Efficient Local Scheduling of Parallel Tasks and Communications on Heterogeneous Networks of Workstations [*]

Xing Du Yingfei Dong Xiaodong Zhang
High Performance Computing and Software Laboratory
University of Texas at San Antonio
San Antonio, Texas 78249

Abstract

The network of workstations (NOW) we consider for local scheduling of parallel tasks and communications is heterogeneous and nondedicated, where computing power varies among the workstations, and multiple jobs may interact with each other in execution. Local scheduling is a standard method on networks of workstations, where each node in the system makes independent scheduling decisions. Coordinating parallel tasks and minimizing communication delays are two important and difficult issues for local scheduling. We address the two issues by presenting a scheduling technique, called *self-coordinated local scheduling*. To coordinate parallel tasks, each local scheduler schedules both parallel tasks and sequential user jobs based on a static power preservation in each workstation. Thus, the parallel tasks in one computational phase can be guaranteed to complete within a time period. To minimize the communication latency, each local scheduler uses a non-preemptive strategy — tries to make one communication activity complete in a continuous time-frame. Our experiments show that the power preservation in each workstation improves the performance of both sequential and parallel jobs, and network utilization. Furthermore, the non-preemptive communication strategy reduces general communication contention and delay.

Key Words: Co-scheduling, heterogeneous networks of workstation, local scheduling, parallel processing and time-sharing.

[*]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.

1 Introduction

Networks of Workstations (NOW) have become important distributed platforms for large-scale scientific computations. A NOW system is highly cost effective, and widely available. In practice, a NOW system is heterogeneous and time-sharing. These two unique performance factors have been quantitatively characterized by metrics and experiments. The speedup, efficiency and scalability of parallel computing on a NOW are also defined by considering the heterogeneity and time-sharing effects [11]. Scheduling policies on multiprocessor/multicomputer systems are classical topics, and have been studied by many people on different types of systems, such as shared-memory systems [8] [10], distributed memory hypercube [3] and mesh [6]. Since these policies/algorithms are designed for a dedicated homogeneous system, they may not be directly applicable to heterogeneous NOW systems.

In order to effectively manage the interaction between parallel jobs and user jobs and well utilize the resources of a heterogeneous NOW, a scheduler must consider both architecture- and application- specific information as well as system-wide characteristics. Local scheduling is a standard method on networks of workstations, where each node in the system makes independent scheduling decisions. Coordinating parallel tasks and minimizing communication delays are two important and difficult issues for local scheduling. We address the two issues by presenting a scheduling technique, called *self-coordinated local scheduling*. To coordinate parallel tasks, each local scheduler schedules both parallel tasks and sequential user jobs based on a static power preservation in each workstation. Thus, the parallel tasks in one computational phase can be guaranteed to complete within a time period. To minimize the communication latency, each local scheduler uses a non-preemptive strategy — tries to make one communication activity complete in a continuous time-frame. The objectives of this study are to provide insight into the performance of scheduling policies on heterogeneous NOW, to examine ways of taking advantage of heterogeneity and time-sharing effects, and to effectively use local scheduling on a NOW platform for both parallel and sequential jobs.

The organization of the paper is as follows. Section 2 characterizes the unique features of heterogeneous NOW, and describes network and programming models. We also introduce experimental evaluation methods for this study. In section 3, we present the self-coordinated local scheduling — the motivation, coordination of parallel tasks and the management of message-passing and synchronizations. The experiment and simulation results used to evaluate the performance of the local scheduling are reported. We summerize the work and discuss the future work in Section 4.

2 Models of NOW and evaluation methods

2.1 Unique system features of NOW

Besides heterogeneity and time-sharing effects, a NOW system has three other unique features in comparison with a multiprocessor/multicomputer system.

- *Low bandwidth communication.* The communication speed of current networks of workstations is relatively low. Even when high speed networks, such as ATM networks, are used, the internode communication is still a more serious bottleneck of parallel computation than a computation on a modern multiprocessor/multicomputer system. Therefore, only coarse-grained or medium-grained parallel applications are suitable for running on NOW. This is a consideration for designing scheduling policies of NOW.

- *Random network topology.* A multiprocessor/multicomputer system usually connects processors by some standard topology, such as mesh (Intel Paragon), hypercube (Intel and NCube hypercubes), fat-tree (CM-5) and ring (KSR). In comparison, a NOW system connects workstations in a random way, and its topology may change from time to time in practice. Therefore, scheduling algorithms for NOW should not assume a particular network topology, but a more general network structure.

- *Multidirectional scaling.* A NOW system can be scaled in three directions: by increasing the number of workstations (physical scaling), by upgrading workstation powers (power scaling), and by combining both physical scaling and power scaling. In physical scaling, a parallel job tends to increase average overhead latency due to more workstations involved in communication and synchronization. In power scaling, communication complexity remains constant, but the computation and communication ratio increases, which would reduce the computation efficiency. How to balance the two scaling methods to achieve optimal performance for a parallel job is a complex issue.

These unique features of NOW introduce new requirements for developing scheduling policies. In this paper, we only consider heterogeneity, time-sharing and network communication issues. We are conducting another study focusing on NOW scalability.

In developing scheduling algorithms on heterogeneous NOW, major questions include determining how heterogeneity affects a standard scheduling policy on multiprocessor/multicomputer, such as co-scheduling, and how much we can use a standard scheduling policy on a NOW, such as local scheduling. The co-scheduling policy [7] generally results in good parallel program performance [5], and is widely used to schedule parallel processes involving frequent communication and synchronization. The basic idea of co-scheduling is to group parallel processes involving communications and synchronization together, and to ensure that all the processes in the group start at the same time and execute at the same pace. This method is particularly effective for parallel applications partitioned into multiple tasks of equal size, running on a homogeneous multiprocessor/multicomputer system. Co-scheduling will ensure that no process will wait for a non-scheduled process for synchronization/communication, and will minimize the waiting time at the synchronization point. In order to make the co-scheduling policy effective on a heterogeneous NOW system, the capability of each workstation must be considered so that the waiting time could be minimized or used. This capability, called *power weight*, is quantitatively measured for each workstation [11]. In a practical NOW system, a commonly used scheduling method is to use the OS kernel, such as Unix, on each workstation to schedule the parallel tasks independently. This is called local scheduling [1] [9]. However, local scheduling may delay the execution time of a parallel job significantly because a process may wait for a non-scheduled process for communication/synchronization. Since local scheduling is standard on a NOW system, we should take advantage of it. One issue we address in the paper is how to effectively revise local scheduling policies using the co-scheduling principle for parallel computation on heterogeneous NOW.

