# A Comparison among Grid Scheduling Algorithms for Independent Coarse-Grained Tasks

Noriyuki Fujimoto
Graduate School of Information Science and Technology,
Osaka University
1-3, Machikaneyama, Toyonaka, Osaka, 560-8531, Japan
fujimoto@ist.osaka-u.ac.jp

Kenichi Hagihara
Graduate School of Information Science and Technology,
Osaka University
1-3, Machikaneyama, Toyonaka, Osaka, 560-8531, Japan
hagihara@ist.osaka-u.ac.jp

## Abstract

*The most common objective function of task scheduling problems is makespan. However, on a computational grid, the 2nd optimal makespan may be much longer than the optimal makespan because the computing power of a grid varies over time. So, if the performance measure is makespan, there is no approximation algorithm in general for scheduling onto a grid. In contrast, recently the authors proposed the computing power consumed by a schedule as a criterion of the schedule and, for the criterion, gave* $(1 + \frac{m(\log_e(m-1)+1)}{n})$*-approximation algorithm RR for scheduling* $n$ *independent coarse-grained tasks with the same length onto a grid with* $m$ *processors. RR does not use any prediction information on the underlying resources. RR is the first approximation algorithm for grid scheduling. However, so far any performance comparison among related heuristic algorithms is not given. This paper shows experimental results on the comparison of the consumed computing power of a schedule among RR and five related algorithms. It turns out that RR is next to the best of algorithms that need the prediction information on processor speeds and task lengths though RR does not require such information.*

## 1 Introduction

**Public-resource computing** [2], such that the project SETI@home [2] has been carrying out, is the computing which is performed with donated computer cycles from computers in homes and offices in order to perform large scale computation faster. Public-resource computing is one form of grid computing. In public-resource computing, the original users also use their computers for their own purpose. So, their use may dramatically impact the performance of each grid resource. In the following, this paper refers to a set of computers distributed on the Internet and participated in public-resource computing as a **computational grid** (or simply a **grid**).

This paper addresses task scheduling of a single parameter-sweep application onto a computational grid. A **parameter-sweep application** is an application structured as a set of multiple "experiments", each of which is executed with a distinct set of parameters [3]. There are many important parameter-sweep application areas, including bioinformatics, operations research, data mining, business model simulation, massive searches, Monte Carlo simulations, network simulation, electronic CAD, ecological modeling, fractals calculations, and image manipulation [1, 10]. Such a application consists of a set of independent coarse-grained tasks such that each task corresponds to computation for a set of parameters. For example, each SETI@home task takes 3.9 trillion floating-point operations, or about 10 hours on a 500MHz Pentium II, yet involves only a 350KB download and 1KB upload [2]. Therefore, for the purpose of scheduling a single parameter-sweep application, a computational grid can be modeled as a heterogeneous parallel machine such that processor speed unpredictably varies over time and communication delays are negligible.

The most common objective function of task schedul-

ing problems (both for a grid and for a non-grid parallel machine) is makespan. However, on a grid, makespan of a non-optimal schedule may be much longer than the optimal makespan because the computing power of a grid varies over time. For example, consider an optimal schedule with makespan $OPT$. If a grid is suddenly slowed down at time $OPT$ and the slow speed situation continues for a long period, then the makespan of the second optimal schedule is far from $OPT$. So, if the criterion of a schedule is makespan, there is no approximation algorithm in general for scheduling onto a grid. In contrast to this, recently the authors proposed a novel criterion of a schedule called TPCC and gave $(1 + \frac{m(\log_e(m-1)+1)}{n})$-approximation algorithm RR for minimum TPCC scheduling of a coarse-grained parameter-sweep application with the same length tasks where $n$ is the number of tasks and $m$ is the number of processors [5]. TPCC represents the total computing power consumed by a parameter-sweep application. RR does not use any prediction information on the performance of underlying resources. Hence, this result implies that, regardless how the speed of each processor varies over time, the consumed computing power can be limited within $(1 + \frac{m(\log_e(m-1)+1)}{n})$ times the optimal one in such a case. This is not trivial because makespan cannot be limited even in the case.

However, so far any performance comparison among related algorithms is not given. This paper compares the proposed algorithm RR with several related algorithms by simulation. The remainder of this paper is organized as follows. First, Section 2 defines the grid scheduling model used in this paper. Next, Section 3 reviews the proposed algorithm RR. Then, Section 4 surveys related works. Last, Section 5 shows some experiments and the implication of the results.

