A Scheduling Approach Considering Local Tasks in the Computational Grid

Zhan Gao, Siwei Luo and Ding Ding

School of Computer and Information Technology
Beijing Jiao tong University
bjtugaozhan@gmail.com

Abstract

Task scheduling under a grid environment is an important research area, on which much attention has been paid. However, either in the meta-task scheduling problems or DAG (Direct Acyclic Graph) scheduling problems, it is usually assumed that tasks are submitted to dedicated hosts and that these tasks are processed in FIFO (First In First Out) order. This is not practical in a grid, in which a host may be shared between grid users and its owner and local tasks, which belong to resource owners, may compete with grid tasks for the hosts. EBGSA (Estimation Based Grid Scheduling Approach) is proposed, which allows for the simultaneous processing of grid tasks and local tasks. In EBGSA we use history information about the execution of tasks to estimate the performance of non-dedicated hosts. Two heuristic scheduling algorithms, MCT (Minimum Completion Time) and Min-min are selected to perform the simulation experiment. Both experiments obtain a smaller make span, proving EBGSA feasible for grid task scheduling.

1. Introduction

Grid [1] is a kind of distributed computing infrastructure, which allows large scale resource sharing and system integration. It is based on networks and able to enable large-scale aggregating and sharing of computational, data, sensors and other resources across institutional boundaries. As a heterogeneous computing system, the task scheduling strategy directly influences the performance of gird applications. And there are now many researches on how to schedule gird tasks properly in order to achieve high performance.

Under a grid environment, tasks can be classified as independent tasks, which have no communications between each other, and communication depended tasks. As for independent tasks, mapping (matching and scheduling) heuristics can be grouped into two categories: on-line mode and batch mode mapping. In the on-line mode, a task is scheduled as soon as it arrives at the mapper. In the batch mode, tasks are collected into a set, which is called a meta-task, and mapped at mapping events. And there are many mapping heuristics for independent tasks such as MCT (Minimum Completion Time), MET (Minimum Execution Time), SA, Min-min, Min-max, Sufferage [2], etc. It is common to treat communication based tasks as a DAG. There are many heuristics for DAG scheduling including the list scheduling [3],[4] the critical path heuristics [5],[6] the clustering algorithms [7],[8] the guided search algorithms [9],[10] and the duplication based algorithms [11],[12]. Either in the scheduling of independent tasks or DAGs, it is assumed that each task has exclusive use of the machine and that the machine will execute its tasks in FIFO order. However, this assumption is very unpractical in a grid environment. Though the grid is aiming at coordinated resource sharing, this sharing is often conditional: resource owners make resources available, subject to constraints on when, where and what can be done [13]. When a grid user has submitted his task to the grid task manager, the task will enter a task queue maintained by the task managing organization. After that, the grid scheduling module will select proper tasks from the task queue and allocate suitable resources to these tasks to make them run. There are usually more than one task being scheduled to the same resource and these tasks have to enter a local task queue of the resource and be scheduled by the local scheduler before their running. So after submitted to the grid, a task will be scheduled by the grid scheduler and subsequently the local scheduler before its completion unless it's migrated or killed by the grid. Due to the site autonomy of the grid, the task manager may have no control of and even no information about the local schedulers. For a certain grid resource, there are usually many tasks on it, including the gird tasks and the resource owner's local tasks. And the local scheduler may schedule all these tasks in a parallel manner, which makes the execution unrecurrable. So the grid task scheduling can hardly promise the realization of its target. Though there are many researches in the grid scheduling and many scheduling algorithms are proposed, little work involves this issue. In this paper, we propose EBGSA, which allows for the simultaneous processing of grid tasks and local tasks. We also apply EBGSA to MCT algorithm and Min-min algorithm. The result of the experiment shows that a smaller makespan is obtained using EBGSA.

The rest of this paper is structured as follows. The next section gives the background of grid scheduling problem and also presents MCT and Min-min. In section 3, we propose EBGSA and apply it to MCT and Min-min. In section 4, the simulation experiment is discussed. The last section includes the conclusion and future work.

