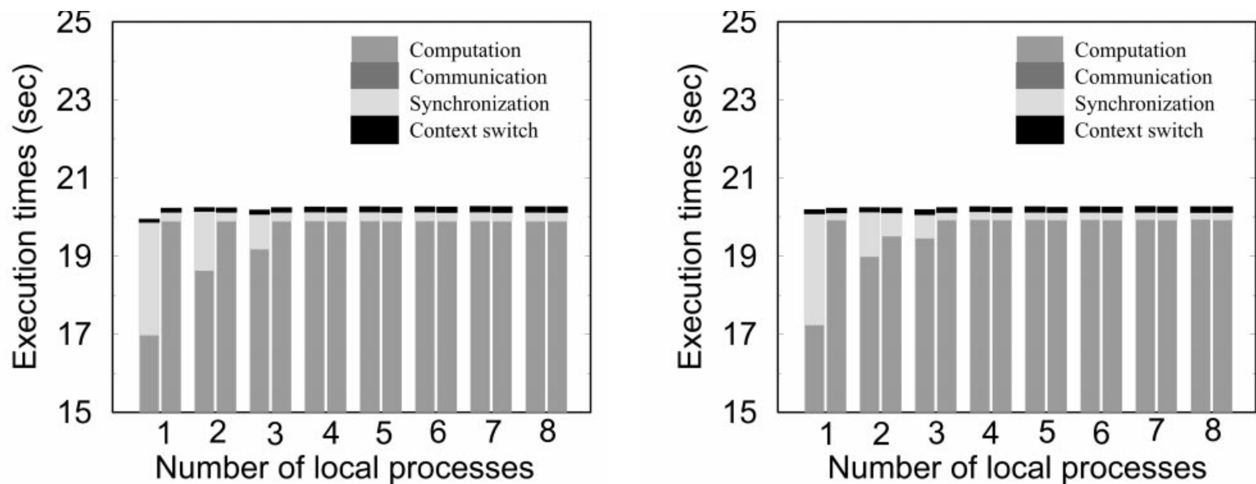

FIG. 3. EP on NOW1 (left) and NOW2 (right).

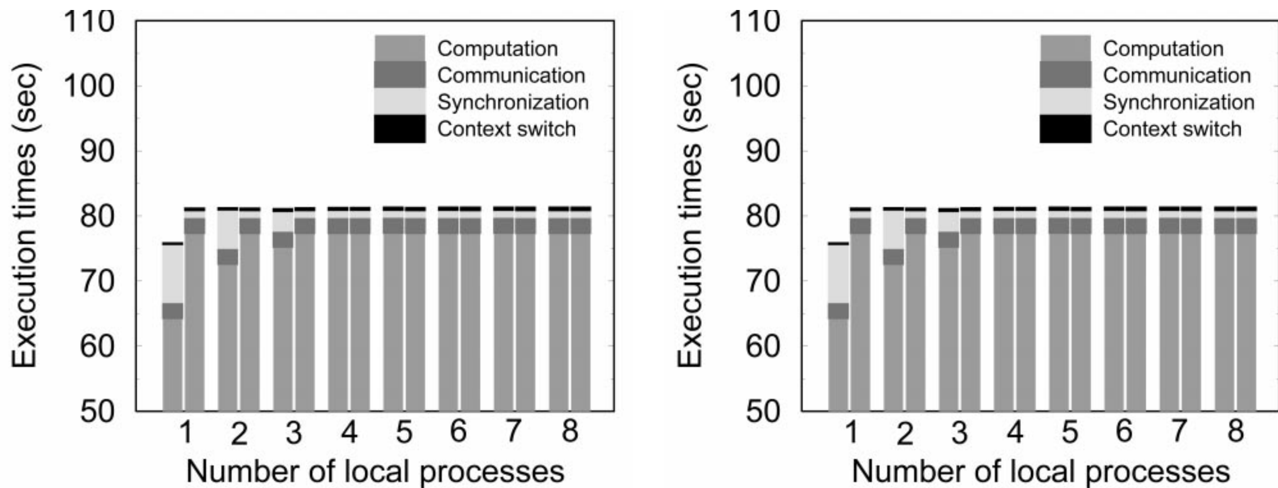

FIG. 4. MG on NOW1 (left) and NOW2 (right).