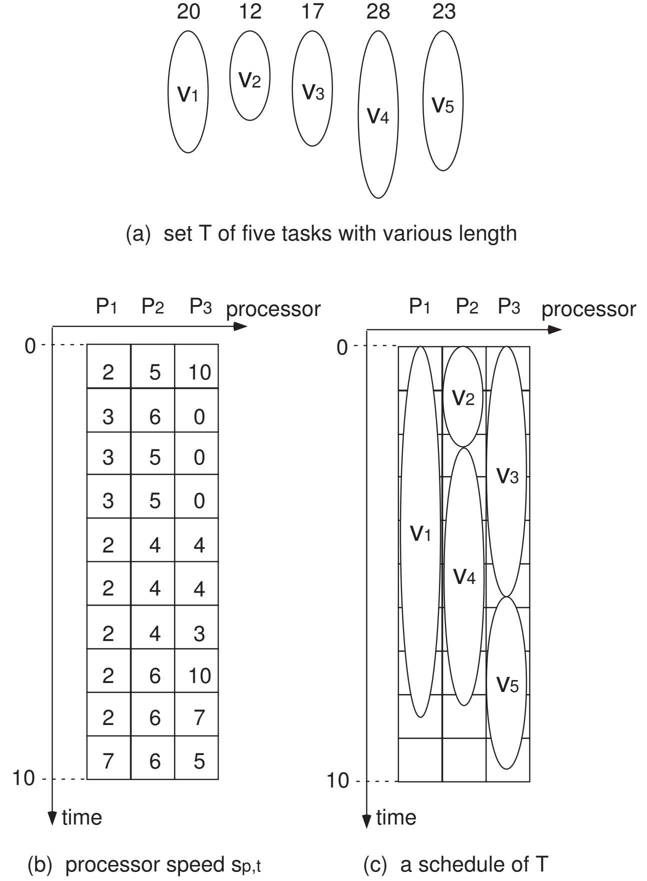


**Figure 1. The proposed grid scheduling model**

## 2 A Grid Scheduling Model

### 2.1 A Performance Model

The **length** of a task is the number of instructions in the task. The **speed** of a processor is the number of instructions computed per unit time. A grid is heterogeneous, so processors in a grid have various speed by nature. In addition, the speed of each processor varies over time due to the load by the original users in public-resource computing. That is, the speed of each processor is the excess computing power of the processor which is not used by the original users and is dedicated to a grid. Let $s_{p,t}$ be the speed of processor $p$ during time interval $[t, t+1)$ where $t$ is a non-negative integer. Without loss of generality, we assume that the speed of each processor does not vary during time interval $[t, t+1)$ for every $t$ by adopting enough short time as the unit time. We also assume that we cannot know the value of any $s_{p,t}$ in advance. $s_{p,t}$ may be zero if the load by the original users is very heavy or the processor is powered off. For simplicity, processor addition, processor deletion, and any failure are not considered in this paper. Fig. 1(a) shows an example of a set of tasks. Fig. 1(b) shows an example of processor speed distribution. Note that processor $P_3$ has speed zero during time interval $[1, 4)$. This means one of the following things:

- $P_3$ has no excess computing power due to very heavy load by the original users during time interval $[1, 4)$.
- $P_3$ is powered off during time interval $[1, 4)$.

### 2.2 A Schedule

Let $T$ be a set of $n$ independent tasks with the same length $L$. Let $m$ be a number of processors in a computational grid. We define a schedule of $T$ as follows. A **schedule** $S$ of $T$ onto a grid with $m$ processors is a finite set of triples $\langle v, p, t \rangle$ which satisfies the following rules R1

and R2, where $v \in T$, $p$ $(1 \leq p \leq m)$ is the index of a processor, and $t$ is the **starting time** of task $v$. A triple $\langle v, p, t \rangle \in S$ means that the processor $p$ computes the task $v$ between time $t$ and time $t+d$ where $d$ is defined so that the number of instructions computed by the processor $p$ during the time interval $[t, t+d)$ is exactly $L$. We call $t+d$ the **completion time** of the task $v$. Note that starting time and completion time of a task are not necessarily integral.

R1 For each $v \in T$, there is at least one triple $\langle v, p, t \rangle \in S$.