2. Problem Definition

2.1. Performance Metrics

For simplicity, in this paper we only consider the scheduling of independent tasks and use throughput as the only scheduling criterion thought there are other criterions, for instance the quality of service. The expected execution time e_{ij} is defined as the amount of time taken by machine m_j to execute task t_i, given m_j has no load when t_i is assigned. The expected completion time c_{ij} of task t_i on machine m_j is defined as the wall-clock time when m_j completes t_i. Let m be the total number of the machines in the grid and K the total number of tasks to be scheduled. Let the arrival time of task t_i be a_i, and let the begin time of t_i be b_i. From the above definitions, $c_{ij} = b_i + e_{ij}$. Let c_i be c_{ij}, where machine j is allocated to execute task i. The makespan for the complete schedule is then defined as $\max_{t_i \in K}(c_i)$ [14]. Makespan is a measurement of the throughput of the computational grid.

2.2. Problems with Existing Algorithms

Task scheduling is a well-known NP-complete problem if throughput is the optimization criterion [15] and various scheduling heuristics are proposed both for independent and communication based tasks. Most of these heuristics are based on the following two assumptions. First, the expected execution time e_{ij} is deterministic and will not vary with time. Second, each task has exclusive use of the machine. As we discussed in section 1, this is not the case actually. This *inconsistency* is unavoidable and has a great influence on many heuristics. In order to illustrate this influence, consider MCT and Min-min algorithms.

Minimum Completion Time (MCT) Algorithm

The MCT heuristic assigns each task to the machine that will finish it earliest. The algorithm is described below:

(1) for all the tasks t_i **(in an arbitrary order)**
(2) for all machines m_j **in the gird**
(3) $c_{ij} = e_{ij} + r_j$
(4) find machine m_p **which will finish** t_i **earliest**
(5) schedule t_i **to** m_p

Min-min Algorithm

Min-min begins by scheduling the task that changes the expected machine ready time status by the least amount that any assignment could. If two tasks compete for a particular machine m_j, Min-min will select the one that changes the ready time r_j of machine m_j less and assigns it to m_j. The algorithm is described below:

(1) for all tasks t_i **in meta-task** M **(in an arbitrary order)**
(2) for all machines m_j **in the grid**
(3) $c_{ij} = e_{ij} + r_j$
(4) do until M **is empty**
(5) for each task in M **find the earliest completion time and the corresponding machine that obtains it**
(6) find the task t_p **with the minimum earliest completion time**
(7) assign task t_p **to the machine** m_q **that gives the earliest completion time**
(8) delete task t_p **from** M
(9) update r_q
(10) update all c_{iq} **for all** i

Now, let's consider two scheduling examples of MCT and Min-min. Table 1 gives a scenario in which four tasks will be scheduled onto two machines using MCT algorithm. Table 2 gives a scenario in which four tasks will be scheduled onto two machines using Min-min algorithm.

Table 1. A scenario for MCT scheduling

	t_0	t_1	t_2	t_3
m_0	2	4	5	4
m_1	3	8	7	3

Table 2. A scenario for Min-min scheduling

	t_0	t_1	t_2	t_3
m_0	5	1	6	4
m_1	6	2	7	4

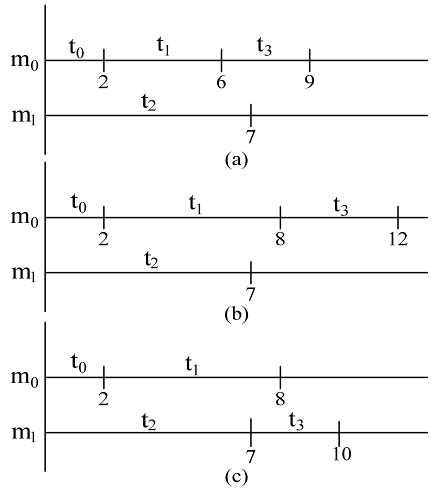

Figure 1. Different schedules made by MCT: (a) the schedule on dedicated machines (b) the schedule on non-dedicated machines (c) the schedule on non-dedicated machines with prediction

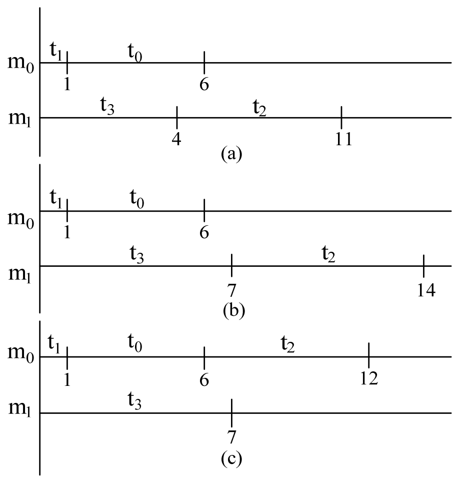

Figure 2. Different schedules made by Min-min: (a) the schedule on dedicated machines (b) the schedule on non-dedicated machines (c) the schedule on non-dedicated machines with prediction

