J Supercomput (2013) 64:177–203
DOI 10.1007/s11227-012-0845-z

Task graph pre-scheduling, using Nash equilibrium in game theory

Marjan Abdeyazdan · Saeed Parsa · Amir Masoud Rahmani

Published online: 12 February 2013

Abstract Prescheduling algorithms are targeted at restructuring of task graphs for optimal scheduling. Task graph scheduling is a NP-complete problem. This article offers a prescheduling algorithm for tasks to be executed on the networks of homogeneous processors. The proposed algorithm merges tasks to minimize their earliest start time while reducing the overall completion time. To this end, considering each task as a player attempting to reduce its earliest time as much as possible, we have applied the idea of Nash equilibrium in game theory to determine the most appropriate merging. Also, considering each level of a task graph as a player, seeking for distinct parallel processors to execute each of its independent tasks in parallel with the others, the idea of Nash equilibrium in game theory can be applied to determine the appropriate number of processors in a way that the overall idle time of the processors is minimized and the throughput is maximized. The communication delay will be explicitly considered in the comparisons. Our experiments with a number of known benchmarks task graphs and also two well-known problems of linear algebra, LU decomposition and Gauss–Jordan elimination, demonstrate the distinguished scheduling results provided by applying our algorithm. In our study, we consider ten scheduling algorithms: min–min, chaining, A^*, genetic algorithms, simulated annealing, tabu search, HLFET, ISH, DSH with task duplication, and our proposed algorithm (PSGT).

M. Abdeyazdan (✉) · A.M. Rahmani
Department of Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran
e-mail: m.abdeyazdan@srbiau.ac.ir

A.M. Rahmani
e-mail: rahmani@srbiau.ac.ir

S. Parsa
Department of Computer Engineering, Iran University of Science and Technology, Tehran, Iran
e-mail: parsa@iust.ac.ir

Keywords Prescheduling · Scheduling · Task graph · Game theory · Nash equilibrium

1 Introduction

This article proposes a static task graph prescheduling algorithm. Applying control and data dependency analysis, a program can be modeled as a directed acyclic graph (DAG) in which nodes at each level represent a task, which can be executed in parallel with the other nodes at the same level. When the program characteristics including task execution times, task dependencies and amounts of communicated data are known a priori, the static scheduling problem is to assign tasks to a multiprocessor system and arrange the tasks' executions in such a way that a minimum overall completion time [4, 28] is obtained.

The performance of static scheduling approaches suffers due to the trade-off between maximizing parallelism and minimizing communication, and worsens for fine grain tasks with a high communication to computation cost ratio. To alleviate the burden of scheduling, prescheduling algorithms have been applied to prepare task graphs for relatively more efficient and appropriate scheduling [24].

Clustering-based prescheduling approaches [5] tend to allocate heavily dependent tasks onto the same processing elements to reduce the overall communication cost and include two stages. In the first stage, tasks are grouped into a number of clusters and then these clusters are mapped onto the available processing elements by load-balancing heuristics [6]. Another attractive technique is the duplication-based approach, which aims to duplicate the parents of the candidate task to more than one processing element to reduce the candidate task's start or finish time by decreasing the communication overhead. Generally, prescheduling algorithms with duplication tend to perform better than nonduplication algorithms; however, the performance improvement, which originates from the exhausted backward search in the duplication step, usually leads to a higher degree of complexity and causes redundant duplications without contributing to performance.

Correct scheduling of task graph has much effect on optimal distribution of the tasks on parallel processors. To obtain an appropriate schedule, it is suggested to prepare task graphs for appropriate scheduling before the task graph can be scheduled. To achieve this, task merging [7, 8, 16], clustering [20], and duplication [30, 40] techniques are suggested.

In this article, a new task prescheduling algorithm using both task merging and duplication techniques to minimize the overall completion time of task graphs is suggested. In this algorithm, tasks at each level of the task graph attempt to reduce their earliest start time by merging themselves with a subset of their parents. When merging a task with its parents, the earliest start time of its brothers at the same level of the task graph may be reduced. Considering each level of the task graph as a distinct player seeking for appropriate number of processors to execute its independent tasks addressed by its nodes, the Nash equilibrium idea of game theory could be applied to assign appropriate number of processors in such a way that the overall completion time of the task graph and the earliest start time of tasks at each level of the task graph is minimized.

