Computer Languages, Systems & Structures, Vol. 33, No. 1, 2007, pp. 1-10

Parallel Resource Co-Allocation for the Computational Grid

Hui-Xian Li [a,b], Chun-Tian Cheng [b *], K.W.Chau [c]

[a]*School of Electronic and Information Engineering, Dalian Univ. of Tech., Dalian, 116024*
[b]*Institute of Hydroinformatics, Dept. of Civil Engineering, Dalian Univ. of Tech.,Dalian,116024, P. R. China*
[c]*Dept. of Civil and Structural Engineering, Hong Kong Polytechnic Univ., Hung Hom, Kowloon, Hong Kong*

Abstract

Resource co-allocation is one of the crucial problems affecting the utility of the grid. Because the numbers of the application tasks and amounts of required resources are enormous and quick responses to the requirements of users are necessary in the real grid environment, real-time resource co-allocation may be large-scale. A parallel resource co-allocation algorithm based on the framework for mapping with resource co-allocation is proposed in this paper. Through the result of experiments, it is concluded that the parallel method reduces the execution time of the resource co-allocation algorithm significantly, and makes the overall response time to the end-users small.

Keywords : Computational grid; Resource co-allocation; Parallel computing; Execution time

1. Introduction

Grid computing [1] has strong potential for large-scale computation and mass data processing. In a grid, a large number of users and resources are connected. Through it, the proper resources are allocated to users. In other words, an important role of the grid is to match the abilities of resources to the requirements of users. Hence the problem of resource co-allocation in the grid needs to be addressed. Resource co-allocation is necessary since an application submitted to the grid environment often requires several types of resources. The principal goal of resource co-allocation is to utilize effectively the power of various geographically distributed resources, by simultaneously allocating the requested multiple resources to the submitted applications to meet specific performance requirements.

The problem has been discussed recently, such as in the Globus project [2] and the Legion project [3]. However, their focus is on the implementation of resource location and resource co-allocation issues while algorithms for resource co-allocation are not considered. In fact, the general problem of resource co-allocation has been studied extensively and a number of heuristics algorithms proposed for heterogeneous computing systems. These algorithms include dynamic algorithms [4,5,6,7] and static algorithms [8, 9, 10]. However, most of these algorithms

[*] Corresponding author. Tel: +86-411-8470-8768; Fax: +86-411-8467-4141.
E-mail addresses: ctcheng@dlut.edu.cn (C.-T. Cheng), hxli@student.dlut.edu.cn (H.-X. Li).

concentrate on allocating computing resources only, and do not address the co-allocation of multiple types of resources in grids. Multiple types of resources are considered and a parallel task-scheduling algorithm based on a list algorithm was proposed in [11]. A framework for heterogeneous computing systems was proposed in [12], in which multiple kinds of resources are taken into account and a Directed Acyclic Graph (DAG) model is used to represent applications. A static heuristic algorithm was developed for the framework. In [13] a benefit-function resource mapping heuristic for computational grid environments was proposed to map a set of independent meta-tasks to various resources.

The algorithms mentioned above focus on minimizing the overall execution time of all tasks and do not consider the execution time of the allocation algorithms. When the numbers of applications and resources are small, the scheduling time is negligible when compared with the task execution time. But in the real grid environment, the amounts of the application tasks and resources are enormous and quick responses to users are needed. Resource co-allocation may be large-scale and should be real-time. These characteristics cannot be satisfied by the existing approaches since they do not consider the running time of their algorithms, which, therefore, do not scale well. Motivated by these concerns, a parallel resource co-allocation algorithm based on the mapping resource framework is proposed in this paper. The Parallel Virtual Machine (PVM) [14,15], which is a popular system for parallel programming, is used to design the parallel algorithm and to develop a software simulator to evaluate its performance on a cluster system. Through comparison of the execution time of the parallel algorithm with that of a serial one, it is concluded that the parallel computing algorithm is a very efficient method for greatly reducing the execution time of the resource co-allocation algorithm, and that this parallel algorithm makes great strides towards satisfying the large-scale and real-time requirements of grid resource co-allocation.

The rest of the paper is organized as follows. Section 2 describes models for grid resources and applications. Section 3 reviews the resources mapping framework based on a graph-theory formulation. Section 4 presents our parallel resource co-allocation algorithm. Section 5 gives the analysis of the simulation results. Finally, Section 6 concludes the paper.