The scheduling results are illustrated in figure 1 and figure 2 respectively. We suppose that the four tasks, t_0, t_1, t_2 and t_3, in table 1 will be scheduled in the sequence of increasing subscript. If machine m_0 and m_1 are both dedicated, MCT will make a schedule as illustrated in figure 1-(a), which results in a makespan of 9. As we discussed before, it's unpractical to expect all the machines in a grid to be dedicated. Suppose machine m_0 will execute grid task t_1 and *local tasks* together and this will delay e_{10}. Let the actual e_{10} change from 4 to 6. If the scheduler doesn't know this change, it will persist on the former schedule, which leads to a makespan of 12 as illustrated in figure 1-(b). Though the change of e_{10} is unavoidable due to the site autonomy, if the scheduling strategy can predict it in advance, a better schedule will be made. In figure 1-(c), when task t_3 arrives, the scheduler already predicts that the completion time of t_1 will change from 4 to 6 and it will assign t_3 to m_1 in stead of m_0 according to MCT, which results in a smaller makespan of 10 compared to 12. Table 2 shows a scenario of Min-min scheduling. Figure 2-(a) and figure 2-(b) are the schedules before and after t_{31} changes from 4 to 7. We can see that the makespan can be cut from 14 (in figure 2-(b)) to 12 (in figure 2-(c)) if the scheduler can predict the change of t_{31}.

3. EBGSA

3.1. Applying EBGSA to MCT and Min-min

In this paper, we propose an Estimation Based Grid Scheduling Approach (EBGSA), which allows for the simultaneous processing of grid tasks and local tasks. In EBGSA, we treat every expected execution time as a random variable in stead of a predetermined constant. By estimating the value of each random variable, the scheduler can make a better schedule, which takes into account the actual resource status in the grid. Consider the two examples in the previous section, we apply EBGSA to MCT and Min-min and modify them to be EMCT (Estimating MCT) and EMin-min (Estimating Min-min). The two algorithms are described below:

EMCT Algorithm

(1) **for all the tasks** t_i **(in an arbitrary order)**
(2) **for all machines** m_j **in the gird**
(3) $ec_{ij} = ee_{ij} + er_{ij}$
(4) **find machine** m_p **which will finish** t_i **earliest**
(5) **schedule** t_i **to** m_p

EMin-min Algorithm

(1) **for all tasks** t_i **in meta-task** M **(in an arbitrary order)**
(2) **for all machines** m_j **in the grid**
(3) $ec_{ij} = ee_{ij} + er_j$
(4) **do until** M **is empty**
(5) **for each task in** M **find the earliest completion time and the corresponding machine that obtains it**
(6) **find the task** t_p **with the minimum earliest completion time**
(7) **assign task** t_p **to the machine** m_q **that gives the earliest completion time**
(8) **delete task** t_p **from** M
(9) **update** er_q
(10) **update all** ec_{iq} **for all** i

Note that line 3 of EMCT algorithm differs from that of MCT. In EMCT algorithm we substitute the estimation of c_{ij}, e_{ij} and r_j, namely ec_{ij}, ee_{ij} and er_j, for c_{ij}, e_{ij} and r_j. And Min-min algorithm is modified in the same way to produce EMin-min algorithm.

3.2. Estimating the Time Variable

The key issue of EBGSA is how to accurately estimate the time variable, ee_{ij}. However, because the distribution of random variable ee_{ij} is unknown, it's impossible to form a definite formula of ee_{ij}. So we should try to approximate it. One way people may easily think of is to figure out the value of ee_{ij} by monitoring the resource status, such as system load. But this method has two shortcomings. First, it isn't fit for the grid environment. Usually the resource owner's local tasks is prior to the grid tasks, so a grid resource will not process the grid tasks assigned to it until it has finished all the local tasks or it is released by all the local tasks. Even different grid tasks have different priorities. So, we can not derive the value of ee_{ij} accurately only from the information of system load. Second, it will cost a lot of time if we have to detect the resource status before scheduling every task.