2.2 Heterogeneous NOW models

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

- $M = \{M_1, M_2, ..., M_m\}$ is set of heterogeneous workstations (m is the number of workstations). The computation capacity of each workstation is determined by the power of its CPU, I/O and memory access speed.
- Net is a standard interconnection network for workstations, such as an Ethernet or an ATM network, where the communication links between any pair of the workstations have the same bandwidth.

Based on the above definition, if a NOW consists of a set of identical workstations, the system is homogeneous. Moreover, a heterogeneous network can be divided into two classes: a dedicated system, where each workstation is dedicated to execute tasks of a parallel job, and a nondedicated system, where each workstation executes its normal routines (also called owner workload), and only the idle CPU cycles are used to execute tasks of the parallel job. The term *owner utilization* of a workstation is used to represent the owner workload rate.

A parallel application program, denoted by $App(I)$, where I represents its input-parameter, is assumed to have m tasks $App_1(I)$, $App_2(I)$, ..., $App_m(I)$. Task $App_i(I)$ $(1 \leq i \leq m)$ is assigned and executed on workstation M_i. The size of program $App(I)$ is denoted by $|App(I)|$, which can be defined as the number of required operations to solve $App(I)$ [12]. Thus, $|App(I)| = \sum_{i=1}^{m} |App_i(I)|$. It will simplify notation significantly if we assume in the remainder of this paper that the program parameters are included for each case. Thus, when we write App, we really mean $App(I)$.

The power weight of a workstation refers to its computing capability relative to the fastest workstation in a system. 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 we define the speed of the fastest workstation in the system as one basic operation per time unit, the power weight of each workstation denotes a relative speed. If $T(App, M_i)$ gives the execution time for executing program App on workstation M_i, the power weight can be calculated by the measured execution times as follows:

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

The total power weight of a system is defined as the sum of the power weight of each workstation, which represents the accumulated computing capability of the system:

Model	HZ	Memory(MB)	Cache (KB)	SPECint92	SPECfp92
S20-HS21	125	32	256	131.2	153.0
S20-HS11	100	32	256	104.5	127.6
S20-50	50	32	36	76.9	80.1
S5-85	85	32	24	65.3	53.1
S5-70	70	16	24	57.0	47.3
S10-30	50	32	36	45.2	54.0
Classic	36	16	6	26.4	21.0

Table 1: Performance parameters of the seven types of Sun workstations.

$$TW(App) = \sum_{i=1}^{m} W_i(App) \tag{2.2}$$

The network heterogeneity can be quantified as the variation of computing power in a network system. Using the power weight of the fastest workstation (=1), we define the network heterogeneity as follows:

$$H = \frac{\sum_{j=1}^{m}(1 - W_j(App))}{m}. \tag{2.3}$$

For detailed information about the power weight and network heterogeneity, the interested reader may refer to [11].

We define the parallelism degree of an application program, denoted by P, as the ratio of the number of workstations actively executing in parallel to the total number of workstations in the whole execution phase of the program. This is measured by accumulating the percentage of the active computing time in each workstation for the parallel program.

2.3 Evaluation methods

We used direct simulation for performance evaluation, which measures the run time of major computation blocks on each target host workstation. The heterogeneous NOW for testing consists of 30 workstations, where 7 types of Sun SPARCstations with different computing powers are available. The 7 types cover a large range of very fast and very slow Sun workstations with a single processor. The workstations are connected by an ATM network and an Ethernet of 10Mbps bandwidth. Table 1 lists the performance parameters for the 7 types of workstations. In this paper, we do not consider the effects of networks and communications on scheduling.

We selected four programs from the NAS parallel benchmarks [2]: CG, EP, MG, and IS, and a standard parallel LU decomposition program. Kernel CG users conjugate gradient method and power method to iteratively approximate the smallest eigenvalue of a symmetric and positive definite sparse matrix. The problem size we used is 14,000 (class A).

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

Table 2: The average power weight of each type of the Sun workstations measured by programs CG, EP, IS, LU, and MG.

Kernel EP (Embarrassing Parallel) represents the computation and data movement characteristics of large-scale computational fluid dynamics applications. It executes 2^n iterations of a loop, where a pair of random numbers are generated and tested for whether Gaussian random deviates can be made from them by a specific scheme, with n as the input parameter. Each parallel process only participates in the communication of a 44 byte packet once at the end of processing. The structure of this program is especially suitable for a homogeneous multicomputer system.

Kernel MG (Multigrid) is a 3D multigrid benchmark program for solving a discrete Poisson problem $\Delta^2 u = v$ with periodic boundary conditions on a 3D grid of $256 \times 256 \times 256$ cells. The amount of border communication increases as the number of processor nodes increases.

Kernel IS (Integer Sort) performs 10 rankings of 2^{23} integer keys in the range of $[0, 2^{19}]$. The keys are evenly distributed to each workstation, which may not be suitable for a heterogeneous system. Message-passing is frequent.

Program LU (LU decomposition) factors an $n \times n$ matrix A into the product of a lower triangular matrix L and upper triangular matrix, U. In this program, the matrix A is distributed by blocks in a natural wrap mapping. There are three communication phases: factor broadcast, reduction, and row swapping when a pivot occurs. The computation load gets more unbalanced among the workstations as the each column is processed.

The power weight of each workstation is application dependent. Our experiments indicate this is mainly related to the problem size. If the memory allocation of a program is not larger than the available memory size of a workstation, the differences among measured power weights using different application programs on the workstation is insignificant. Therefore, by carefully choosing the problem size of each program, we measured the power weight of the 7 types of Sun workstations. The power weight is finally calculated by averaging all the power weights measured by running the different programs. All the experiments were repeated 10 times in a dedicated system. The standard deviations of multiple measurements are less than 0.05. Table 2 gives the average power weights.

Each type of workstation and its power weight are represented by a letter for simplicity (see Table 2), such as A for S20-HS21, ..., and G for Classics. In all our experiments, the total power of 10 was used, which is equivalent to ten S20-HS21 workstations. The definition heterogeneity in (2.3) indicates that for a given total power weight (TW) and a given heterogeneity (H), the total number of workstations, m, is determined. However, for the given m, different heterogeneous systems can be formed by combining different types of workstations. For example, for $TW = 10$,

(H, m)	Combination-1	Combination-2	Combination-3	Combination-4
(0.1, 11)	9A+B+G	9A+C+D	8A+3B	N/A
(0.3, 15)	3A+4B+5C+D+E+G	2A+4B+8C+G	3A+3B+6C+D+2E	A+5B+9C
(0.6, 25)	A+15D+9G	A+24F	N/A	N/A

Table 3: The workstation combinations we used for each given pair of H and m, where the total power is 10.