2 System Model

2.1 Resource Model

In the computational grid, there are different types of resources to be shared. In [12], resources are divided into two groups from the perspective of computation: computing resources, such as networks of workstations, personal computers, etc.; and, non-computing resources, such as distributed parallel storage server (DPSS), redundant arrays of inexpensive disks (RAID), data depositories, science instruments, I/O devices, etc. Moreover, it is assumed that only one task can use a resource at any given time. The assumption is not reasonable in the computational grid. In this case, some non-computing resources can be shared by different computational grid users [11], such as DPSS and RAID, while others cannot be shared and can only be used by a single user at any given time, such as measurement instrumentation and I/O devices. In the work presented here, non-computing resources are further divided into two groups: sharable and non-sharable. It is the non-sharable resources that cause conflicts in use.

Three sets are used to denote the resources: n computing resources are represented as $C = \{c_1, c_2, ..., c_n\}$; m sharable non-computing resources are represented as $SR = \{sr_1, sr_2, ..., sr_m\}$; and, h non-sharable non-computing resources are represented as $NR = \{nr_1, nr_2, ..., nr_h\}$. Each resource has two attributes: type and capability. The type attribute denotes the type of the resource and the capability attribute denotes the available capability provided by the resource. Communication costs between resources are given by three matrixes, *CC*, *CSR* and *CNR*. *CC* is an $n \times n$ matrix, where CC_{ij} ($CC_{ij} \geq 0$), is the communication cost for transferring a byte between computing resources c_i and c_j. *CSR* is an $n \times m$ matrix, where CSR_{ij} ($CSR_{ij} \geq 0$), is the communication cost for transferring a byte between the computing resource c_i and the non-computing resource sr_j. *CNR* is an $n \times h$ matrix, where CNR_{ij} ($CNR_{ij} \geq 0$), is the communication cost for transferring a byte between the computing resource c_i and the non-computing resource nr_j.

2.2 Application Model

Let A be a set of N applications, $A = \{A_1, A_2, \cdots, A_N\}$, which is collected in a certain time period. When the application is processed in the grid, it should be decomposed into tasks such that every task is executed on a single computing resource. So each task is executed on a computing resource, and it may need multiple non-computing resources, which is similar to [12]. Each application consists of a set of tasks with precedence constraints and resource sharing constraints [12] among the application's tasks. Precedence constraints capture the order-of-execution requirements of tasks and data dependencies among tasks. Resource sharing constraints arise from the fact that tasks, which may belong to the same or different applications, may require use of the same non-sharable non-computing resources. Tasks with resource sharing constraints are called "incompatible". They cannot be executed concurrently even if there is no precedence constraint among them. Two graphs are used to represent the applications. One of them is the DAG, $G = (V, E)$, in which nodes represent both tasks and their resource requirements, directed edges represent both precedence and communication requirements, and the weight of an edge denotes the amount of data communication needed. Tasks of an application are called *Start* nodes, if and only if they are executed first and have no predecessors. Let P_{t_i} denote the predecessor set of a task t_i. The other is the compatibility graph,

$CG=(V,E)$, in which nodes denote tasks, and an edge denotes a resource sharing constraint between the two tasks linked by it. In the compatibility graph, the concept of an independent set [16] is used to determine tasks that are not precluded from being executed simultaneously. An independent set is a set of vertices of *CG* such that no two vertices of the set are adjacent. An independent set is called a maximal independent set if there is no other independent set of *CG* that contains it. A maximal independent set of *CG* represents a maximal set of tasks that can be executed concurrently, if there are no precedence constraints between them.

3. Review of the Mapping Framework

In [12], a static co-allocation algorithm is developed, and the purpose of the algorithm is to minimize the overall scheduling length for a given set of applications. First, all the DAGs of submitted applications are united into a single one through a hypothetical zero-cost node, which connects all the *Start* nodes of these DAGs with zero-weight edges. Then the single DAG is partitioned into levels in terms of the order of execution determined by precedence. There are corresponding orders of precedence between these levels; although each level may contain independent tasks without precedence constraints. Tasks within a level cannot necessarily be executed simultaneously since resource sharing constraints may exist among them. Finally, for each level, the maximal independent sets are selected and tasks in each maximal independent set are allocated their required resources.