In EBGSA, we statistically estimate the random variable ee_{ij} from the past observations. The relation between e_{ij} and ee_{ij} can be expressed as (1).

$$ee_{ij} = e_{ij} + \sigma_{ij} \quad (1)$$

In (1), σ_{ij} is the additional amount of time needed by machine m_j to finish task t_i, caused by the execution of local tasks. Let $\eta_{ij} = \frac{ee_{ij}}{e_{ij}}$ $(\eta \geq 1)$. Suppose that before task

t_i, which is assigned to machine m_j, is executed, m_j has already accomplished m tasks. We use (2) to estimate η_{ij}, where η_{pj} refers to the p_{th} task accomplished by m_j.

$$\eta_{ij} = \sum_{p=1}^{m} x_p \eta_{pj}, \quad \sum_{p=1}^{m} x_p = 1 \quad (2)$$

In (2) x_p is the weight of η_{pj}, and usually the bigger p is, the more proportion x_p will take up. That means the execution of the latest task will influence the estimation the next execution most. After the estimation of η_{ij}, we can derive ee_{ij} and er_j from (3) and (4) respectively.

$$ee_{ij} = e_{ij}\eta_{ij} \quad (3)$$

$$er_j = \sum_{1 \leq i \leq k} ec_{ij} \quad (4)$$

In (4), k stands for the number of tasks that machine m_j allows to run simultaneous and we assume that all the k tasks assigned to m_j start from time 0.

4. Simulation

4.1. Simulation Environment

In our simulation experiment we scheduler a meta-task of 200 tasks onto 4 machines. The expected execution time of task t_i on machine m_j, namely e_{ij}, varies from 1 to 50.

In order to simulate the concurrent running of grid tasks and local tasks, we make use of Java MultiThreading Programming [16]. In our experiment, for simplicity we generate a Java thread object, *localTaskThread*, with a high priority, *Thread.MAX_PRIORITY*, to present a local task which is running on a certain machine (Of course, it ispossible to generate more than one thread with *Thread.MAX_PRIORITY* to simulate more than one local task running concurrently). And we also generate three Java thread objects, *gridTaskThread*, with a low priority, *Thread.MIN_PRIORITY*, to present three gird tasks running right on the previously mentioned machine concurrently. Here, we suppose that a machine will allow at most three grid tasks running concurrently on it. The experiment is performed using JDK1.5.0 and on the platform of Windows XP. According to the features of Java and Windows XP scheduling strategy, the JVM (Java Virtual Machine) scheduler will run the thread with the highest priority first. When all the threads with a high priority are dead or blocked, the threads with a low priority are able to get the opportunity to run. In addition, the JVM running on Windows XP will allocate amount of CPU cycles to each of the threads with the same priority and schedule them in turn. In our experiment, a machine will prefers a *localTaskThread* to a *gridTaskThread* and we will call the *sleep()* method at intervals to let a *localTaskThread* sleep for a certain period so that the *gridTaskThreads* are able to be scheduled. By doing this, we simulate an actual computation grid environment described in section 1.

4.2. Simulation Procedures

The specific simulation procedures are described as follows.

1. Schedule the previously mentioned 200 tasks on to 4 different machines using MCT algorithm.
2. Find the machine m_{Mct} which produces the makespan and denote the corresponding schedule of tasks as s_{Mct}.
3. Suppose m_{Mct} to be non-dedicated. Generate on it a *localTaskThread* and three *gridTaskThreads* for every three tasks in s_{Mct} as described in the previous subsection.
4. Execute s_{Mct} on m_{Mct} and denote the finish time as $makespan_{1.}$
5. Schedule the 200 tasks again using EMCT algorithm and denote the corresponding makespan as $makespan_2$.
6. Compare $makespan_1$ with $makespan_2$.
7. Substitute Min-min for MCT and repeat steps 1 to 6.

In step 1, we treat each of the 200 tasks as a Java Thread object, initialized with an expected execution time e_{ij} between 1 and 50. For task t_i assigned to machine m_j, when the total time of t_i's running on m_j reaches e_{ij} we will kill the thread. So the period from the time when t_i is generated to the time when it is dead is the actual execution time, which is denoted as ae_{ij}. And in step 5, the key of EMCT is how to make the estimation of ee_{ij} as close to ae_{ij} as possible. Here we use the observation of the last meta-task to estimate the next meta-task. Assume the non-dedicated machine is m_j. We generate 50 tasks and execute them on m_j. Let $\eta_{pj} = \frac{ae_{pj}}{e_{pj}}$, $x_p = \frac{1}{50}$. According to (2) and (3), for task t_i in the next meta-task,