$H = 0.1$, and $m = 11$, the system could consist of 9 S20-HS21, one S20-HS11 and one Classic (9A+B+G), 9 S20-HS21, one S5-85 and one S5-70 (9A+C+D), or 8 S20-HS21 and 3 S20-HS11 (8A+3B). For given TW, H, and m, different workstation combinations may provide different degrees of parallelism, P, for a program implementation. Table 3 lists the workstation combinations we used for $(H = 0.1, m = 11)$, $(H = 0.3, m = 15)$ and $(H = 0.6, m = 25)$ respectively, where the total power weight is 10.

3 Self-coordinated local scheduling

This local scheduling technique consists of two parts: parallel task coordination and communication/synchronization management. In this section, we first present our motivation through observations and evaluation of task scheduling and communications. We describe the self-coordinated local scheduling after then.

3.1 Motivation

3.1.1 Observation and motivation of parallel task scheduling

In order to design an efficient scheduling algorithm for parallel jobs on NOW without considering network contention, two issues must be well addressed: how to coordinate the execution of simultaneous tasks of a parallel job (inter processor coordination), and how to manage the interaction between parallel jobs and local sequential user jobs (intra processor coordination). Co-scheduling provides a good service for inter processor coordination on multicomputers, but may not be necessary on heterogeneous NOW for three reasons. First, it is expensive to implement the policy, which may cause high overheads due to relatively low bandwidth networks to handle frequent communications and context switches. Second, studies have shown that co-scheduling is necessary for fine-grained parallel application programs [1]. A heterogeneous NOW platform is targeted at coarse-grained parallel applications. Third, in practice, a NOW system is nondedicated, and fast response times are important to both parallel job users and sequential job users. However, co-scheduling of parallel programs would interact poorly with sequential jobs. Particularly if the sequential job is a owner job of a workstation, then the performance of parallel jobs with frequent communications and synchronizations would be degraded significantly. Poor scheduling may also increase the response times of both sequential and parallel jobs and make them unpredictable. This may be particularly unacceptable for interactive local users. A common approach for solving this problem is to avoid the coexistence of both parallel jobs and local sequential jobs by process migration [1]. The cost of process migration may be significant, and it is not trivial to decide where to migrate a process.

We have seen the potential to improve performance of both parallel jobs and sequential jobs on a heterogeneous NOW by two facts: the workstations are under-utilized when parallel jobs are co-scheduled in a dedicated environment, and the workstations are under-utilized when sequential jobs are run without parallel jobs. Thus, it is possible to combine them together to increase system utilization. However, when the two types of jobs are combined, the response times of both suffer. A goal of this study is to exploit the potential, to increase the system utilization, and to minimize the response times for both parallel and sequential jobs.

As we discussed previously, simply implementing co-scheduling on a heterogeneous NOW is not a good choice. However, its principle of coordinating multiple task execution rates to minimize waiting time is essential for scheduling policies. If the local scheduler on each workstation preserves a certain amount of power for parallel jobs and the rest of the power for local sequential jobs based on its power weight and load balance of the system, the parallel jobs may be able to coordinate themselves without a global co-scheduler. The response times of both parallel jobs and sequential jobs are adjustable and predictable. Parallel jobs and sequential jobs can use each other's preserved power if free cycles are available. Furthermore, if the local scheduler gets some knowledge about parallel jobs, such as the task size, the execution pace of a parallel job may be able to more precisely coordinate locally. One condition to make this local scheduling efficient is that tasks of a parallel job should start simultaneously with a reasonable time skew, which can be implemented by some programming tools, such as PVM [9].

3.1.2 Observation and motivation of network contention

In many large scale scientific computations suitable for NOW, program structures are in a "divide and conquer" type which partitions a problem into smaller parts, and then combines the individual solutions into the solution for the whole iteratively. This requires that data be distributed among workstations according to the problem partitioning. The distributed data are updated by a certain method at each iteration, and then communicated to other workstations until a solution tolerance is satisfied. In each iteration, considered as a computational phase, usually no data communication is required among the workstations, and each workstation computes independently. At the end of each iteration, each workstation acquires the updated data from other workstations. In addition, a synchronization barrier is required to guarantee that the next iteration does not start until all workstations finish the computation and data acquisition. Task coordination based on the co-scheduling principle which was discussed in the previous section is particularly effective for this type of program structure. However, when the communication/synchronization phase is considered, local scheduling becomes more complex. This is because task coordination in the computational phase and message-passing management in the communication/synchronization phase may have conflict interests. Task coordination tries to schedule the tasks to run at a certain pace to reach the synchronization point. Thus, the waiting time is minimized. This effort may make multiple tasks arrive at the communication/synchronization point at nearly the same time to compete the network, thus the network contention increases.

The contention is particularly high when an Ethernet bus is used, which is a commonly used network for current NOW systems. We measured execution performance of 5 benchmark programs using PVM on an Ethernet of 8 workstations (B+5E+2F). The communication patterns of the 5 programs are classified into following 4 types:

- *all-to-all:* Each communication task sends messages to all other tasks in the system.
- *k-to-k:* Message passing is conducted between two set of communication tasks. Each set has k tasks, where k is larger than 1 and less than the total number of communication tasks.
- *to-neighbors:* Communications tasks are logically connected as a mesh. Each task sends messages to its 4 neighbors.
- *transpose:* For $i = 1, ..., m$, communication task p sends messages to task $(p + i)\ mod\ m$, where m is the total number of communication tasks.

Table 4 lists the communication type, the total number of messages, average message length, the total communication time and the network contention ratio of each program. The network contention ratio measures the percentage of contention time in the total communication time. The contention ratio was application program dependent, which varied from 75% to about 96%. Communication type is a factor affecting communication performance. In general, the all-to-all type generates a larger number of messages than other types, and causes higher network contention. For example, the contention ratio of the IS program was 96%. Another important factor is the average message length. The smaller the message, the higher possibility to cause contention would be. For example, the LU program of the smallest average message length among the programs still had 75% contention ratio although the amount of communications was less. In addition, the CG program with relatively small average message size had 88% contention ratio. Our measurements indicate that an effective communication task management in the communication/synchronization phase is necessary because the performance improvement potential is quite high.