Resource allocation for each level is independent, and moreover, the same method of the resource allocation is taken in each level, so resource allocation processes of all levels can be executed concurrently. In the next section, a parallel algorithm is developed based on this framework.

4. Parallel Resource Co-allocation Algorithm

4.1 Parameters Definition

Let t_i denote a task of the submitted application. It is considered that the static applications, including the set of the applications and the execution time of each task on a computing resource, are known in advance. It is assumed that the resource allocation for each task is an atomic transaction [2], and a task cannot run until it gets all the required resources. Before the parallel algorithm is described, some parameters used are defined as follows.

- $CERT_{c_i}$: The earliest available time of a computing resource, c_i.
- $RERT_{nr_i}$: The earliest available time of a non-sharable non-computing resource nr_i.
- $EET_{t_i}^{c_j}$: The estimated execution time of a task t_i on a computing resource c_j. If the task t_i cannot be executed on the computing resource, the value of $EET_{t_i}^{c_j}$ is defined as infinity.

- $S_{t_i}^{c_j}$: The earliest start time of a task t_i on a computing resource c_j. $S_{t_i}^{c_j}$ is defined as $S_{t_i}^{c_j} = \{ CERT_{c_i}, DP_{t_i}^{c_j} \}$, where $DP_{t_i}^{c_j}$ is the time when a task t_i receives all the needed data from all tasks in its predecessor set, P_{t_i}, if it is mapped onto a computing resource c_j.
- $F_{t_i}^{c_j}$: The earliest completion time of a task t_i on a computing resource c_j, which is defined as

$$F_{t_i}^{c_j} = S_{t_i}^{c_j} + EET_{t_i}^{c_j}. \quad (1)$$

4.2 Parallel resource co-allocation algorithm

PVM is applied to develop the parallel resource co-allocation algorithm. The parallel algorithm consists of a master program and a slave program. The master program is responsible for constructing the whole DAG for all the submitted applications and distributing tasks to the slaves. The slave program is the real resource allocator and each slave program allocates the required resources for tasks in a level of the whole DAG.

In the master program, all the DAGs of the submitted applications are combined into a single DAG, SDAG, through a hypothetical zero-cost *Entry* node. The master program divides *SDAG* into *l* levels and creates *l* slave programs. Each level, except for level zero, is assigned to a corresponding slave program. Their relationships are shown schematically in Figure 1.

Insert fgure 1

The task node information is then sent to corresponding slave programs. The pseudo-code for the master program is shown in Figure 2.

Insert fgure 2

Slave programs allocate required resources to tasks in each level. There are four main steps in the slave program. First, the compatibility graph *CG* is constructed for tasks in level *j* of *SDAG*, which is used to find the maximal independent sets of tasks. Second, a maximal independent set *S* is selected from *CG*. Third, the required resources are allocated to tasks in *S*. Fourth, a new maximal independent set is determined from tasks still awaiting resource allocation. The third step and the forth step are then repeated, until all tasks in level *j* are allocated the required resources. The algorithm of the slave program is shown in Figure 3. Some set variables are defined here: *WorkList* is the set of tasks still awaiting resource allocation; and, *AllocList* is the set of tasks with resources already allocated.

Insert fgure 3

The functions in the slave program are now introduced. The function of *SelectInSet* is to select a maximal independent set from the task set *WorkList*. Note that the function of *SelectNextInSet* is also to select a maximal independent set from the task set *WorkList*, but it is a little different from the function of *SelectInSet*, because it needs to consider the allocated task set *AllocList*. We will introduce the function *SelectNextInSet* in the latter part of this subsection. Because the problem of the maximal independent set is NP-complete [17], a heuristic approach [12] is used. The key idea is to select a critical vertex *v* and add it to the maximal independent set *S* that is initially empty. The node with the highest out-degree in the compatibility graph is chosen as *v* so that many tasks can be ready for mapping after the highest out-degree task execution. *CG* is then traversed to enlarge *S*. In the course of traversing *CG*, a node is added to *S* if it is an outlier, that is, it has no neighbors in *CG*, or is not adjacent to any node in *S*. The function of *SelectInSet* is shown in Figure 4.