$$ee_{ij} = \frac{1}{50} e_{ij} \sum_{p=1}^{50} \frac{ae_{pj}}{e_{pj}}.$$

4.3. The Result and Discussion

The scheduling results of Min-min, EMin-min, MCT and EMCT are illustrated in figure 3 and figure 4. From figure 3, we can see that the actual makespan mp_2 (568.7) is much larger than that expected by Min-min algorithm, mp_1 (512.2). This great increase of makespan is caused by the scheduler's neglect of the non-dedication of machine m_{Min} when using Min-min algorithm. When the scheduler considers scheduling task t_i to machine m_{Min} has the earlier completion time than any other assignment, it will Assign t_i to m_{Min} according to Min-min algorithm. In fact, due to the simultaneous running of local tasks as well as other gird tasks on m_{Min}, t_i may not be the task with the earliest completion time. Since our EMin-min algorithm uses the estimation of execution time ee_{ij} of grid tasks instead of the predetermined expected execution time e_{ij}, it can make a schedule more suitable to the actual grid environment than Min-min. This can explain why mp_3 (537.2) decreases by 5.5% compared to mp_2 (568.7). The experiment of MCT and EMCT has a similar result. We can see that in figure 4, mp_3 (579.3) is 5.2% less than mp_2 (610.8). Note that, in our experiment we let x_p be $\frac{1}{50}$, which makes each task of the meta-task of 50 tasks has the same weight. This is because since we use the observation of the last meta-task to estimate ee_{ij} of the next meta-task, every task in the last meta-task has the same influence on the estimation. If the scheduling heuristic belongs to the on-line mode instead of the batch mode, we can give a larger weight to the more recently finished task, making it have more influential on the estimation of the next task. Since it's not the point of this paper, we do not discuss this issue here.

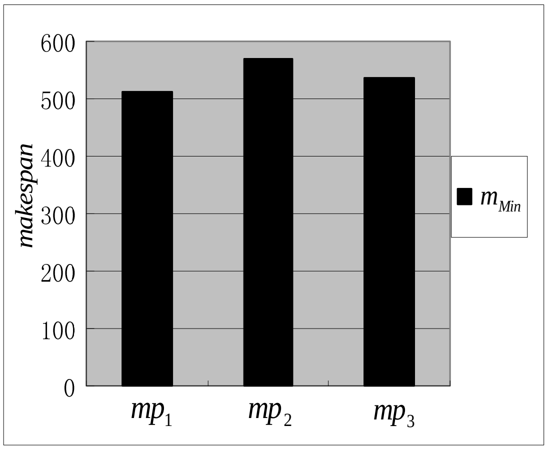

Figure 3. Different makespans made by Min-min and EMin-min: mp_1 is the makespan of Min-min; mp_2 is the actual makespan; mp_3 is the makespan of EMin-min

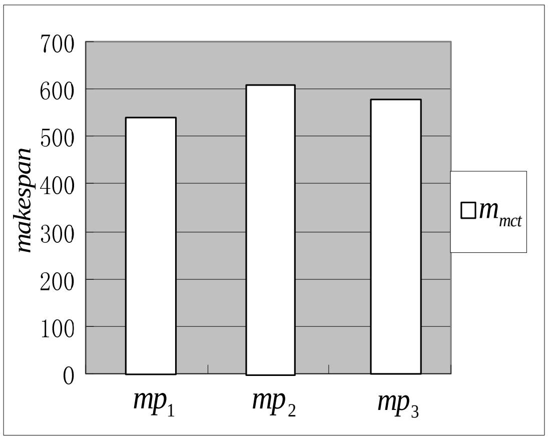

Figure 4. Different makespans made by MCT and EMCT: mp_1 is the makespan of MCT; mp_2 is the actual makespan; mp_3 is the makespan of EMCT

5. Conclusion

Most grid scheduling heuristics assume the precondition that all the machines in the grid are dedicated and idle and that every grid task has exclusive use of each machine. In fact, due to the site-autonomy feature of grid, a machine may simultaneously execute local tasks and grid tasks. Thus, the scheduler can not promise that its scheduling goal can be achieved. In this paper EBGSA is presented, which allows for the simultaneous processing of grid tasks and local tasks. We apply EBGSA to MCT and Min-min algorithms and the simulation experiment shows that EMCT and EMin-min outperform MCT and Min-min respectively in the measurement of makespan.

While the MCT and Min-min algorithms are considered in this paper, we should note that EBGSA is applicable to any other scheduling heuristic where the predetermined execution time and completion time are used. In this paper, to estimate the execution time of a grid task we use the linear estimation mode which is simple but not very adaptable to the dynamic grid environment. The future work focuses on developing a new estimation mode for our EBGSA, which can estimate the execution time of grid tasks more accurate under the dynamic grid environment.