Equilibrium is a key concept in game theory. As optimization problems seek to optimal solutions, a game looks for equilibrium. Given a game with strategy sets for players, a pure Nash equilibrium is a strategy profile in which each player deterministically plays her chosen strategy and no one has an incentive to unilaterally change her strategy. Nash proved that every game with a finite number of players, each having a finite set of strategies, always possesses a mixed Nash equilibrium [41]. The concept of Nash equilibrium has become an important mathematical tool for analyzing the behavior of selfish users in noncooperative systems. The celebrated result of Nash [32, 33] guarantees the existence of Nash equilibrium in mixed strategies for every (finite) strategic game, and many algorithms have been devised to compute one [31].

Some research has been presented in the literature for energy aware scheduling of tasks with Since finding an optimal schedule is an NP-complete problem in general, researchers have resorted to devising a plethora of heuristics using a wide spectrum of techniques, including branch-and-bound, integer programming, searching, graph-theory, randomization, genetic algorithms, and evolutionary methods [37]. Some research has been presented in the literature for energy aware scheduling of tasks with precedence constraints [36, 38], and without precedence constraints [34] for parallel machines [22, 27].

The scheduling/mapping problem becomes a MOO problem in which the execution time is traded off with overall energy consumption [23, 26]. The solution is based on game theory that can solve this problem with a fast turnaround time. It also strikes a balance between the two goals. The goal is to provide algorithms for a rich class of energy and time tradeoffs that can be effectively utilized by the resource manager. For such a multiobjectives optimization problem, there is no unique solution [35].

Figure 1 illustrates the concept of the MOO problem with conflicting objective functions. For a given application and multiprocessing machine, M, the curve AB is the paretooptimal frontier, along which no further improvement can be done on energy consumption, P, or schedule length, SL, without sacrificing the other one. Note that points A and B are the operating points for the schedule length minimization problem and energy minimization problem, respectively. By formulating a MOO problem, we can choose an appropriate operating point along curve AB based on the current situation.

The appropriate number of processors for scheduling a given task graph could be computed before scheduling the task graph. Considering each level of the task graph

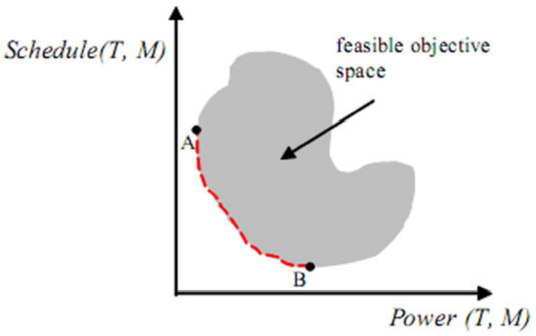

Fig. 1 Illustration of MOO problem [19]

as a player looking for appropriate number of processors to execute its independent tasks addressed by its nodes, the Nash equilibrium idea of game theory could be applied to compute the number of processors in order to minimize the idle time of the processors and overall throughput of processors is maximized where the goal is to balance the completion time of a parallel application with the overall energy (CPU) consumption.

The remaining parts of this article are organized as follows: In Sect. 2, related work is presented and in Sect. 3, a new algorithm is presented. This algorithm uses relations, which are fully described in subsections. In Sect. 3.1, is presented selecting the number of appropriate processors, in Sect. 3.2, is presented merging tasks, in Sect. 3.3, is presented applying game theory, and in Sect. 3.4, is presented the PSGT algorithm. In Sect. 4, is presented the results of applying our algorithm to some benchmark task graphs is compared with the results of applying some known scheduling algorithms, and finally in Sect. 5 the conclusion is presented.

2 Motivation and related work

Prescheduling techniques are mainly applied to reshape task graphs in a form suitable for scheduling. To reshape a task graph, task merging [1], clustering [2], and duplication [3] techniques are most commonly applied. ChouLai and Yang [20] have proposed a new duplication-based task scheduling algorithm for distributed heterogeneous computing (DHC) systems. For such systems, many researchers have focused on solving the NP-complete problem of scheduling directed acyclic task graphs to minimize the makespan. In this respect, a heuristic strategy called the Dominant Predecessor Duplication (DPD) scheduling algorithm, allowing for system heterogeneities and communication band width to exploit the potential of parallel processing, is presented. This algorithm can improve system utilization and avoid redundant resource consumption, resulting in better schedules. Many heuristics based on the directed acyclic graph (DAG) have been proposed for the static scheduling problem. These heuristics most often assign DAGs to processors to minimize overall run-time of the application. But applications on embedded systems, such as high performance DSP in image processing, multimedia, and wireless security, need schedules which use low energy, also. Most of these algorithms apply a simple model of the target system that assumes fully connected processors, a dedicated communication subsystem and no contention for the communication resources. Only a few algorithms consider the network topology and the contention for the communication resources [30].