R2 There are no two triples $\langle v, p, t \rangle, \langle v', p, t' \rangle \in S$ with $t \leq t' < t+d$ where $t+d$ is the completion time of $v$.

Informally, the above rules can be stated as follows. The rule $R1$ enforces each task $v$ to be executed at least once. The rule $R2$ says that a processor can execute at most one task at any given time. A triple $\langle v, p, t \rangle \in S$ is called the **task instance** of $v$. Note that $R1$ permits a task to be assigned onto more than one processor. Such a task has more than one task instances. To assign a task onto more than one processor is called **task replication**.

## 2.3 Criteria of a Schedule

### 2.3.1 Makespan

The **makespan** of schedule $S$ is the maximum completion time of all the task instances in $S$. For example, Fig. 1(c) shows a schedule of $T$, i.e., $\{\langle v_1, P_1, 0\rangle, \langle v_2, P_2, 0\rangle, \langle v_3, P_3, 0\rangle, \langle v_4, P_2, 11/5\rangle, \langle v_5, P_3, 23/4\rangle\}$. The makespan of the schedule is $47/5$.

### 2.3.2 TPCC

Let $T$ be a set of $n$ independent tasks with the same length $L$. Let $S$ be a schedule of $T$ onto a grid with $m$ processors. Let $M$ be the makespan of $S$. Let $s_{p,t}$ be the speed of processor $p$ during the time interval $[t, t+1)$. Then, the **total processor cycle consumption** (TPCC, for short) of $S$ is defined as $\sum_{p=1}^{m} \sum_{t=0}^{\lfloor M \rfloor -1} s_{p,t} + \sum_{p=1}^{m} (M - \lfloor M \rfloor) s_{p,\lfloor M \rfloor}$. For example, TPCC of the schedule in Fig. 1(c) is $21 + 45 + 38 + 7 \times \frac{2}{5} + 6 \times \frac{2}{5} + 5 \times \frac{2}{5} = 111.2$.

The criterion means the total computing power dedicated to the parameter-sweep application. The longer makespan is, the larger TPCC is. Conversely, the larger TPCC is, the longer makespan is. That is, every schedule with good TPCC is a schedule also with good makespan. The goodness of the makespan seems to be reasonable for the dedicated computing power, i.e., the corresponding TPCC . In this sense, the criterion is meaningful.

## 2.4 A Grid Scheduling Problem

This paper addresses the following grid scheduling problem:

- Instance: A set $T$ of $n$ independent tasks with the same length $L$, a number $m$ of processors, unpredictable speed $s_{p,t}$ of processor $p$ during the time interval $[t, t+1)$ for each $p$ and $t$

- Solution: A schedule $S$ of $T$ onto a grid with $m$ processors

- Measure: either makespan or The TPCC $\sum_{p=1}^{m} \sum_{t=0}^{\lfloor M \rfloor -1} s_{p,t} + \sum_{p=1}^{m} (M - \lfloor M \rfloor) s_{p,\lfloor M \rfloor}$ of $S$ where $M$ is the makespan of $S$

A **makespan optimal schedule** is a schedule with the smallest makespan among all the schedules. An **TPCC optimal schedule** is a schedule with the smallest TPCC among all the schedules. Note that the set of makespan optimal schedules is the same as the set of TPCC optimal schedules.

# 3 The Proposed Algorithm RR

In this section, dynamic scheduling algorithm RR is illustrated. First of all, a data structure called a ring is defined. Then, using a ring of tasks, RR is described.

A **ring** of tasks is a data structure which manages a set of tasks. The tasks in a ring have a total order such that no task has the same order as any other task. A ring has a **head** which points to a task in the ring. The head in a ring is initialized to point to the task with the lowest order in the ring. The task pointed to by the head is called the **current task**. The **next task** in a ring is defined as follows. If the current task is the task with the highest order in the ring, then the next task in the ring is the task with the lowest order in the ring. Otherwise, the next task in a ring is the task with the minimum order of the tasks with higher order than the current task. A head can be **moved** so that the head points to the next task. Hence, using a head, the tasks in a ring can be scanned in the **round-robin fashion**. Arbitrary task in a ring can be **removed**. If the current task is removed, then a head is moved so that the next task is pointed to.