References

[1] I. Foster, C. Kesselman, “The Grid : Blueprint for a New Computing Infrastructure”,Morgan Kaufmann Publishers, 1999.
[2] Muthucumaru Maheswaran, Shoukat Ali, “Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Syetems”, Proceedings of the 8th IEEE Heterogeneous Computing Workshop (HCW' 99), IEEE Computer Society Press, 1999.
[3] Martin Grajcar, “Genetic list scheduling algorithm for scheduling and allocation on a loosely coupled heterogeneous multiprocessor system”, Proceedings of the 36th Design Automation Conference, pp. 280-285, 1999.
[4] Wai-Yip Chan, Chi-Kwong Li, “Scheduling tasks in DAG to heterogeneous processor system”, Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing, pp. 27-31, January 1998.
[5] Haluk Topcuoglu, Salim Hariri and Min-You Wu, “Task scheduling algorithms for heterogeneous processors”, Proceedings of the Eighth Heterogeneous Computing Workshop, pp. 3-14, April 1999.
[6] GyungLeen Park, Behrooz Shirazi, Jeff Marquis and Hyunseung, “Decisive path scheduling: a new list scheduling method”, Proceedings of the 1997 International Conference on Parallel Processing, pp. 472-480, August 1997.
[7] Wai-Yip Chan, Chi-Kwong Li, “Heterogeneous Dominant Sequence Cluster (HDSC): a low complexity heterogeneous scheduling algorithm”, IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, pp. 956-959, August 1997.
[8] Zhen Liu, “Scheduling of random task graphs on parallel processors”, Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS’95), pp. 143-147, January 1995.
[9] Sung-Ho Woo, Sung-Bong Yang, Shin-Dug Kim and Tack-Don Han, “Task scheduling in distributed computing systems with a genetic algorithm”, High Performance Computing on the Information Superhighway (HPC Asia’97), pp. 301-305, May 1997.
[10] E.S.H. Hou, N. Ansari and Hong Ren, “A genetic algorithm for multiprocessor scheduling”, IEEE Transactions on Parallel and Distributed Systems, Vol. 5, No. 2, pp. 113-120, February 1994.
[11] Yu-Kwong Kwok, “Parallel program execution on a heterogeneous PC cluster using task duplication”, Proceedings of the 9th Heterogeneous Computing Workshop, 2000 (HCW 2000), pp. 364-374, May 2000.
[12] Zhenying Liu, Binxing Fang, Yi Zhang and Jianqi Tang, “Scheduling algorithms for a fork DAG in a NOWs”, Proceedings of the Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, pp. 959-960, May 2000.
[13] I. Foster, C. Kesselman, “The Grid: Blueprint for a New Computing Infrastructure”, China Machine Press, second edition, pp. 41-41, April 2005.
[14] Michael Pinedo, “Scheduling: Theory, Algorithms, and Systems”, Prentice Hall, 1995.
[15] O. H. Ibarra, C. E. Kim, “Heuristic algorithms for scheduling independent tasks on nonidentical processors”, Journal of the ACM, Vol. 24, No. 2, pp. 280-289, April 1997.
[16] Bruce Eckel, “Thinking in Java”, China Machine Press, pp. 825-901, January 2002.

Authors

Zhan Gao

Born on Feb. 18, 1982. He received a B.E. degree in Computer Science & Technology from Beijing Jiaotong University , China, 2004. He is currently a PH.D. Candidate in Computer Application Technology at Beijing Jiaotong University. His research interests include Grid and Distributed Computing.

Siwei Luo

Born on Dec. 23, 1943. He obtained his Ph.D. degree in computer science form Shinshu University, Japan, in 1984. He is currently a professor and doctoral supervisor of the School of Computer and Information Technology, Beijing Jiaotong University. His research interests include neurocomputing, neural networks, pattern recognition, and parallel computing.

Ding Ding

Born on Jan. 20, 1980. She received a B.E. degree in Computer Science & Technology from Beijing Jiaotong University, China, 2000, and M.E. degree in Computer Application Technology from Beijing Jiaotong University, China, 2003. In 2003 she joined the faculty of Beijing Jiaotong University, China where she is currently a lecturer in the School of Computer & Information Technology. She is also a member of China Computer Federation. Her current research interests include Parallel and Dstributed Coming grid computing.