Insert figure 4

The function of *Allocate* is to allocate resources to tasks. In order to minimize the completion time of tasks, the maximum-finish-time-first strategy is taken to determine the scheduling order for tasks in *S*. We calculate the best completion time of each task, i.e. the earliest finish time of each task on some computing resource, which is earlier than that on any other computing resource, and then place all tasks in a list by the order of the non-increasing best completion times. Each task is allocated to the required resources by this order, and then the parameters of resources are updated. The function of *Allocate* is shown in Figure 5.

Insert figure 5

The function of *SelectNextInSet* is to select the next maximal independent set from tasks remained in the *WorkList*. The function is shown in Figure 6.

Insert figure 6

At the end of this section, some constraints of the parallelization need to be pointed out. In the parallel computing, the master program executes two main tasks: starting the parallel slave process, and sending and receiving data to and from slave nodes. The former requires a little computation, which hardly has influence on the parallel computing. If the amount of data transmitted is very large, the latter may become a bottleneck of the parallel computing, especially when there is data communication between slave nodes. However, in our algorithm, applications that need to be allocated resources are collected in a certain time period. Thus if the certain time period is chosen appropriately, the application tasks data is not very large. In addition, our parallel algorithm is asynchronous such that data transmission from slave nodes to master node can be overlapped and there is no communication between slave nodes. So as the number of the tasks increases, data transmission can affect the efficiency and scalability of our algorithm at a slight degree.

5. Simulation Results

In order to evaluate the performance of the parallel algorithm discussed in Section 4, a software simulator is developed on a cluster system. The PVM system consists of 8 Pentium Ⅳ 1.7G PCs connected by 10Mbits/sec Ethernet. We choose ten minutes as the certain time period to collect applications. The parameters of resources and applications are given to the simulator as inputs, such as the communication cost between resources, the start times and the completion times of tasks, the topology among tasks, the resources requirements of tasks, etc. In order to compare this parallel algorithm with the serial algorithm, the algorithm in the framework [12], which is serial and proceeds level-by-level, is also implemented. In the experiment, the number of tasks ranges from 100 to 500 with increments of 100. Owing to the limit on the number of the PCs, the numbers of task levels are selected to be 4 and 8, corresponding to 4 and 8 PCs, respectively. The execution times of the two algorithms are shown in Figures 7 and 8, in which PRCA denotes the parallel resource co-allocation algorithm and SRCA denotes the serial resource co-allocation algorithm. From the simulation results, it can be observed that the execution time of PRCA algorithm is reduced significantly, which represents a quicker grid response to requirements of users. In addition, the degree of reduction of the execution time is proportional to the number of task levels. In Figure 7, the scheduling time of the serial algorithm is about three times longer than that of the parallel algorithm while, in Figure 8, the scheduling time of serial algorithm is over six times longer than that of the parallel algorithm. The more task levels, the more execution time reduction.

Insert figure 7

Insert figure 8

6. Conclusions

Parallel computing is an efficient method for resource co-allocation in computational grids since it involves a huge amount of resources and applications and needs to quickly process user's requirements. A parallel resource co-allocation algorithm is proposed in this paper. The simulation results show that the execution time is reduced significantly in the parallel algorithm when compared with a serial one. This adaptation to the large-scale and real-time requirements of resource co-allocation is of great benefit to grid performance. Similarly to the serial algorithm, only static resources and applications are considered in this parallel algorithm. In future work, the parallel algorithm will be developed to adapt to the dynamic nature of the grid environment, and to make modifications to the allocation policy in terms of the dynamic information of changes of resources and applications. We will further investigate the influence of the bottleneck due to the existence of the master program, and take some measures to reduce the influence. In addition, it is necessary to consider other QOS constraints, such as the deadline of tasks, the cost of resources usage, so that the parallel algorithm is practical and applications can get a high QOS of grid.

Finally, we want to point out the few restrictions of the proposed parallel algorithm for grid resource co-allocation. If the co-allocation and scheduling times are a small overhead in a grid environment of large computation and I/O times, then the impact of resource co-allocation on the system performance is unimportant. It is unnecessary to take the parallel computing in the resource co-allocation algorithm. Therefore, we explicitly do not propose our parallel algorithm for general use, but only for the scenario where the co-allocation and scheduling times are a great overhead for grid performance. For reasons of effectiveness, we sternly advise to consider the impact of the resource allocation on the system performance, when choosing the grid resource co-allocation algorithm.