To keep using the preserved local scheduling in a round-robin fashion in the communication/synchronization phase may cause a workstation to complete a message-passing at different time slice discontinuously. Frequent context switches and network contention would significantly delay the communication/synchronization phase. The proposed strategy is to let workstation complete the phase as soon as possible and as continuous as possible. We will give detailed descriptions of the management technique and its performance evaluation and comparisons with other techniques following the presentation of task coordination.

Programs	type	message length (KB)	communication time (sec)	contention ratio
IS	all-to-all	36	222	96%
MG	transpose	34	89	88%
CG	k-to-k	1.3	109	88%
FT	transpose	61	80	84%
LU	neighbor	0.15	87	75%

Table 4: Communication measurements of the five application programs on a heterogeneous network of 8 workstations.

3.2 Self-coordinated scheduling in the computational phase

3.2.1 Power preservation for parallel task coordination

A key issue of this local scheduling method is to preserve a portion of the power in each workstation for parallel jobs. In particular, each workstation's power is divided to serve three types of jobs: the required OS kernel processes, sequential jobs, and parallel jobs. Based on our measurements, the kernel processes in each workstation used no more than 5% of the power of the machine. Here we propose two different methods for power preservation. The first way is based on the power weight of the slowest workstation. Thus, the preserved power weight for parallel jobs in each workstation, denoted by ρ, is determined by the maximum available power weight for parallel jobs in the slowest workstation, which is equivalent to

$$\rho = W_s(1 - R_{kernel}(s) - R_{user}(s)), \tag{3.4}$$

where W_s is the power weight of the slowest workstation, $R_{kernel}(s)$ is the percentage of power for the kernel in the slowest workstation, and $R_{user}(s)$ is the percentage of power for user jobs in the slowest workstation. We define free power weight on the slowest workstation as:

$$F_s = 1 - R_{kernel}(s) - R_{user}(s). \tag{3.5}$$

Figure 1 (left) presents an example of this power preservation method. In this example, 15 Sun workstations of 4 types (A, B, C and G) are used. The slowest workstation is G, which provides the power weight of 0.2 to parallel jobs. Thus, 0.2 is the preserved power weight in the NOW for parallel jobs.

In practice, the bottleneck of parallel jobs may not be the slowest workstation, but a powerful workstation which has a large number of local user jobs, and is only able to provide a small percentage of its power for parallel jobs. The second method of power preservation is to identify the minimum capable power weight for parallel jobs among workstations. Each workstation first calculates its own capable power weight for parallel jobs:

$$\rho_i = W_i(1 - R_{kernel}(i) - R_{user}(i)), \tag{3.6}$$

where W_i is the power weight of workstation i, $R_{kernel}(i)$ is the percentage of power for the kernel in workstation i, and $R_{user}(i)$ is the percentage of power for user jobs in workstation i $(i = 1, ..., m)$. We define the free power weight on workstation i, $(i = 1, ..., m)$, as:

$$F_i = 1 - R_{kernel}(i) - R_{user}(i). \tag{3.7}$$

Then 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:

$$\rho = min_{i=1}^{m} \rho_i, \tag{3.8}$$

where m is the number of workstations in the system. The second power preservation method is more flexible. Figure 1 (right) presents an example of this methods. Again, 15 Sun workstations of 4 types (A, B, C and G) are used. The minimum available power weight for parallel jobs is 0.15, which is not provided in the slowest workstation (G), but in the 4th workstation of the B type. Thus, the preserved power weight in the system is based on this value.

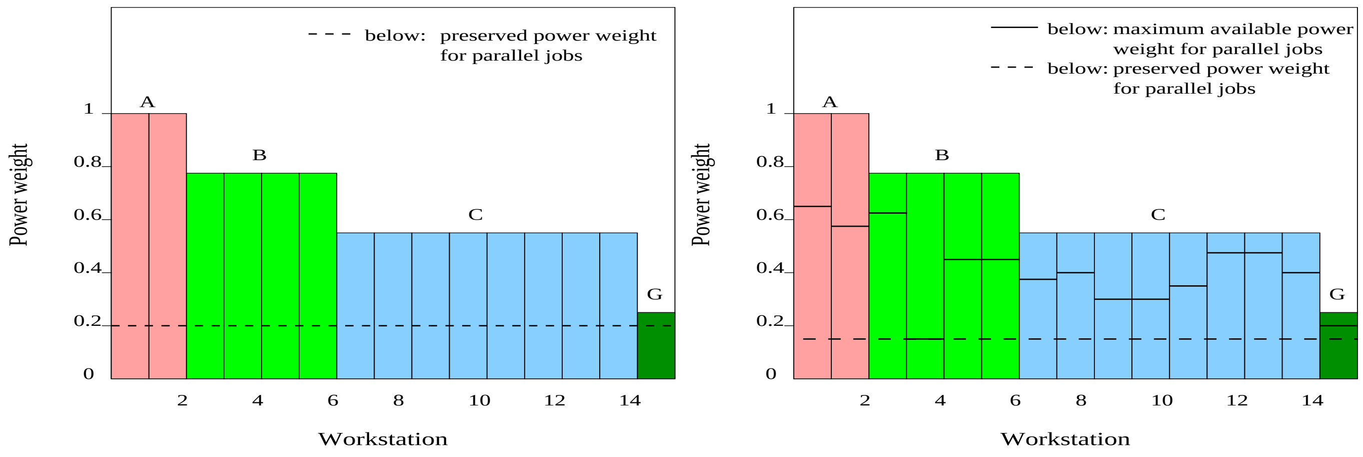

Figure 1: Examples of power preservation for parallel jobs based on the slowest workstation power weight (left), and based on the minimum available power weight in the workstation (right).

The equivalent power preservation in each workstation for parallel jobs allows us to "simulate" co-scheduling in a virtual homogeneous system. Our system administration trace file indicates that the slow workstations have little chance to be accessed by interactive users, but are used as a node in a parallel system, because interactive users intend to use the fast workstations for short response time. The remaining percentage of the power in a workstation can be used for sequential jobs. The more powerful a workstation is, the larger percentage of its power can be preserved for the sequential jobs.