There are several nondeterministic approaches [25, 39] to solve the multiprocessor task scheduling problem that is an NP-hard problem. The genetic algorithm presented in [25] has provided relatively better scheduling in comparison with the other nondeterministic approaches. The algorithm is bipartite in a way that each part is based on different genetic schemes, such as genome presentation and genetic operators. In the first part, it uses a genetic method to find an adequate sequence of tasks and in the second part it finds the best matching processors. However, these nondeterministic approaches provide different scheduling for a given task graph in different runs of their underlying algorithms. In order to cope with large scale task graphs, a cluster-based search (CBS), for scheduling large task graphs in parallel on a heterogeneous cluster of workstations connected by a high-speed network, is presented.

Consider the problem of routing n users on m parallel links under the restriction that each user may only be routed on a link from a certain set of allowed links for the user. This problem is equivalent to the correspondingly restricted scheduling problem of assigning n jobs to m parallel machines. In Nash equilibrium, no user may improve its own Individual Cost (latency) by unilaterally switching to another link from its set of allowed links [21]. Programs using parallel tasks can be modeled as task graphs so that scheduling algorithms can be used to find an efficient execution order of the parallel tasks.

Static scheduling has been well accepted for its predictability and online simplicity. Traditional static schedule generation techniques are usually based on the assumption of constant rate of resource supply known at design time. A preschedule is a static schedule without assuming constant and completely predictable rate of resource supply. The concepts of supply function and supply contract are introduced to define the actual online resource supply rate and the constraints to this rate known off-line [20]. Based on these concepts, the prescheduling problem is defined and sound preschedulers are presented [1–3, 20].

3 The proposed algorithm

There are three major steps in our proposed algorithm, PSGT: in the first step, determining the appropriate number of processors for initial graph. The second step is merging the initial graph and making the last graph and last step is scheduling of tasks. We are sure that the number of processors in first step appropriate for second step. Because in the merging step, the number of tasks (nodes) in the same level does not increase (the criteria for choosing is according to the number of tasks in the same level) and every task will be merged with its parents. The relationship between parent and child are into two sequential levels. Therefore, the number of parallel processors that are chosen in first step are appropriate for the merged graph in the second step.

In the first step, we have applied the idea of Nash equilibrium in game theory to compute the appropriate number of processors for scheduling a given task graph. In this step, each level of the task graph is considered as a player looking for appropriate number of parallel processors to execute its independent tasks in parallel. In the second step, considering the appropriate number of processors estimated in the first step, a task merging algorithm is applied to merge tasks with their parents in a way that the earliest start time of the task and the overall completion time of the task graph are reduced. In this step, each task is considered as a player seeking for reducing its earliest start time as much as possible. In last step, tasks assign to processors. In summary, the main steps of PSGT are as follows:

PSGT process:

Step 1: Compute the appropriate number of processors

1.1 Determine the average load of the processors as a parabola type function of the number of tasks

1.2 Apply Nash equilibrium to determine the appropriate number of processors

Step 2: Task merging

2.1 Computing the earliest start time of each task in the task graph
2.2 Sort the parents of each node in descending order of the earliest start time plus communication cost of their parents
2.3 Compute the benefit of merging each node with one or more of its parents
2.4 Apply Nash equilibrium to determine the appropriate merging of each node with its parents

Step 3: Apply scheduling algorithm (tasks assign to processors)

The PSGT method establishes a trade-off between time and energy. In order to minimize the energy consumption the number of processors should be minimized while to minimize the makespan the number of available processors may be increased. To determine the optimal number of processors, we consider each level of a task graph as a selfish player attempting to get suitable number of processors to execute its tasks in parallel. The overall benefit of applying the processors is maximized when the Nash equilibrium is reached among the levels. Afterward, considering the determined number of processors, the tasks are merged where applicable, to lessen their earliest start time (EST).