Acknowledgments

This research was supported by the National Natural Science Fund of China (No. 50479055), the Natural Science Foundation of Liaoning Province, China (No. 20032113), and the Central Research Grant of Hong Kong Polytechnic University (G-T592).

References

[1] Foster I, Kesselman C, Tuecke S. The anatomy of the grid: enabling scalable virtual organizations. International Journal of High Performance Computing Applications 2001; 15(3): 200−222.

[2] Czajkowski K, Foster I, Kesselman C. Resource co-Allocation in computational grids. In: Proc. of the 8-th IEEE Int'l Symp. on High Performance Distributed Computing, Redondo Beach, CA, USA, 1999. p. 219−228.

[3] Legion Web Page. Http://legion.virginia.edu

[4] Leangsuksun C, Potter J, Scott S. Dynamic task mapping algorithms for a distributed heterogeneous computing environment. In: Proc. of the 4th Heterogeneous Computing Workshop (HCW- 95), San Juan, Puerto Rico, 1995. p. 30−34.

[5] Maheswaran M, Siegel H J. A Dynamic matching and scheduling algorithm for heterogeneous computing systems. In: Proc. of the 7th IEEE Heterogeneous Computing Workshop (HCW'98), 1998. p. 57−69.

[6] Freund R, Carter B, Watson D, Keith E, Mirabile F. Generational scheduling for heterogeneous computing systems. In: Proc. of Int'l Conf. Parallel and Distributed Processing Techniques and Applications (PD PTA '96), 1996. p. 769−778.

[7] Iverson M, Ozguner F. Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment. In: Proc of 7th Heterogeneous Computing Workshop (HCW '98), 1998. p. 70−78.

[8] Sih G C, Lee E A. A Compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. IEEE Trans. On Parallel and Distributed Systems 1993; 4(2): 175−187.

[9] Shroff P, Watson D W, Flann N S, Freund R F. Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environment. In: Proc. of 5th Heterogeneous Computing Workshop (HCW '96), 1996. p. 98−117.

[10] Wang L, Siegel H J, Roychowdhury V, Maciejewski A. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. Journal of Parallel and Distributed Computing 1997; 47(1): 8−22.

[11] Wang L Z, Cai W T, Lee B S, See S, Jie W. Resource co-allocation for parallel tasks in computational grids. In: Proceedings of the International Workshop on Challenges of Large Applications in Distributed Environments, 2003. p. 88−95.

[12] Alhusaini A H, Prasanna V K, Raghavendra C S. A framework for mapping with resource co-allocation in heterogeneous computing systems. In: 9th Proceedings Heterogeneous Computing Workshop (HCW 2000), 2000; p. 273−286.

[13] Ding Q, Chen G –L. A benefit function mapping heuristic for a class of meta-tasks in grid environments. In: Proc. of First IEEE/ACM International Symposium on Cluster Computing and the Grid, 2001; p. 654 –659.

[14] Geist G A　Sunderam V S. The PVM system: supercomputer level concurrent computation on a heterogeneous network of workstations. In: Proceeding of Distributed Memory Computing Conference, 1991; p. 258–261.

[15] Geist G A, Sunderam V S. The evolution of the PVM concurrent computing system. Compcon Spring '93, Digest of Papers. , 1993; p. 549–557.

[16] Christofides N. Graph theory: an algorithmic approach. New York: Academic Press, 1975.

[17] Garey M R, Johnson D S. Computers and intractability: a guide to the theory of NP-Completeness. San Francisco: W H Freeman and Co., 1979.

Hui-Xian Li received her MS degree in Computer Software and Theory form Xi Dian University, Xi'an, China, in 2003. Currently she is a Ph.D. candidate in the Institute of Hydroinformatics, Dalian University of Technology, Dalian, China. Her main research interests focus on the fields of parallel computation and grid computation.

Chun-tian Cheng received his MS and Ph.D. degree in hydro from Dalian University of Technology, Dalian, China in 1989 and 1994. Currently he is a professor in the Institute of Hydroinformatics, Dalian University of Technology, Dalian, China. His research interests include knowledge management system, decision support system and intelligent algorithm, model technology for space information and grid computation. He has been a member of International Association of Hydrological Sciences since 1999. He also serves as paper reviewer of 13 SCI Journals such as Information Sciences, Decision Support Systems, Pattern Recognition Letters, Journal of the American Water Resources Association, Journal of Hydrology, Environmental Modelling & Software, and so on.

