Effectively Scheduling Parallel Tasks and Communications on Networks of Workstations *

Xing Du[1], Yingfei Dong[2], and Xiaodong Zhang[1]

[1] Department of Computer Science, College of William and Mary
Williamsburg, VA 23187, USA
[2] Department of Computer Science, University of Minnesota
Minneapolis, MN 55455, USA

Abstract. Coordinating parallel tasks and minimizing communication delays are two important issues for the design of scheduling policies on networks of workstations. We address the two issues by presenting a scheduling scheme, called *self-coordinated local scheduling*. It consists of two parts: computation scheduling and communication scheduling. The computation scheduler on each workstation schedules its parallel task and local user jobs independently based on a static power preservation in that workstation so that parallel tasks on different workstations are executed at the same pace to achieve a global coordination. To minimize the communication latency, each local scheduler uses a non-preemptive strategy to try to make communication activities complete as soon as possible. The scheduling scheme is evaluated by simulating the execution of four NAS benchmark programs, and comparing it with a Unix based scheduling policy and the co-scheduling policy. Our experiments show that the power preservation in each workstation guarantees the performance of both parallel and local jobs. Furthermore, the communication delay is reduced significantly by the communication scheduling compared with a spinning based scheme and a Unix based scheme.

1 Introduction

Networks of Workstations (NOWs) are cost-effective platforms for parallel computations. Effectively scheduling parallel jobs on NOWs is very important. However, most existing scheduling policies on multiprocessor/multicomputer systems (e.g. [3] [7]) are not applicable because of the heterogeneous and non-dedicated features of NOWs [1]. In order to effectively manage the interaction between parallel jobs and user jobs and provide good performance to both kinds of jobs, two issues must be well addressed: how to coordinate the execution of tasks of a parallel job on different workstations (inter-processor coordination), and how

* 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. Xing Du is on leave from Computer Science Department, Nanjing University, Nanjing, P.R.China.

to manage the interaction between a parallel job and local user jobs on a workstation (intra-processor coordination). Co-scheduling policy [6] generally results in good parallel job performance [5] on multiprocessor systems, and is a good policy for inter-processor coordination. However it is hard to be effectively implemented on NOWs where the communication overheads are high. In addition, no intra-processor coordination is provided.

We propose a scheduling scheme, called *self-coordinated local scheduling* based on the principle of co-scheduling. By preserving certain amount of computing power on each workstation, each local scheduler schedules independently the parallel task and local user jobs on its workstation. The coordination between parallel tasks running on different workstations is achieved, and the performance of both parallel and local jobs is guaranteed. To minimize the communication latency, the scheme uses a non-preemptive strategy to expedite the communication. Simulation results of four NAS benchmark programs indicate that it is an effective approach to scheduling parallel and local user jobs on NOWs.

2 Scheduling Scheme

We propose a scheduling scheme, called *self-coordinated local scheduling* for bulk synchronous applications. It is composed of two parts: computation scheduling which is scheduling parallel tasks when they are in computational phases, and the other, communication scheduling for scheduling tasks when they are in communication phases. A key issue in computation scheduling is to preserve a portion of power in each workstation for parallel tasks. We define the power weight of a workstation as 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 $T(App, M_i)$ 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:

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

where m is the number of workstations in a NOW. (For detailed information about the power weight, the interested reader may refer to [8].) Each workstation first calculates its own capable power weight for parallel jobs:

$$\rho_i = W_i(1 - R_{kernel}(i) - R_{user}(i)) \tag{2}$$

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 local user jobs in the workstation. We define the free power weight on workstation i, $(i = 1, ..., m)$, as:

$$F_i = 1 - R_{kernel}(i) - R_{user}(i). \tag{3}$$

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:

$$\rho = min_{i=1}^{m}\rho_i. \tag{4}$$

The equivalent power preservation in each workstation for parallel tasks allows us to "simulate" co-scheduling in a virtual homogeneous system. The remaining percentage of the power in a workstation can be used for local user jobs.

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. 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 relatively slowest workstation is

$$t_s = \frac{TS}{Pow(s) \times F_s}, \tag{5}$$

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

$$t_i = \frac{TS}{Pow(i) \times F_i}, \tag{6}$$

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

$$N_{slice}(s) = \frac{t_s}{\delta_s}, \tag{7}$$

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

$$N_{slice}(i) = \frac{t_i}{\delta_i}, \tag{8}$$

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 (5), (6), (7) and (8) 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 (5) and (6), we further obtain

$$\frac{t_s}{t_i} = \frac{Pow(i) \times F_i}{Pow(s) \times F_s} \tag{9}$$

The time quantum in (9) is architecture dependent rather than application program dependent.