3.1 Compute the appropriate number of processors

Before applying our algorithm to a given task graph, the appropriate number of processors for scheduling the task graph should be computed. We have applied the idea of Nash equilibrium in game theory to compute the appropriate number of processors. Then the task graph is topologically sorted, each level of the task graph is considered as a player. The selfishness appears in the sense that the player, P_k, at level k with N_k nodes seeks for N_k parallel processors without considering the demands of the other levels. It is shown in Sect. 3.1.1 that the average load or in other words the utility of the processors could be defined in terms of a parabola type relation. In Sect. 3.1.2, the use of Nash equilibrium to compute the near optimal utilization of the processors is described.

3.1.1 Determine the average load of the processors

Let T be a task graph with n levels 1 to n where the level k includes N_k nodes. If the number of available processors, N_{av}, is less than or equal to the minimum value of N_1 to N_k, the idle time of the processors will be decreased. On the other hand, if the number of available processors, N_{av}, is equal to the maximum value of N_1 to N_k, the average load, L, of the processors will be minimized, but the degree of parallelism will be maximized. In general, as shown in Fig. 2, the average load, L, of the processors assigned to a task graph T can be computed by the following parabola type relation:

$$L(N) = -\alpha N^2 + L_{\max} \tag{1}$$

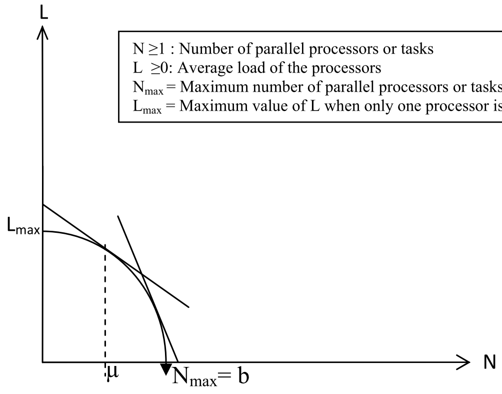

Fig. 2 The curve of the average load, L, of processors in terms of number, N, of parallel tasks (processors) [29]

where $N \geq 1$ indicates the number of parallel tasks, $L_{\max}$ is the maximum value of the processors load, and α is the slop of L. The maximum number of parallel tasks is equal to the number of available parallel processors. Therefore, the maximum load of processors, $L_{\max}$, is achieved when there is only a single processor available. The minimum load is when the number of tasks is equal to $\max(N_k)$. The reason for using a parabola type relation is further described below.

Let T be a task graph with n levels 1 to n where the level k includes N_k nodes each representing a separate task tasks and

$$\mu = (N_1 + N_2 + \cdots + N_k + \cdots N_n)/n \tag{2}$$

is the average number of nodes in T.

If there is only one processor, its economic value will be maximum. On the other hand, the economic value will be minimized when the number of available processors is equal to $N_{\max}$. Since increasing the number from μ up to the maximum required number of processors, $N_{\max}$, reduces the economic value of the available processors, therefore, there will be a negative slop for the value and thereby the load of the processors. It means that the first offshoot of the relation, L, for computing the load of the processors in terms of the number, N, of the available processors is as follows:

$$\frac{d(L(N))}{d(N)} = L'(N) < 0. \tag{3}$$

The second offshoot of $L(N)$ is also negative because when increasing the number of available processors from μ up to $N_{\max}$ the economic value of the processors reduces more sharply than when the number of processors are increased from 1 up to μ. By considering $L(N)$ as a parabola kind, we experienced the appropriate numbers of processors for task graphs scheduling.

3.1.2 Apply Nash equilibrium to determine the appropriate number of processors

In this section, Nash equilibrium is applied to the processors load model in relation (1), to determine the appropriate number of processors for scheduling a task graph T. Let T be a task graph with n levels from 1 to n where the level k includes

N_k nodes. The average number, μ, and the maximum number, N_{Max}, of the required processors are computed as follows:

$$N_{\mathrm{Max}} = \mathrm{Max}(N_1, N_2, \ldots, N_k, \ldots, N_n)$$

where n is the number of the task graph levels. The strategy, S_k, of each player, k, is to have access to 1 to N_{Max} processors. In summary the appropriate number of processors, N, for scheduling a task graph, T, can be computed as follows:

$S_k = [1, N_{\mathrm{Max}}]$ where S_k in {set of strategies, S} and $k \in [1..n]$