K.W.Chau is currently an Associate Professor in Department of Civil and Structural Engineering of The Hong Kong Polytechnic University. He is very active in undertaking research works and the scope of his research interest is very broad, covering artificial intelligence, knowledge-based system development, knowledge management, numerical flow modeling, water quality modeling, hydrological modeling and computer-aided design.

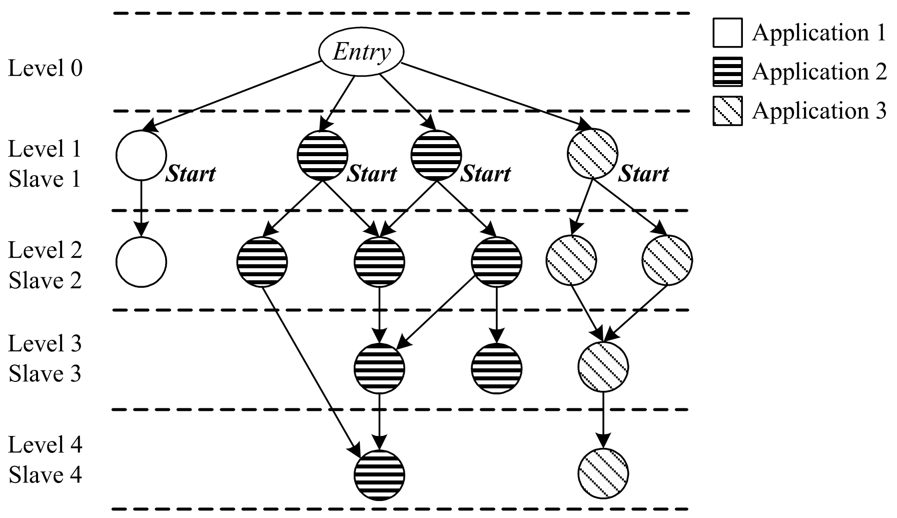

Figure 1. The corresponding relationship between slave programs and levels in the *SDAG*

Algorithm of the master program

Inputs: DAGs of all submitted applications, the resource requirement of each task, the cost of computation and communication

Outputs: the scheduling order of tasks (*SOrder*), the schedule length (*SLength*)

```
{
    Combine all submitted applications DAGs into SDAG;
    Do level partitioning of SDAG;
    Broadcast the slave program to each child node;
    For each child j do{
        Send information for tasks in level j to child node j;}
    For each child j do{
        Receive results from child node j;}
    Output SOrder and SLength of all tasks;
    Exit PVM;
}
```

Figure 2. The algorithm of the master program

Algorithm of the slave program at node *j*

```
{
    Receive level $j$ task information from master;
    Initialize $WorkList$ to contain all tasks in level $j$;
    Initialize $AllocList = \phi$;
    Let $SOrder = \phi$ and $SLength = 0$;
    Construct the compatibility graph $CG$ for all tasks in level $j$;
    $S = SelectInSet(WorkList, CG)$;        /* Find a maximal independent set of tasks $S$ from
                                                $WorkList$ */
    While ( $WorkList \neq \phi$ ) {
        $Allocate(S, SOrder, SLength)$;    /* Allocate all required resources to each task
                                                in $S$ */
        Add all tasks in $S$ to $AllocList$ and remove them from $WorkList$;
        $S = SelectNextInSet(AllocList, WorkList, CG)$;
                                    /* Select the next maximal independent set of tasks */
    } /* end while */
    Send $SOrder$ and $SLength$ to master;
    Exit PVM;
}
```

Figure 3. The algorithm of the slave program

Function of *SelectInSet*

Inputs: tasks (*WorkList*), compatibility graph (*CG*)

Outputs: the maximal independent set of tasks (*S*)

```
{
    Select the highest out-degree node v in CG;
    Add v to S;
    For each task node u in WorkList do{
        If (u is an outlier in CG or u is not adjacent to any task node in S){
                /* In CG, we only consider the compatibility of the node in the WorkList */
            Add u to S; }
    }
}
```

Figure 4. The function of *SelectInSet*

Function of *Allocate*

Inputs: the maximal independent set (*S*), the schedule length (*SLength*)