and communication amount of 1000 bytes. We first examine the effects of local processes with or without system calls on NOW1. When there was only one local process, the computation time of EP was reduced by 15% when there were system calls. This is because system calls may sometimes block the local process, thus more CPU cycles than preserved may be used by EP. However, when more than one local process existed, the difference between these two kinds of local workloads (with or without system calls) became insignificant. The overall performance (execution time) was also affected by this factor, but not much. Synchronization time is determined by the balance of the execution times of the computation because the fast processes have to wait for slow processes to go through. The synchronization waiting time decreased from 2881 to 223 ms when the number of local processes increased from 1 to 8. The execution time remained the same (20250 ms). This is because the earlier a process finishes, the more waiting times it will spend for the barrier. Context switch overhead contributed about 0.6% (121 ms) to the overall execution time. Running EP on a different NOW (NOW2) resulted in very similar performance.

All other three types of parallel programs performed well under the self-coordinated scheduling scheme as well. Figure 4 presents the performance of MG which was characterized by 9 computations separated by transpose communications. The computation sizes were 4000, 2000, 1000, 500, 250, 500, 1000, 2000, and 4000 ms. A total amount of 17 Mbytes of messages were sent from point to point. In the IS program, there were mainly three computations of 1000, 4000, and 2000 ms, respectively. The communication pattern was all-to-all, and the point-to-point communication was 10 Mbytes. Figure 5 presents the result of this application. The LU application in Fig. 6 was characterized by six computations with lengths of 8000, 4000, 2000, 1000, 500, and 250 ms, respectively, and neighbor communications transmitting 700 Kbytes of messages to its neighbor processes.

In summary, we find from the simulation that (1) the number of local jobs do not affect the parallel jobs significantly. In

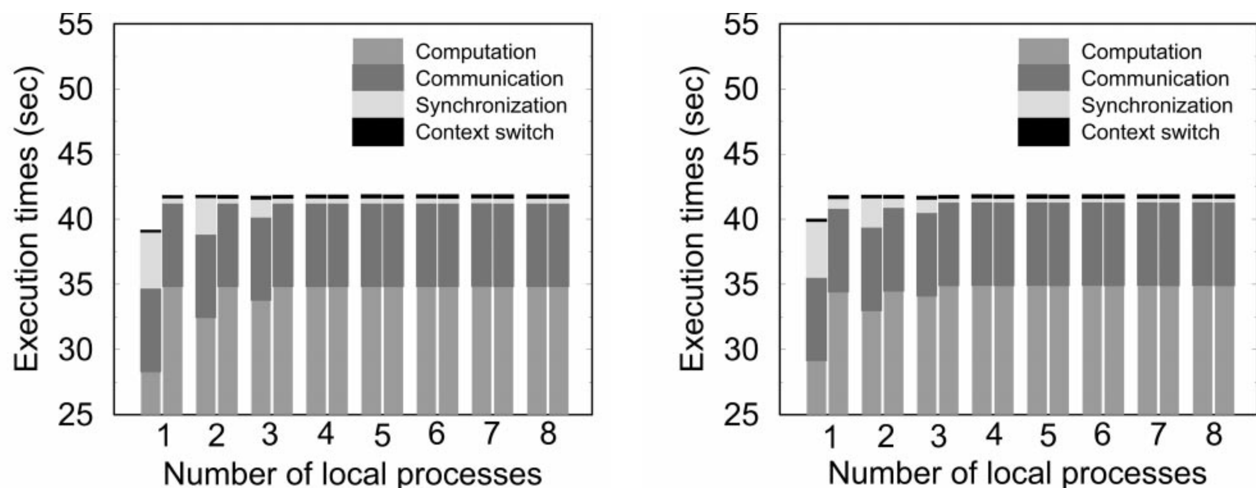

FIG. 5. IS on NOW1 (left) and NOW2 (right).