RR runs as follows. At the beginning of the dynamic scheduling by RR, every processor is assigned exactly one task respectively. If some task of the assigned tasks is completed, then RR receives the result of the task and assigns one of yet unassigned tasks to the processor. RR repeats this process until all the tasks are assigned. At this point in time, exactly $m$ tasks remain uncompleted. RR manages these $m$ tasks using a ring of tasks. Then, RR repeats the following process until all the remaining $m$ results are received: If the task instance of some task $v$ is completed on processor $p$, then RR receives the result of $v$ from $p$, kills all the task instances of $v$ running on processors except $p$, removes $v$ from the ring, selects task $u$ in the ring in the round-robin fashion, and replicates the task $u$ onto the processor $p$.

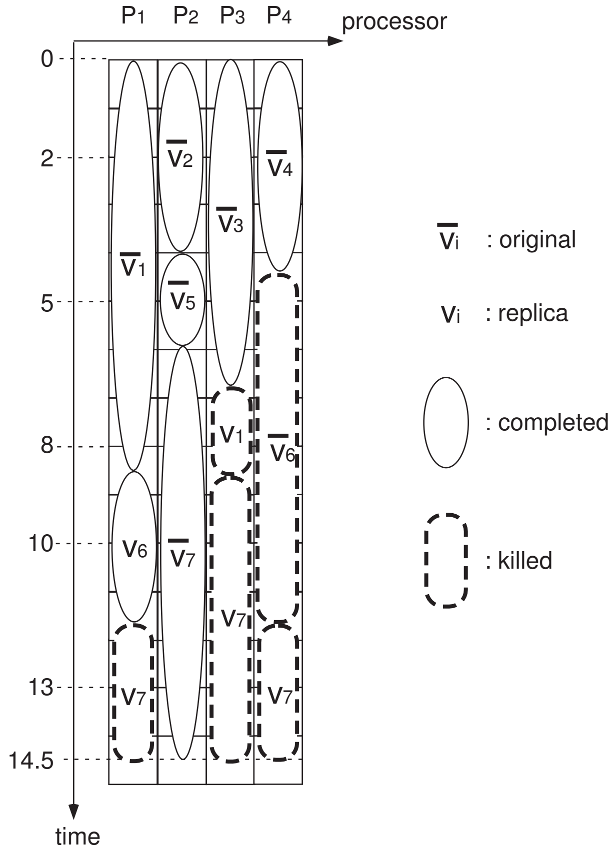


**Figure 2. An example of a schedule generated by RR**

The **original** of task $v$ is the task instance which is assigned earliest of all the task instances of $v$. The other task instances of $v$ are called the **replicas** of $v$. Notice that the original of every task $v$ is unique in a schedule generated by RR. Fig. 2 shows a schedule which RR generates from seven tasks $\{v_1, v_2, \cdots, v_7\}$ for four processors $\{P_1, P_2, P_3, P_4\}$ in the case that the queue $Q$ is initialized $\langle v_1, v_2, \cdots, v_7\rangle$ and the ring $R$ is initialized $\{v_1, v_3, v_6, v_7\}$ with the total order $v_1 < v_3 < v_6 < v_7$ where $x < y$ means that task $x$ has lower order than task $y$. The bar over $v_i$ $(i \in \{1, 2, \cdots, 7\})$ represents that the task instance is the original. A task instance without the bar is a replica. A dotted line represents that the task instance is killed because one of the other task instances completes earlier than the task instance. As for $v_6$, the replica completes earlier than the original. On the other hand, as for $v_1$ and $v_7$, the replica completes later than the original.

# 4 Related Works

**Static scheduling** is the scheduling such that all decisions are done before the execution of a schedule. In contrast, **dynamic scheduling** is the scheduling such that some or all decisions are done during the execution of a schedule. In a grid, the processor speed of each processor varies over time. So, dynamic scheduling is more appropriate than static scheduling.

This section summarizes dynamic grid scheduling algorithms which are recently developed and/or used to compare with other grid scheduling algorithms [3, 10]. These algorithms are DFPLT, Suffrage-C, Min-min, Max-min and WQ. In Section 5, all these five algorithms are compared with the proposed algorithm RR. These algorithms are based on the following framework: First, all the tasks are enqueued to a task queue; Whenever a processor becomes available, the task with the highest priority is dequeued and that processor are allocated to that task; When several processors simultaneously become available, ties are broken in the manner that depends on each algorithm; This process is repeated until all the tasks are completed. The difference among the five algorithms are in how to compute task priorities and how to break ties.