The scheduling scheme in the computational phase is as follows:

1. Determine ρ_i, the available power weight for parallel jobs on workstation i based on its local user's decision.
2. Broadcast $Pow(i)$ and F_i to other local schedulers.
3. Receive Pow and F from all other local schedulers.
4. Decide the minimum available power weight based on (4), and the relatively slowest workstation s.
5. The local scheduler in workstation i calculates its pace to assign a time slice to its parallel task by (9).

The communication phase is another essential section of parallel jobs. It has different requirements for schedulers. A message-passing can only be performed when it obtains both the CPU and the network. If a communicating task is given more chance to obtain the CPU, it would have a better chance to complete the communication phase as soon as possible. This would also increase the possibility of overlapping this communicating task with computational tasks on other workstations, and decrease the possibility of network contention. Motivated by this, we use such a scheduling scheme for communications:

1. The local scheduler is notified that the parallel task has been in communication phase (becomes a communicating task).
2. Set a larger time slice for this communicating task.
3. If the CPU is serving for a local user job, it does nothing until the local job finishes its time slice.
4. When the communicating task acquires the CPU, if the network is available, the above larger time slice is given to it, otherwise, the scheduler gives a time slice to a local job immediately.
5. The local scheduler is notified that the communicating task has finished its communication.
6. The local scheduler resumes its power preservation scheme.

Thus, the scheduler gives a larger time slice to a parallel task to complete the communication if the network is available, otherwise it is switched to process local user jobs. This would reduce unnecessary context switches and network contention. In the scheme, the CPU is switched to a local user job rather than to spin when the network is not available for the communication. This is because 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 deadlocks.

3 Simulation Methodology

We designed and developed a simulator to perform event-driven simulation. The structure of the simulator is illustrated in Figure 1. The effect of computation scheduling was event-driven simulated. The communication effect was evaluated by a network simulator which simulated an Ethernet of 10Mbps bandwidth. We selected Ethernet because it is a popular network used to connect workstations.

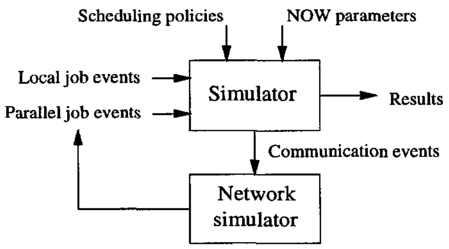

Fig. 1. The simulator structure.

The simulated heterogeneous NOW consisted of 7 types of Sun SPARCstations with different computing powers. We measured the power weight of each workstation (Table 1) by running 4 benchmark programs discussed later.

Table 1. The average power weight of 7 types 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

We selected four programs from the NAS parallel benchmarks [2]: EP (Embarrassing Parallel), MG (Multigrid), IS (Integer Sort), and LU (LU decomposition). All of them follow the bulk synchronous model. The communication patterns of the 4 programs were classified into three types: *all-to-all* (IS, EP), *neighbors* (LU), and *transpose* (MG). The parallel job events were characterized by their computation sizes, communication patterns, the number of bytes to send in each communication, communication starting time, process arrival time, etc. The local workloads were only characterized by their starting time and computation sizes.

The scheduling policies in the simulator included a local scheduling scheme based on Unix (System V Release 4), co-scheduling using the matrix scheme in [6], and the self-coordinated local scheduling.

4 Performance Evaluation

The performance evaluation was done by simulating the execution of the 4 programs on NOWs with different workstation combinations. For space limitation, we only present results on 4 workstations of type B, C, F and G here. In the simulation, the following system parameters were used: the time slice was set 0.1 second; and a context switch took 200 μs. EP was iterated 10 times to increase its synchronization activities.

Table 2. The effect of varying the number of local jobs on the slowdowns of four programs using the self-coordinated local scheduling (*coordinated*) and local scheduling (*local*) in comparisons with co-scheduling.

# of local jobs	EP		IS		MG		LU	
	coordinated	*local*	*coordinated*	*local*	*coordinated*	*local*	*coordinated*	*local*
1	1.1	1.31	1.32	2.05	1.22	1.89	1.34	2.0
2	1.1	1.92	1.34	2.65	1.23	2.78	1.35	3.08
3	1.1	2.56	1.36	3.51	1.25	3.84	1.37	4.19
4	1.2	3.14	1.38	4.12	1.27	4.27	1.39	5.12
5	1.2	3.90	1.39	4.98	1.28	5.01	1.40	6.03

Table 2 presents the effect of varying the number of local user jobs on slowdown factors of the self-coordinated local scheduling and standard local scheduling. The slowdown is relative to the execution times of the four programs using co-scheduling on the corresponding dedicated NOW. The execution time of EP using self-coordinated local scheduling was very close to that using co-scheduling, differing by a factor of up to 1.2. The execution times of IS, MG and LU by self-coordinated local scheduling increased 22% to 34% in comparison with the times of co-scheduling when there was one local job. Besides scheduling skew and context switch overhead, there was another reason for the slowdown. The computation size of the three programs were dynamically changed as the programs proceeded. When the execution time of a computation size was close to a time slice, the execution pace was not well coordinated, because the time frame in the relatively slowest workstation for the task, t_s, in (5) became very small, and the scheduling became difficult in a tiny time space. However, with the increase of the number of local jobs, the slowdown of self-coordinated scheduling changed very slightly. In other words, the performance of parallel jobs was affected slightly by the change of local workload, and was guaranteed by the scheduling scheme. Meanwhile, the performance degradation of each local job was expected by the local user because he agreed to denote that amount of his workstation power to parallel jobs (see Step (1) in the scheme) and added the local workload by himself.