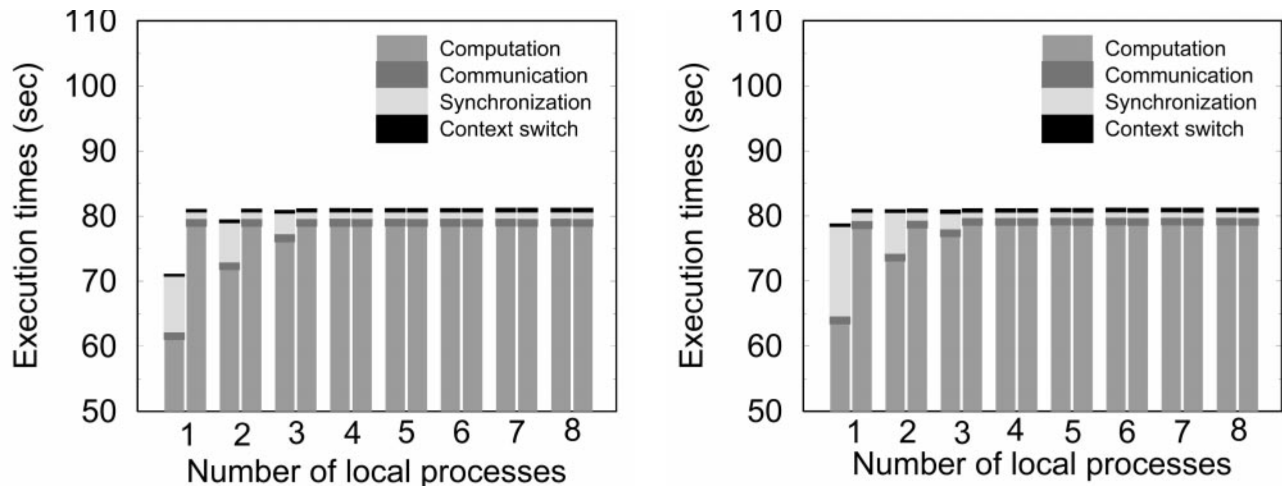

FIG. 6. LU on NOW1 (left) and NOW2 (right).

other words, the performance of parallel jobs is guaranteed no matter whether workstations are lightly or heavily loaded by local jobs. (2) Different combinations of workstations do not affect the execution performance of parallel jobs if the preserved powers are the same, and the fastest workstations are identical. (3) The behaviors of local jobs (with or without system calls) affect very slightly the effectiveness of the scheme.

Finally, we compared the performance of the four programs using self-coordinated scheduling with that using coscheduling. The execution times of the EP and IS programs using self-coordinated local scheduling are very close to that using coscheduling, differing by a factor of only 1.2%. The execution times of MG and LU using the self-coordinated scheme increased about 1.7% in comparison with the times of coscheduling. The slowdown comes from context switch overhead and power preservation precision problem we discussed in the previous subsection.

6. RELATED WORK

Using direct simulation, Arpaci *et al.* [1] evaluate effects of interactions between parallel jobs and local user jobs. They also study feasibilities of process migrations in order to avoid the interaction. Another proposal for the interaction is discussed in [11], which allows parallel jobs to stay in a workstation, but run at the lowest priority when local user jobs exist. This method avoids the process migration, but does not guarantee the performance of parallel jobs.