**DFPLTF** (Dynamic FPLTF) [10] is the result of a modification Silva et al. made on static FPLTF (Fastest Processor to Largest Task First) [9] so as to make it adaptive for a grid. FPLTF is a static scheduler that presents good makespan on a heterogeneous parallel machine. DFPLTF gives the highest priority to the largest task. When several processors simultaneously become available, ties are broken arbitrarily. DFPLTF needs prediction information on processor speeds and task lengths.

**Suffrage-C** [3] is a revised version Casanova et al. made on Suffrage [8] so as to make it easier to implement. There is no performance difference between Suffrage-C and Suffrage [3]. Suffrage-C gives each task its priority according to its suffrage value. For each task, its suffrage value is defined as the difference between its best completion time and its second best completion time. The suffrage value of each task varies over time because of the change of processor speed in a grid. The idea behind Suffrage-C is that a processor is assigned to a task that would suffer the most if that processor would not be assigned to that task. Suffrage-C needs prediction information on processor speeds and task lengths.

Min-min and Max-min are based on static algorithms presented in [7]. Min-min and Max-min were implemented in SmartNet [4] and presented in [8]. **Min-min** gives the highest priority to the task which can be completed earliest. The ties are broken arbitrarily. The idea behind Min-min is that assigning tasks to processors that will execute them fastest. **Max-min** gives the highest priority to the task with

the maximum earliest completion time. The ties are broken arbitrarily. The idea behind Max-min is that overlapping long-running tasks with short-running ones. For example, if there is only one long task, Max-min will execute many short tasks in parallel with the long task. In contrast, Min-min will execute short tasks in parallel and the long task will follow the short tasks. So, in such a case, Max-min is superior to Min-min. Min-min and Max-min need prediction information on processor speeds and task lengths.

**WQ** (Work Queue) is a classic algorithm that was originally developed for homogeneous parallel machines [6]. WQ arbitrarily gives priorities to tasks and arbitrarily breaks the ties. The idea behind WQ is that faster processors will be allocated more tasks than slower processors. WQ does not use any prediction information on processor speeds and task lengths.

Note that all the above algorithms are heuristic algorithms, i.e., algorithms without performance guarantee. In contrast to this, the proposed algorithm is an approximation algorithm, i.e., an algorithm with performance guarantee. The proposed algorithm is the first approximation algorithm for grid scheduling.

## 5 Experiment

We performed 4,370 simulations to compare TPCC of the proposed algorithm RR with TPCCs of the five related algorithms, i.e., WQ, DFPLTF, Suffrage-C, Max-min, and Min-min. We used maximum 1,024 tasks and 256 processors. To perform the simulations, we developed a grid scheduling simulator. Our simulator is faithfully based on the grid scheduling model described in Section 2. For a given number $n$ of tasks, maximum task length $L$, a number $m$ of processors, a maximum processor speed $s$, and a scheduling algorithm $A$, our simulator simulates scheduling $n$ independent tasks with task length $\in \mathrm{U}(1, L)$ onto a grid with $m$ processors with speed $\in \mathrm{U}(0, s)$ by $A$ where $\mathrm{U}(a, b)$ represents the uniform distribution from $a$ to $b$. Finally, our simulator outputs a Gantt chart, makespan, TPCC, and an upper bound of the approximation ratio of the TPCC. The upper bound is computed as follows. Let $W$ be the total number of instructions in given $n$ tasks. Since the grid must perform at least $W$ instructions until the execution of a schedule completes, the optimal TPCC is at least $W$. So, the obtained TPCC divided by $W$ is an upper bound of the approximation ratio.

FPLTF, Suffrage-C, Min-min, and Max-min need the prediction information on task lengths and processor speeds. RR and WQ does not require such information. In our experiments, whenever the prediction information is needed, our simulator gives the algorithm the precise information.

Fig. 3 through Fig. 11 show upper bounds of approximation ratios of TPCC of each algorithms in various cases. Each plot is the average value of 20 trials.