In contrast, when using local scheduling, the performance of parallel programs degraded significantly. The degradation increased almost proportionally with the increase of the number of local jobs. For example, the performance of IS decreased by a factor of 498% when 5 local jobs were executed with it.

Table 3. The breakdown of the execution times for EP and IS.

	EP				IS			
#	Comput.	Comm.	Sync.	Switch	Comput.	Comm.	Sync.	Switch
1	17.0	0.0007	2.88	0.06	28.3	6.42	4.29	0.09
2	18.6	0.0007	1.52	0.07	32.5	6.42	2.79	0.13
3	19.2	0.0007	0.89	0.08	33.8	6.42	1.37	0.15
4	19.9	0.0007	0.23	0.10	34.9	6.42	0.38	0.17
5	19.9	0.0007	0.23	0.10	34.9	6.42	0.37	0.18

Table 4. The breakdown of the execution times for MG and LU.

	MG				LU			
#	Comput.	Comm.	Sync.	Switch	Comput.	Comm.	Sync.	Switch
1	64.3	2.5	8.9	0.2	61.1	1.2	8.5	0.2
2	72.6	2.5	5.9	0.3	71.8	1.2	6.1	0.3
3	75.2	2.5	3.0	0.3	76.1	1.2	3.2	0.3
4	77.4	2.5	1.1	0.4	78.5	1.2	1.0	0.4
5	77.4	2.5	1.1	0.4	78.5	1.2	1.0	0.4

Table 3 and 4 list the breakdown of the execution times in simulated clock ticks for the four programs when the number of local jobs changes. The execution time includes computation time, communication time, synchronization time, and context switch time. The communication time includes the startup time (software overhead) and message transmission time (network latency). The synchronization time is that a workstation spends on waiting at a synchronization point in a program or in a synchronous send/receive protocol.

We further evaluate the efficiency of the communication scheduling, and compare it with two other policies. One is local scheduling, which means no special policy is used for scheduling communicating tasks. The other is either using the network if it is available or spinning for its turn. We call this policy, spinning scheduling. This policy may result in low system utilization, and deadlock if multiple parallel jobs are executed. Two communication patterns were studied, one is *all-to-all* and the other is *transpose* which is from IS and MG respectively. Table 5 presents the communication delays for two types of patterns changing with the problem sizes. For both patterns, our communication scheduling performed better than any of the two other schemes for different problem sizes.

Table 5. The communication delay of two types of communication patterns scheduled by the *Spinning* scheduling, the *Self-coordinated* scheduling and the *Local* scheduling.

Communication pattern	Problem size	Communication delay		
		Spinning	*Coordinated*	*Local*
all-to-all	2^4	6.5	5.2	14.9
	2^8	4.0	4.0	13.1
	2^{12}	6.5	4.5	11.6
transpose	32	8.0	6.3	10.4
	64	17.4	11.9	20.8
	128	35.0	12.6	18.1

5 Current Work

We propose a scheduling scheme for scheduling parallel tasks on non-dedicated NOWs. It guarantees the performance of both parallel and local jobs. The simulation results show its effectiveness. We are currently studying a way to preserve power in UNIX and how to effectively schedule a wider range of applications, and investigating adapting the communication scheduling in high speed networks.

References

1. R. H. Arpaci et al.: The interaction of parallel and sequential workloads on a network of workstations. Proceedings of ACM SIGMETRICS Conference, (1995)
2. D. Bailey et al.: The NAS parallel benchmarks. International Journal of Supercomputer Applications. **5**(3) (1991) 63-73
3. M.-S. Chen and K. G. Shin: Subcube allocation and task migration in hypercube multiprocessor. IEEE Transactions on Computers. **C-39**(9) (1990) 1146-1153
4. A. C. Dusseau, R. H. Arpaci and D. E. Culler: Effective distributed scheduling of parallel workloads. Proceedings of ACM SIGMETRICS Conference, (1996)
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 ACM SIGMETRICS Conference. (1991) 120-132
6. J. Ousterhout: Scheduling techniques for concurrent systems. Proceedings of the 3rd International Conference on Distributed Computing Systems. October, (1982) 22-30
7. 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) 159-166
8. 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) 25-34