Several variations of coscheduling have been proposed to reduce the overhead of coscheduling. For example, Sobalvarro and Weihl [14] propose a demand-based coscheduling policy to schedule parallel processes simultaneously only if they communicate. There are two options: dynamic coscheduling and predictive coscheduling. When parallel processes are dynamically coscheduled, an arriving message will cause the targeted process at that node to start running. Predicative coscheduling uses the recent history of communications among processes to predict coming communication activities of each process. When a process is scheduled on one node, an attempt is made to schedule its correspondents on other nodes for simultaneous execution.

Dusseau *et al.* [7] address the scheduling issue from another perspective. In their study, they find that local scheduling is a feasible alternative to coscheduling for parallel applications with barrier synchronization. They propose a blocking algorithm, called the two-phased fixed-spin policy to avoid the unnecessary context switching and to increase the possibility of running several parallel processes simultaneously at their synchronization points. Coscheduling efforts are made by spinning at the blocking point for a specific period of time through using a revised SVR4-based local scheduler.

Atallah *et al.* [2] redefine coscheduling from an effective speedup perspective. In their study, they first give a concept called "duty cycle," which is defined as the ratio of cycles a workstation commits to local jobs to the number of cycles available for parallel processes. Based on this and the earliest starting time at each workstation, they present an algorithm to select a subset of workstations from a given set to maximize the effective speedup.

Our scheme is also different from real-time systems. As we discussed before, a process of a parallel job is not a real-time process. It should be executed in the time-sharing mode and be neither fast nor slow but just to catch up with the slowest process to have them reach the communication phase on time. In contrast, the real time process has the highest priority to run. It will affect the local user response time significantly, which violates the resource sharing principle between local user jobs and parallel jobs. The difference between real-time scheduling and power preservation is that for the first, the scheduler allocates *all* CPU power to the real time process while for the second, it allocates only the preserved *part* of its CPU power to the process.

In addition, power preservation is more flexible than the real-time mode. When there are local jobs, it preserves a certain amount of power for parallel jobs and gives the remaining to local jobs. Consequently, the performance of local jobs is predictable. When there are no local jobs, all power is allocated to parallel jobs. As we discussed in Section 4, if no time-sharing process exists, PJ0 will be in the highest priority. The parallel process is executed even when it has PJ0 priority. The tspj_dptbl table indicates that it keeps the same priority after it runs out of its time slice. However, when some local user processes are available, their priorities are higher than that of PJ0. They interrupt the execution of parallel processes and run. So the system resumes to its power preservation state. Similarly, if there are no parallel jobs, the TS processes are executed as if there were no any change to SVR4, and use all the CPU power to proceed.

7. CONCLUSIONS

Our performance evaluation results indicate that to effectively schedule parallel processes on a heterogeneous NOW, we should consider both architecture information and system-wide characteristics. Using specific heterogeneous NOW information such as the power weight and the preserved power for parallel jobs in each workstation, and an abstract application program model, we propose the self-coordinated local scheduling scheme. This method coordinates the execution pace of a parallel job using the local scheduler based on coscheduling principles. Simulation results show its effectiveness. The power preservation in each workstation makes a fair power distribution to guarantee the performance of both local and parallel jobs and achieves the global coordination by local scheduling, which reduces the cost of coscheduling in NOWs significantly. The scheme can be applied to schedule multiple parallel jobs as well.

The scheme can be extended in many directions. One direct extension is that the preserved power may change from time to time based on requirements of local and parallel job users and the utilization of workstations in NOWs. We are studying these variants. Besides computing power, there are some other factors such as memory and I/O capability which need to be modeled and taken into considerations. We are also investigating the communication interaction between local user jobs and parallel jobs at a lower network level [6] and applying the scheme to wider area NOW scheduling applications.

ACKNOWLEDGMENTS

We thank our colleagues Jon Weissman and Neal Wagner at the University of Texas at San Antonio for reading the paper and making helpful comments. We appreciate Andrea Dusseau at University of California, Berkeley, and Evangelos Makatos at FORTH in Greece for reading an early version of this paper and for their constructive suggestions. We thank the anonymous reviewers for their comments and suggestions.