3.2.2 Local scheduling coordination

Preserving the power in each workstation for parallel jobs does not guarantee coordination of parallel tasks because tasks need to be further locally scheduled in each workstation to ensure all the tasks will finish within a reasonable time period. This seems somewhat application dependent. Before describing activities in the local scheduler of each workstation, we begin defining a general model for parallel programs running on NOW. A typical NOW parallel program is coarse-grained, and consists of one task per workstation on a fixed number of workstations throughout execution. The computation completes after a number of loops. In each iteration, phases of local computation alternates with phases of communications and synchronizations. The basic structure of the general model is as follows

```
Loop

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

end Loop
```

If the local scheduler can ensure all the tasks finish within a reasonable time period, all possible communication and synchronization activities following the local computation will be coordinated. Here we temporarily use an application program dependent parameter for the local scheduler, the size of parallel tasks, denoted by TS, 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 task on the slowest workstation is

$$t_s = \frac{TS}{Pow(s) \times F_s}, \tag{3.9}$$

where $Pow(s)$ is the power of the slowest workstation, and F_s is the free power weight in the slowest workstation; and the time to finish a task on workstation i is

$$t_i = \frac{TS}{Pow(i) \times F_i}, \tag{3.10}$$

where $Pow(i)$ is the power of workstation i, and F_i is the free power weight of workstation i. If round-robin time-sharing fashion is used in the local scheduler of each workstation, the number of time slices used to finish a task on the slowest workstation is

$$N_{slice}(s) = \frac{t_s}{\delta_s}, \tag{3.11}$$

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

$$N_{slice}(i) = \frac{t_i}{\delta_i}, \tag{3.12}$$

where δ_i is the length of a time slice in workstation i, and $i = 1, ..., m$. Since the slowest workstation is the bottleneck of parallel jobs, if all the tasks finish within the time period of t_s in each loop, the performance would be optimal and equivalent to that in a dedicated NOW using co-scheduling. However, within t_s, there are $\frac{t_s}{\delta_i}$ time slices available in workstation i, which is larger than $N_{slice}(i)$. This means that all workstations except the slowest one have more time slices for additional processes. In order to make even distributions of time slices for parallel tasks and additional processes, we use (3.9), (3.10), (3.11) and (3.12) to quantify the time interval for a time slice assignment to a parallel task in workstation i, ($i = 1, ..., m.$). Therefore, if a time slice is given to the parallel task in workstation i within no less than a time quantum of $\frac{t_s}{\delta_i N_{slice}(i)} = \frac{t_s}{t_i}$, the local scheduler will ensure that within t_s, all the tasks in a local computation phase will finish. By (3.9) and (3.10), we further obtain

$$\frac{t_s}{t_i} = \frac{Pow(i) \times F_i}{Pow(s) \times F_s} \tag{3.13}$$

The time quantum in (3.13) is architecture dependent rather than application program dependent.

The scheduling technique in the computational phase on a heterogeneous NOW is outlined as follows:

1. Determine ρ, the available power weight for parallel jobs on s, the slowest workstation.

2. Broadcast $Pow(s)$ and F_s to each local scheduler.

3. The local scheduler in workstation i calculates its pace to assign a time slice to a parallel task by (3.13).

This self-coordinated local scheduling is able to make parallel jobs perform as well as jobs on a dedicated heterogeneous NOW by co-scheduling. On the other hand, the response times of local user jobs would be acceptable and predictable, because local user jobs on workstation i are guaranteed to be assigned at least $W_i \times R_{user}(i)$ power weight. Figure 2 gives an example of the self-coordinated scheduling for 15 simultaneous tasks in one iteration, where four types of Sun workstations (A, B, C, and G) are used. In t_s, the slowest workstation G uses all 12 time slices, while workstation C uses 6, workstation B uses 4 and the fastest workstation A uses 3 time slices for their local parallel tasks, respectively. The self-coordinated local scheduling ensures that all simultaneous tasks complete in the end of the t_s.

There are two ways to handle multiple parallel jobs. The first way is simply to increase TS by accumulating the multiple parallel task sizes in (3.9) and (3.10). Using the increased t_s and t_i, we readjust the other parameters for the local scheduler, such as $N_{slice}(s)$, and $N_{slice}(s)$, $(i = 1, ..., m)$.

The other way to handle multiple jobs is a dynamic approach assuming more than m workstations can be used for parallel jobs. Instead of increasing the values of t_s and t_i, we form another group of workstations for parallel jobs, which include the workstations which currently process a parallel job and have additional powers, and new workstations. Again, W_i (the power weight) and ρ_i (the preserved power weight) of each workstation are used by the local scheduler for power distributions to coordinate the new parallel job.

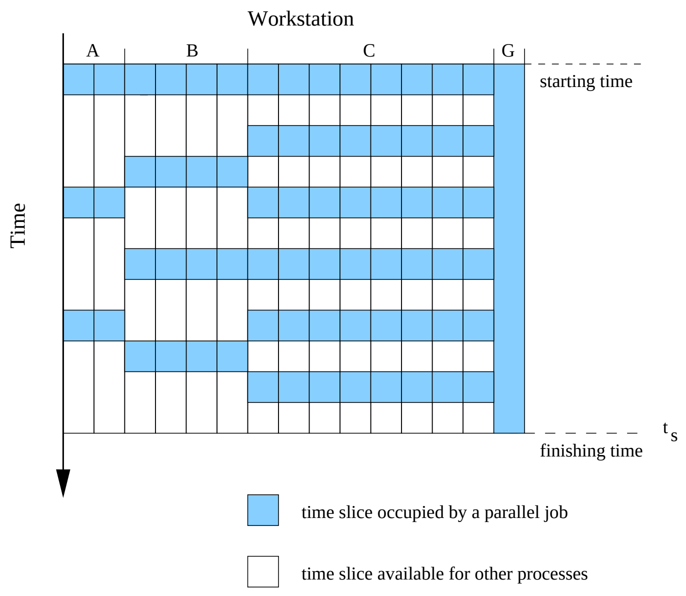
Workstation
A
B
C
G
Time
starting time
finishing time
t_s
time slice occupied by a parallel job
time slice available for other processes

Figure 2: An example of the self-coordinated local scheduling for 15 simultaneous tasks in one iteration where four types of Sun workstations (A, B, C, and G) are used.