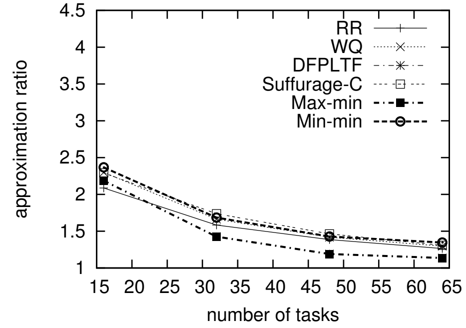


**Figure 3. Approximation Ratios Of TPCC (**task length $\in U(1, 30)$, processor speed $\in$ **U**$(0, 10)$, 16 processors**)**

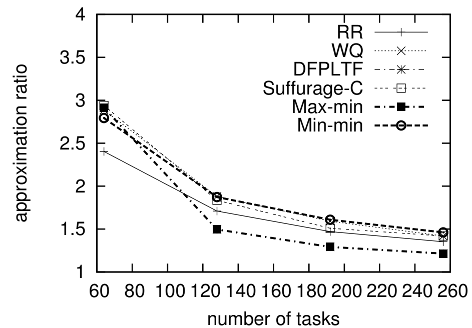


**Figure 4. Approximation Ratios Of TPCC (**task length $\in U(1, 30)$, processor speed $\in$ **U**$(0, 10)$, 64 processors**)**

The approximation ratio of every algorithm tends to decrease with an increase in the number of tasks. If the number of tasks is at least three times the number of processors, the approximation ratio is less than two.

Usually Min-min is the worst. On the other hand, in almost all cases, Max-min achieves the best approximation ratio and RR achieves the best or the second best approximation ratio. However, Max-min (also Min-min and Suffrage-C) needs prediction information on the processor speeds and task lengths. In contrast, RR does not require any prediction information on the underlying resources and tasks. Nevertheless, RR is next to the best of algorithms that use prediction information. So, RR is a good algorithm when performance information can not be well predicted.

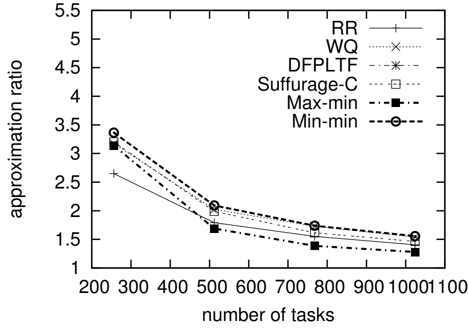


**Figure 5. Approximation Ratios Of TPCC (**task length $\in U(1,30)$, processor speed $\in \mathbf{U}(0,10)$, 256 processors**)**

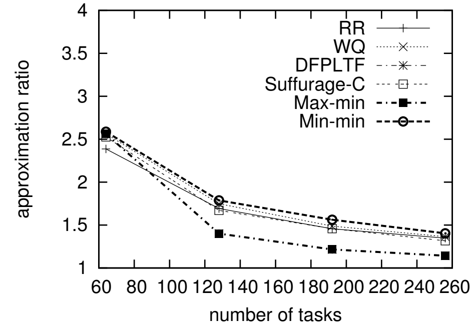


**Figure 7. Approximation Ratios Of TPCC (**task length $\in U(1,30)$, processor speed $\in \mathbf{U}(0,5)$, 64 processors**)**

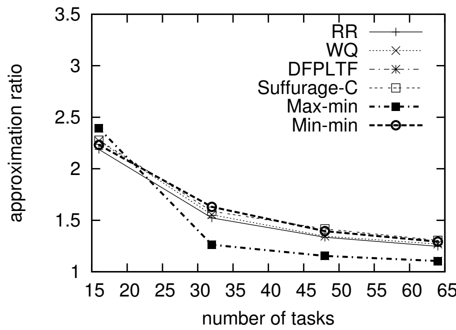


**Figure 6. Approximation Ratios Of TPCC (**task length $\in U(1,30)$, processor speed $\in \mathbf{U}(0,5)$, 16 processors**)**

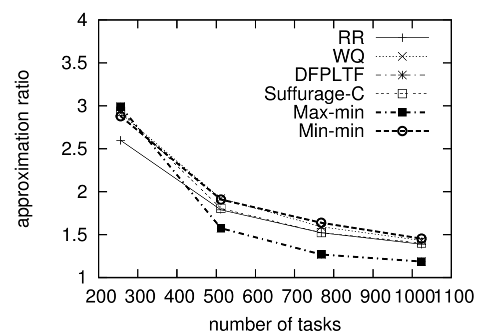


**Figure 8. Approximation Ratios Of TPCC (**task length $\in U(1,30)$, processor speed $\in \mathbf{U}(0,5)$, 256 processors**)**

# 6 Conclusion

This paper has compared by simulation the consumed computing power of grid scheduling algorithms for a parameter-sweep application. The compared algorithms include algorithm RR which was recently proposed by the authors. RR is an approximation algorithm if the task lengths are the same though any other existing algorithm is not an approximation algorithm even in the case. It has turned out that RR is next to the best of algorithms that need the prediction information on processor speeds and task lengths though RR does not require such information.