1. $\frac{d(L(N))}{d(N)} = L'(N) < 0 \Rightarrow \forall k \in 1..n$: $(N_k > \mu \Rightarrow U_k > 0)\ \wedge\ (N_k < \mu \Rightarrow U_k < 0)\ \wedge\ (N_k = \mu \Rightarrow U_k = 0)$
2. $U(N_k, N_{-k}) = (N_k - \mu) \times L(\mu) = N_k \times L(\mu) - \mu \times L(\mu)$
3. $\mathrm{Max}(U(N_k, N_{-k})) = \mathrm{Max}(N_k \times L(\mu) - \mu \times L(\mu)) = \mathrm{Max}(N_k \times L(\mu) - \mu \times L(\mu))$
4. $\frac{d(U(N_k, N_{-k}))}{d(N_k)} = U'(N_k) = 0$
6. $\Rightarrow\ U'(N_k) = L(\mu) + N_k \times L'(\mu) - (1/n) \times L(\mu) - \mu \times L'(\mu) = 0$
7. $\Rightarrow\ U'(N_k) = (1 - 1/n)L(\mu) + (N_k - \mu)L'(\mu) = 0$
8. $\Rightarrow\ U'(N_k) \approx L(\mu) + (N_k - \mu)L'(\mu) = 0$
9. $\Rightarrow\ (N_k^* - \mu)L'(\mu) = -L(\mu)$
10. $\Rightarrow\ (N_k^* - \mu) = -(L(\mu)/L'(\mu))$
11. $\Rightarrow\ N_k^* = -(L(\mu)/L'(\mu)) + \mu$
12. $\Rightarrow\ N_k^* = (-(-\alpha\mu^2 + L_{\max})/(-2\alpha\mu)) + \mu$
13. $\Rightarrow\ N_k^* = ((-\alpha\mu^2)/(2\alpha\mu) + (L_{\max})/(2\alpha\mu)) + \mu$
14. If $L(N) = 0 \Rightarrow N = N_{\max} \Rightarrow 0 = -\alpha N_{\max}^2 + L_{\max} \Rightarrow L_{\max} = \alpha N_{\max}^2$
15. $\Rightarrow\ N_k^* = ((-\alpha\mu^2)/(2\alpha\mu) + (\alpha N_{\max}^2)/(2\alpha\mu)) + \mu$
16. $\Rightarrow\ N_k^* = (-\mu/2) + (N_{\max}^2/(2\mu)) + \mu$
17. $\Rightarrow\ N_k^* = (N_{\max}^2 + \mu^2)/(2\mu)$
18. $\Rightarrow\ N_k^* = (\mu^2 + N_{\max}^2)/(2\mu)$
19. $N^* = N_k^*$ where N^* is the appropriate number of processors for all levels

$$\rho = \left(N^*/N_{\max}\right)(\mu/N_{\max}) \times 100. \tag{4}$$

In relation (4), the parameter ρ represents the percentage of the appropriate number of processors, N^*, compared with the maximum number of the required parallel processors, $N_{\max}$. If N^* equals $N_{\max}$ the value of the parameter ρ will be equal to 100. To determine the value of $N_{\max}$, the task graph is topologically sorted.

The algorithm for determining the appropriate number of processors for a task graph is as follows:

1. Strategies for employing processors that every player can choose are from 1 to $N_{\max}$ where $N_{\max}$ is the maximum number of available processors. Every level, can choose at most $N_{\max}$ processors, which is equal to the maximum number of parallel tasks in the task graph.
2. The appropriate number of processors is achieved at the Nash equilibrium amongst the players' choice. We calculate offshoot (derive) the function $L(N)$,

the answer of offshoot should be less than zero in order to have the best answer. Therefore, regarding Fig. 2, we should calculate the utility. Considering the curve in Fig. 2, representing the load of processors versus the number of available processors, if the number of tasks, N_k, is less than the average number of parallel tasks, $N_k < \mu$, in the program task graph, then the utility, U_k, of the kth player at the level k is negative; otherwise, if $N_k > \mu$, then the utility, U_k, of the kth player at the level k is positive. If $N_k > \mu$, then there are N_k processors, then $(N_k - \mu)$ processors will be idle, and there will be no shortage of processors; also, μ processors will be busy most of the times, therefore, their utility will be positive. On the other hand, if $N_k < \mu$, then there are less than μ processors and all the processors will be busy most of the time, and there will be shortage of processors. Also, $(N_k - \mu)$ processors will be required and their utility will be negative.