3.2.3 Performance evaluation

We used direct simulation to test the self-coordinated local scheduling by running the 4 programs on a heterogeneous NOW with different workstation combinations. In order to focus on performance comparisons between the self-coordination local scheduling and standard local scheduling, we selected results on 4 workstations of B, C, F and G to present here. For a system of a larger number of workstations, the slowdown differences are highly significant between the two local scheduling methods, and it is difficult to observe the small changes of the self-coordinated local scheduling. In the direct simulation, the following system parameters were used: the time slice was 0.1 second; and a context switch took 200 μs. The EP program was iterated 10 times to increase its synchronization activities. Structures of the four application programs follow the programming model we defined in the previous section, except the IS, MG and LU have different task sizes. Figure 3 presents the slowdown factors of the self-coordinated local scheduling and standard local scheduling, which are relative to the parallel job response times of the four programs using co-scheduling on the dedicated NOW. The execution times of the EP program using self-coordinated local scheduling are very close to that using co-scheduling, differing by a factor of 1.09 times. The difference came from the scheduling skew when the parallel tasks started to run and from context switch overhead. Since the task size is equal in the EP program, and only one synchronization point occurs at the end of each iteration, the execution pace by the local scheduling is almost identical to that by co-scheduling.

The execution times of IS, MG and LU by self-coordinated local scheduling increased 15% to 25% in comparison with the times of co-scheduling. Besides scheduling skew and context switch overhead, there is another reason for the slowdown. The task size of the three programs are dynamically changed as the number of iterations increases. Reaching the end of the computations, the task sizes of these programs became quite small. When the execution time of a task size was close to a time slice, the execution pace was not well coordinated, because the time frame in the slowest workstation for the task, t_s, in (3.9) became very small, and round-robin scheduling became difficult in a tiny time space.

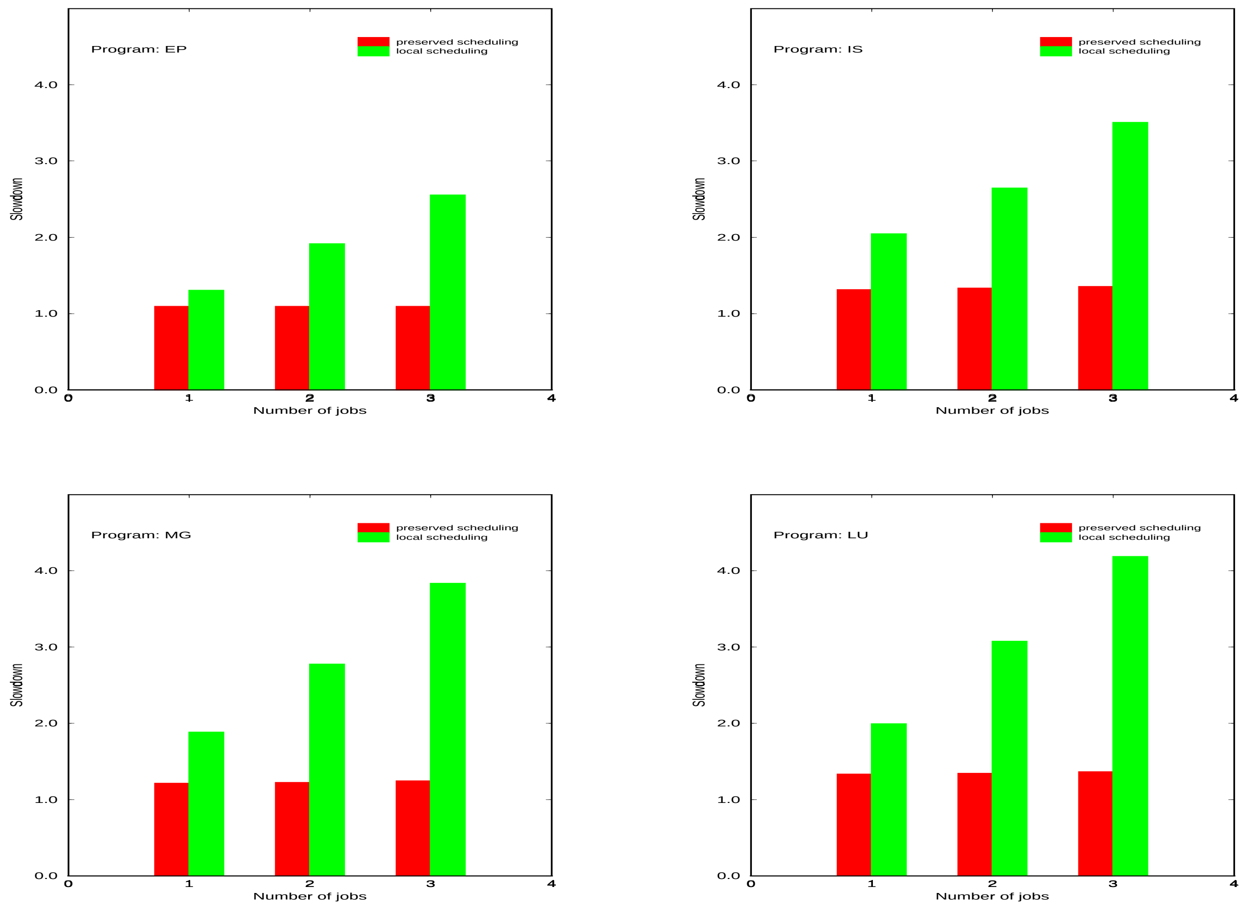

Figure 3: Performance of the self-coordinated local scheduling in comparisons with local scheduling and co-scheduling.

Scheduling its own tasks locally in each workstation following certain rules, self-coordinated local scheduling can almost ensure that all parallel tasks of one phase execute at the same pace, and complete within a period of time. In comparison with co-scheduling, the self-coordinated local scheduling has the following three sources which may disturb coordination of the execution pace. First, to initiate the start of a parallel job, there is some scheduling time skew, but the skew is tolerable. For example, using PVM, the maximum difference between the task start times across 15 workstations was about 20 ms based on our measurements. Second, the number of context switches may be more, because a task may be scheduled to execute at different times interleaved with other user jobs, instead of being executed at one time by co-scheduling. Finally, as we discussed previously, if the task size is small, the execution pace among simultaneous tasks may not be coordinated. However, our simulation experiments show that the overhead from the three sources was less than 30% in comparison with co-scheduling. The slowdown factors of self-coordinated local scheduling relative to that of the programs using co-scheduling remained constant as the number of parallel jobs increased, while the slowdown factors of local scheduling proportionally increased under the same condition.