Onputs: the scheduling order of tasks (*SOrder*)

```
{
    For each task $u$ in $S$ do{
        Compute the best completion time $bt_u$ of $u$;}
    Sort tasks in $S$ by the non-increasing $bt_u$ and save the order in SOrder;
    For each task $u$ in $S$ do {                /* by the scheduling order in SOrder */
        For each $c_i \in C$ do { Compute $F_u^{c_i}$ of task $u$;}
        $F_u^{c_j}$ = min{ $F_u^{c_i}$ | $i$=1, 2, …, $n$};   /* This means that task $u$ will be completed earliest
                                               on computing resource $c_j$ */
        Allocate computing resource $c_i$ to $u$;
        Allocate all the non-computing resources set $R(u)$ to $u$;
        $CEAT_{c_j} = F_u^{c_j}$;            /* Update the available time of $c_j$ */
        For any $r_k \in R(u)$ and $r_k$ is non-sharable do   { $REAT_{c_j} = F_u^{c_j}$; }
        If ( $Comp(u, c_j) > SLength$ ) then $SLength = Comp(u, c_j)$;
    }
}
```

Figure 5. The function of *Allocate*

```
Function of SelectNextInSet
Inputs: allocated tasks (AllocList), tasks (WorkList), compatibility graph (CG)
Outputs: the maximal independent set of tasks (S)
{
    Select the allocated task v with the earliest completion time in AllocList;
    Remove v from AllocList;
    Let $Ca = WorkList$;    /* Ca is the set of candidate tasks that can be allocated next */
    Remove all tasks from Ca that are incompatible with any allocated task in AllocList;
    If ( $Ca \neq \phi$ ) then $S = SelectInSet(Ca, CG)$;
    Else    $S = SelectNextInSet(Alloclist, Worklist, CG)$;
}
```

Figure 6. The function of *SelectNextInSet*

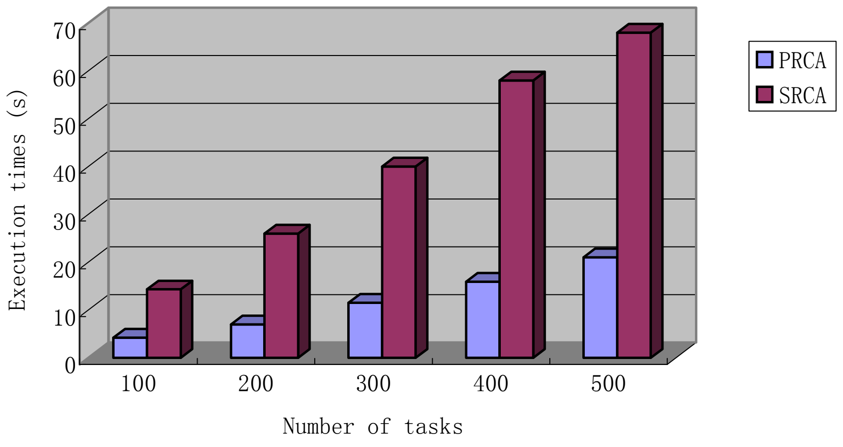

Figure 7. The comparison of execution times for a 4-level task graph

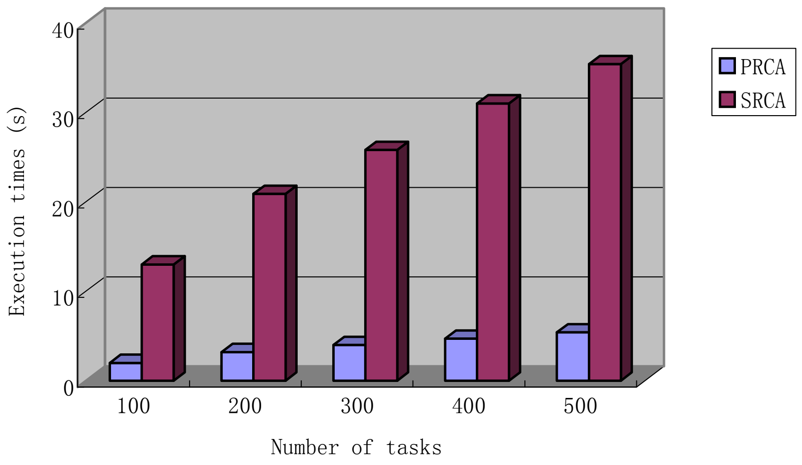

Figure 8. The comparison of execution times for an 8-level task graph