REFERENCES

1. Arpaci, R. H *et al.* The interaction of parallel and sequential workloads on a network of workstations. *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.* May 1995, pp. 267–278.
2. Atallah, M. J. *et al.* Models and algorithms for co-scheduling compute-intensive tasks on a network of workstations. *J. Parallel Distrib. Comput.* **16** (1992), 319–327.
3. Bailey, D. *et al.* The NAS parallel benchmarks. *Int. J. Supercomputer Appl.* **5,** 3 (Fall 1991), 63–73.
4. Crovella, M. *et al.* Multiprogramming on multiprocessors. *Proceedings of 3rd IEEE Symposium on Parallel and Distributed Processing.* 1991, pp. 590–597.
5. Douglis, F., and Ousterhout, J. Transparent process migration: Design alternatives and the Sprite implementation. *Software Practice Exper.* **21,** 8 (1991), 757–785.
6. Du, X., Dong, Y., and Zhang, X. Characterizing communication interactions of parallel and sequential jobs on networks of workstations. *Proceedings of IEEE Annual International Conference on Communications.* June 1997, pp. 1133–1137.
7. Dusseau, A. C., Arpaci, R., and Culler, D. Effective distributed scheduling of parallel workloads. *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.* May 1996, pp. 25–36.
8. Feitelson, D. G., and Rudolph, L. Gang scheduling performance benefits for fine-grain synchronization. *J. Parallel Distrib. Comput.* **16,** 4 (Dec. 1992), 306–318.
9. Goodheart, B., and Cox, J. *The Magic Garden Explained: The Internals of Unix System V Release 4.* Prentice-Hall, New York, 1994.
10. Gupta, A., Tucker, A., and Urushibara, S. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. *Proceedings of the 1991 ACM SIGMETRICS Conference.* May 1991, pp. 120–132.
11. Krueger, P., and Babbar, D. Stealth: A liberal approach to distributed scheduling for networks of workstations. Technical Report, OSU-CISRCI/93-TR6. Ohio State University, 1993.
12. Nichols, D. Using idle workstations in a shared computing environment. *Proceedings of the 11th ACM Symposium on Operating Systems Principles.* 1987, pp. 5–12.
13. Ousterhout, J. Scheduling techniques for concurrent systems. *Proceedings of the 3rd International Conference on Distributed Computing Systems.* Oct. 1982, pp. 22–30.
14. Sobalvarro, P. G., and Weihl, W. E. Demand-based co-scheduling of parallel jobs on multiprogrammed multiprocessors. *Proceedings of the IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing.* 1995, pp. 63–75.
15. Zhang, X., and Yan, Y. Modeling and characterizing parallel computing performance on heterogeneous NOW. *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing.* Oct. 1995, pp. 25–34.

XING DU is a post-doctoral researcher in the Department of Computer Science at the College of William and Mary. He received his B.S., M.S., and Ph.D in computer science from Nanjing University, P.R. China in 1986, 1989, and 1991, respectively. Since 1991, he has been on the computer science faculty at Nanjing University. His research interest includes distributed/parallel processing, performance evaluation, and human–computer interaction.

XIAODONG ZHANG is a professor of computer science at the College of William and Mary. He received his B.S. in electrical engineering from Beijing

Polytechnic University in 1982, and the M.S. and Ph.D in computer science from the University of Colorado at Boulder in 1985 and 1989, respectively. His research interest is primarily in the areas of parallel and distributed computation, computer system performance evaluation, and high performance scientific computing. Before joining William and Mary, he was on the computer science faculty at the University of Texas at San Antonio, where he established and directed the High Performance Computing and Software Laboratory. He is on the Editorial Board of *IEEE Transactions on Distributed Systems* and is currently chairing the Technical Committee on Supercomputing Applications of the IEEE Computer Society.

Received August 27, 1996; accepted August 21, 1997