3.3 Self-coordinated local scheduling in communication/synchronization phase

3.3.1 Pass the phase as soon as possible

The communication/synchronization phase is usually a critical section of parallel jobs. A message-passing can be only performed when it obtains both the workstation CPU and the network. If a communication task is given more chance to obtain the CPU, it would have a better chance to complete the communication/synchronization phase as soon as possible. This would also increase the possibility of overlapping this communication task with computational tasks on other workstations, decrease the possibility of network contention, and reduce the number of context switches. Thus, each workstation is either given a larger time slice to complete the communication when it obtains the network, or it is switched to process local user jobs when it does not obtain the network. In summary, if multiple parallel tasks arrive at the communication/synchronization phase simultaneously, only one task is allowed to pass at a time. Others are switched to process local user processes. This would reduce unnecessary network contention.

Here we give some implementation details of the management strategy. The time slice is non-preemptive. In other words, a communication task would not enter the phase until the current process finishes its time slice. As soon as the local scheduler is informed that the program enters the communication/synchronization phase, it gives all the preserved power of the workstation to the current task, and lets it run as fast as possible to pass the phase. If the task is blocked by its communication/synchronization partners or by network contention, it immediately yields the CPU and lets a local user job to run a time-slice. After that it checks if it can obtain the network. If so, it performs the communication by using all the preserved power.

Considering the two resources of the CPU and the network, there are four possibilities a communication/synchronization task faces. Only when the task gets both the CPU and the network, it perform message-passing. Table 5 lists the four possibilities and the corresponding decision made by the local scheduler. When the task leaves the communication/synchronization phase, the local scheduler resumes its normal scheduling mode.

There are two questions we should address about the decisions made by the local scheduler in the communication/synchronization phase. The first one is why the CPU is switched to a local user job rather than to spin when the network is not available for the communication. There are three

	CPU available	CPU not available
Network available	communications	waiting for a time slice
Network not available	switch to a user job	N/A

Table 5: The four resource combination possibilities and the corresponding decision made by the scheduler in the communication/synchronization phase.

reasons for this. First, experiments reported in [4] indicate that an immediate switch strategy outperforms two-phase blocking strategy (spin and then switch) for coarse grained parallel jobs on relatively slow networks. Second, immediately yielding the CPU to other local sequential jobs increase the system utilization. Finally, if a spin is chosen, how long to spin is hard to determine because an optimal length is architecture- and application- dependent. The strategy of spin-until-it-gets seems to be an effective way, but it may lead to a system deadlock. The second question is why the time slice is non-preemptive rather than preemptive. In the preemptive approach, as soon as a communication task requests the CPU, a currently running process is interrupted and yields the CPU to the task. Frequent context switching is the main feature of this approach. The shuffling effects may degrades the performance significantly. In our non-preemptive approach, if such case occurs, the parallel task waits until the running process finishes its time slice. The non-preemptive approach may delay the communication task at most one time slice. But this disadvantage may be offset by the facts that additional context switching overhead is avoided. In addition, user jobs may have better chance to complete. This is an important consideration for sequential interactive jobs.

3.3.2 Performance evaluation

We use a trace-driven simulation to evaluate and compare communication management policies on an Ethernet bus. The Ethernet bus is a shared media. Contention occurs when several communication tasks try to use the bus simultaneously. Two representative communication patterns we used for evaluation are the all-to-all pattern which is selected from the IS program, and the transpose pattern which is selected from the MG program. We focus on evaluating the changes of network contention caused from changing communication types and sizes, and communication task management policies.

The time delay of a message-passing is accumulated by the following three parts:

- *Effective communication time.* This includes the startup time (software overhead) and message transmission time (network latency). This is the minimum time delay for a message-passing communication.

- *Contention time.* This is the time a workstation spends to wait for a release of the network media, such as communication links, ports, et. al.

- *Blocking time.* This is the time a workstation spends to wait at a synchronization point in a program or in a synchronous send/receive protocol.

Besides the message-passing policy proposed for the self-coordinated local scheduling, we also selected two other policies for comparisons. One policy is the standard round-robin local scheduling, where the communication tasks are scheduled in the same way as the ones in the computational phase. The other policy is to completely ignore the local user jobs when task enters the communication/synchronization phase by either using the network if it is available or by spinning for its turn. Thus, the execution time of a communication task would be minimized from eliminating the context switch times for serving local user jobs. We call this policy, optimized local message scheduling. However, this approach may cause low system utilization, and deadlock may occur if multiple parallel jobs are involved. Since minimizing the time of a communication task is our objective, we use the execution time of the optimized local message scheduling to pass the communication/synchronization phase to compare those of the self-coordinated local scheduling and of the standard round-robin scheduling.

For a given network structure and its message-passing protocol, the effective communication time can be measured and is application program independent. However, the contention and blocking times are highly dependent on communication patterns and communication management policies. We simulated the same heterogeneous NOW where we did contention measurement (8 Sun workstations of B+5E+2F). Table 6 presents the contention ratios and blocking ratios of the simulation. The contention ratios of the two programs by simulation are consistent with our measurement results in Table 4. Both our measurement and simulation results indicate that contention time is the dominant part of the communication on an Ethernet bus. For example, the contention ratio is as high as 70% to 80% for both the IS and MG programs. Even the optimized local scheduling is used, the contention ratio is still above 60%. Compared with the contention, blocking is a relatively small part and communication pattern dependent. For example, the blocking ratio of IS is about 9%, and 17% for MG. This is because IS has the all-to-all communication pattern and message receiving was done randomly so that the blocking time is low. On the other hand MG has the transpose pattern and messages came from specific workstations so that the blocking time to wait for messages was high.

Figure 4 presents the comparisons of communication slowdowns of the self-coordinated local scheduling and the standard round-robin local scheduling for the all-to-all and transpose communication patterns. The communication code for the all-to-all pattern comes from the IS program, and the transpose pattern comes from the MG program. The slowdown factors are relative to the execution time of the same programs using the optimized local message scheduling on the simulated heterogeneous NOW. The simulation shows that the self-coordinated local scheduling reduced the total communication time by a factor of 1.5 in comparison with that of the standard round-robin local scheduling due to significant time reductions of network contention and blocking.