# Acknowledgement

This research was supported in part by Grant-in-Aid for Scientific Research on Priority Areas (15017260) from the Ministry of Education, Culture, Sports, Science, and Technology of Japan and also in part by Grant-in-Aid for Young Scientists (B)(14780213) from the Japan Society for the Promotion of Science.

# References

[1] D. Abramson, J. Giddy, and L. Kotler. High performance parametric modeling with Nimrod/G: Killer application for the global grid? In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 520–528, 2000.

[2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[3] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *9th Heterogeneous Computing Workshop (HCW)*, pages 349–363, 2000.

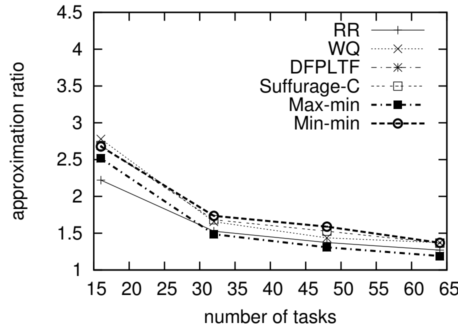


**Figure 9. Approximation Ratios Of TPCC (**task length $\in U(1,20)$, processor speed $\in$ $\mathbf{U}(0,10), 16$ processors**)**

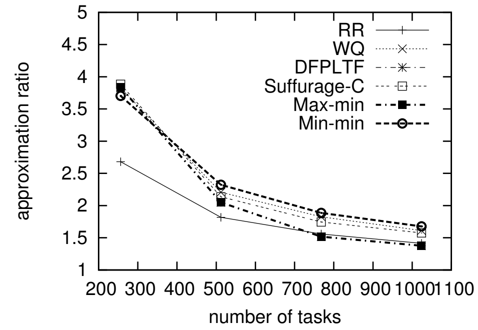


**Figure 11. Approximation Ratios Of TPCC (**task length $\in U(1,20)$, processor speed $\in$ $\mathbf{U}(0,10), 256$ processors**)**

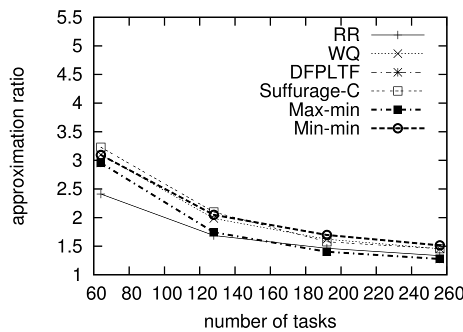


**Figure 10. Approximation Ratios Of TPCC (**task length $\in U(1,20)$, processor speed $\in$ $\mathbf{U}(0,10), 64$ processors**)**


[4] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel. Scheduling resources in multi-user, heterogeneous, computing environments with smartnet. In *the 7th IEEE Heterogeneous Computing Workshop (HCW'98)*, pages 184–199, 1998.

[5] N. Fujimoto and K. Hagihara. Near-optimal dynamic task scheduling of independent coarse-grained tasks onto a computational grid. In *$32^{nd}$ Annual International Conference on Parallel Processing (ICPP-03)*, pages 391–398, 2003.

[6] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[7] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM*, 24(2):280–289, 1977.

[8] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *the 8th IEEE Heterogeneous Computing Workshop (HCW'99)*, pages 30–44, 1999.

[9] D. A. Menascé, D. Saha, S. C. D. S. Porto, V. A. F. Almeida, and S. K. Tripathi. Static and dynamic processor scheduling disciplines in heterogeneous parallel architectures. *Journal of Parallel and Distributed Computing*, 28:1–18, 1995.

[10] D. Paranhos, W. Cirne, and F. Brasileiro. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *International Conference on Parallel and Distributed Computing (Euro-Par), Lecture Notes in Computer Science*, volume 2790, pages 169–180, 2003.