As the problem size of IS increased, the self-coordinated scheduling decreased the slowdown factors of the contention and blocking portions moderately. In contrast, the standard local schedul-

Program	Program size	Contention ratio (%)			Blocking ratio (%)		
		optimized	*coordinated*	*standard*	*optimized*	*coordinated*	*standard*
IS	2^4	63.1	70.7	76.8	6.5	5.2	14.9
	2^8	65.0	70.0	78.5	4.0	4.0	13.1
	2^{12}	62.6	67.2	80.0	6.5	4.5	11.6
MG	32	74.0	79.7	83.0	8.0	6.3	10.4
	64	65.8	75.1	72.5	17.4	11.9	20.8
	128	50.4	75.7	75.3	35.0	12.6	18.1

Table 6: Contention and blocking ratios of the IS and MG programs in the communication/synchronization phase scheduled by the optimized local message scheduling, the self-coordinated local scheduling and the standard local scheduling.

ing did not change the contention and increased the blocking time moderately. When the problem size of the MG program increased from 32 to 128, the number of messages increased. Thus, the self-coordinated scheduling increased the slowdown factor of the contention portion from 1.4 to 1.9. This effect was offset by the decrease of the blocking portion from 1 to 0.5. Therefore, the total slowdown factor almost remained constant. However, using the standard local scheduling, both contention and blocking slowdown increase as the problem size increased. The main reason for this was that the standard local scheduling would not ensure a communication task to complete in a given time slice. A communication task may have to request multiple network accesses, which would significantly increase blocking and contention times.

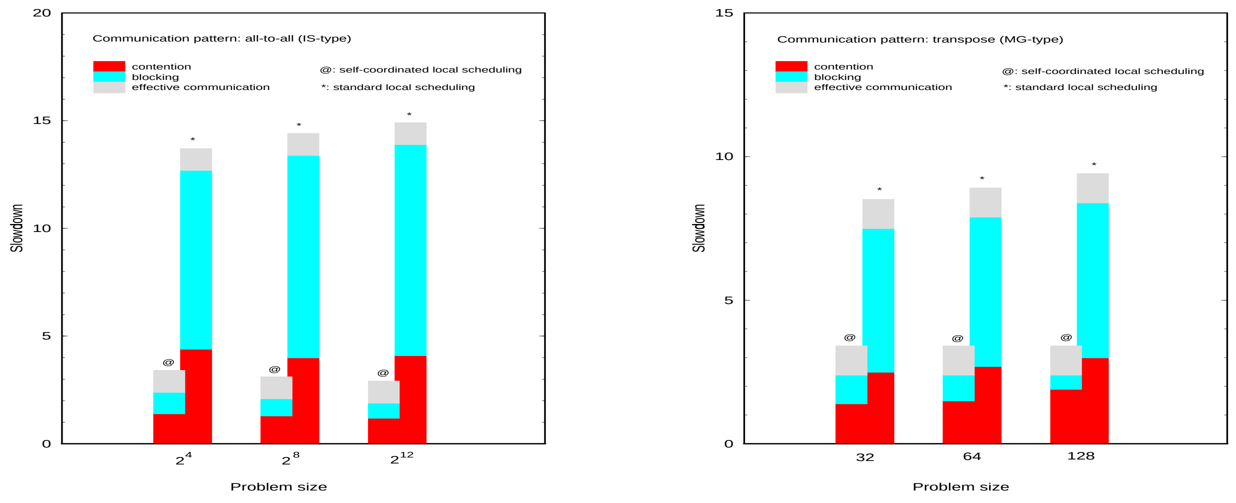

Figure 4: Comparisons of communication performance of the IS and MG programs between the self-coordinated local scheduling and the standard round-robin local scheduling.

4 Summary and current work

Our performance evaluation results indicate that to effectively schedule application programs on a heterogeneous NOW, we should consider both architecture- and application- specific information as well as 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 abstracted application program model, we propose the self-coordinated local scheduling algorithm. This method coordinates the execution pace of a parallel job in the computational phase using the local scheduler based on the co-scheduling principle, and completes the communication/synchronization phase as soon and continuous as possible. Direct simulation results show its effectiveness. The power preservation in each workstation makes a fair power distribution and makes the coordination by local scheduling possible.

There are three issues we have not addressed yet for the self-coordinated local scheduling technique in this paper. First, the current version of the scheduling technique is designed in a standard round-robin time-sharing environment. We are implementing the technique in a practical operating system, the Solaris, where processes are classified into three classes: *real-time*, *time-sharing*, and *interactive*, and a priority-based method is used to schedule them. The second one is related to application program information. The parallel programming model we used may not be practical for some applications, because parallel tasks in some applications may not have equal sizes, and the communications and local computations may frequently interact to each other. We are currently including new functions into this local scheduling method in order to effectively schedule a wider range of application programs. The third issue is related to networks and communications. The communication management is network dependent. If high speed switched networks are used, the management of the communication/synchronization phase may not be highly demanding. We are including network and communication parameters in the scheduling algorithm based on our experiments on different types of networks, such as high speed Ethernet, ATM networks and Myrinet networks.

References

[1] R. H. Arpaci, 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.

[2] D. Bailey et al., "The NAS parallel benchmarks", *International Journal of Supercomputer Applications*, Vol. 5, No. 3, Fall, 1991, pp. 63-73.

[3] M.-S. Chen and K. G. Shin, "Subcube allocation and task migration in hypercube multiprocessor", *IEEE Transactions on Computers*, Vol. C-39, No. 9, 1990, pp. 1146-1153.

[4] A. C. Dusseau, R. H. Arpaci and D. E. Culler, *Effective local scheduling of parallel jobs*, Technical Report, Computer Science Division, University of California, Berkeley, December, 1995.

[5] A. Gupta, A. Tucker, and S. Urushibara, "The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications", *Proceedings of the ACM SIGMETRICS Conference*, May, 1991, pp. 120-132.

[6] C. M. McCann, *Processor allocation policies for message-passing parallel computers*, Ph.D. Dissertation, University of Washington, 1994.

[7] J. Ousterhout, "Scheduling techniques for concurrent systems", *Proceedings of the 3rd International Conference on Distributed Computing Systems*, October, 1982, pp. 22-30.

[8] A. Tucker and A. Gupta, "Process control and scheduling issues for multiprogrammed shared-memory multiprocessors", *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, 1989, pp. 159-166.

[9] V. S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, Vol. 2, No. 4, pp. 315-339.

[10] J. Zahorjan and C. McCann, "Processor scheduling in shared-memory multiprocessors", *Proceedings of ACM SIGMETRICS Conference*, May, 1990, pp. 214-225.

[11] X. Zhang and Y. Yan, "Modeling and characterizing parallel computing performance on heterogeneous NOW", *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing*, IEEE Computer Society Press, October, 1995, pp. 25-34.

[12] X. Zhang, Y. Yan, and K. He, "Latency metric: an experimental method for measuring and evaluating program and architecture scalability", *Journal of Parallel and Distributed Computing*, Vol. 22, No. 3, 1994, pp. 392-410.