3. For every level of N_k, the utility is described as $U(N_k, N_{-k})$. We are looking for the highest utility for the N_k tasks located at the kth level of the task graph. The utility is maximized considering the utilities of the other players. If $N_k \leq \mu$, then the utility, U_k, of the kth player at the level k is negative because N_k processors will be busy at most of the times and $(\mu - N_k)$ processors will be required at some of the times. Otherwise, U_k will be positive because $(N_k - \mu)$ processors will be idle at most of the times. Therefore, the utility, U_k, of each player, k, can be computed as $L(N_k)^*(N_k - \mu)$. To reach the Nash equilibrium, the first offshoot of U is equalized with zero. Solving the resultant equation, the appropriate number of processors, N^*, is computed.
4. In order to obtain the best answer, we calculate the maximum amount of utility at each level.
5. To determine the most appropriate number of processors, N^*, the first offshoot (maximum utility) is set to zero.
6. In this step, calculate the first offshoot based on N_k.
7. Algebraic calculations.
8. The computed number, n, of processors has to be integer. Therefore, it is assumed that $1 - \frac{1}{n} \simeq 1$
9. Put the unknown in the left side of the equation.
10. Same as 9.
11. Same as 9.
12. Replace μ with N in relation (1) and calculate its offshoot.
13. Algebraic calculations.
14. Regarding Fig. 2, B locates at the intersection of $L = 0$ and $N = N_{\max}$. If $L(N) = 0$, therefore, $N = N_{\max}$ and by replacing the variables in relation (1) we can get to the this result $L_{\max} = \alpha N_{\max}^2$.
15. Replace $L_{\max}$ with $\alpha N_{\max}^2$ in line 13.
16. Algebraic calculations.
17. Algebraic calculations.
18. Algebraic calculations.
19. N_k^* is the appropriate number of processors at level k and N^* is the appropriate number of processors for all level.

3.2 Task merging

The critical path of a task graph may be reduced by merging its task (nodes). When merging a task with its related tasks, the communication cost is reduced, but on the other hand, the merged tasks cannot be executed in parallel. Merging a task with its related tasks begins with merging the task (node) with its parents in the task graph [16]. The degree of parallelism is computed as the sum of the execution time of the tasks divided by the length of the critical path.

$$\text{Parallelism} = \frac{\sum_{i=0}^{n} t(i)}{v} \tag{5}$$

where $t(i)$ is the execution time of the task i and v is critical path length. The critical path length gets shorter until the amount of code parallelism will increase.

3.2.1 Computing the earliest start time

A task within a task graph starts immediately after all of its parent nodes execution is completed. Therefore, to compute the earliest start time, EST_v, of a task, v, the earliest start time and the execution time of its parent nodes and the time it takes to receive the results from its parent nodes are required. In order to compute EST_v, the following relation can be used:

$$EST_v = \begin{cases} 0 & \text{Parent}(v) = \phi, \\ \text{MAX}_{p_i \in \text{Parents}(v)}(EST(p_i) + \tau(p_i) + L + \frac{c(p_i, v)}{B}) & \text{Parent}(v) \neq \phi. \end{cases} \tag{6}$$

In relation (6), $c(p_i, v)$ is the size of the data to be received by the task v from the parent p_i, L is the latency, B is the bandwidth of the communication network, and Parent(v) indicates the set of the parent nodes of the task v. If a node v has no parent nodes, its earliest start time will be zero.

3.2.2 Earliest time to collect data from parent nodes

In Fig. 3, a node v is merged with the subset of its parents, which reduces its earliest start time, the most. To achieve this, the time, $R_{pi,v}$, at which the outputs of each parent node, p_i, can be collected by the child, v, is computed, first. In order to compute the value of $R_{pi,v}$, the following relation can be applied:

$$R_{p_i,v} = EST(p_i) + \tau(p_i) + L + \frac{c(p_i, v)}{B}. \tag{7}$$

In relation (7), $c(p_i, v)$ indicates the size of the data that must be transmitted from p_i to v, L represents the latency and B indicates the bandwidth of the communication network. After, the value of $R_{pi,v}$ is computed, and the parents of v are sorted on the value of their $R_{pi,v}$, in a descending order. Starting with the node with the highest value of $R_{pi,v}$, the benefit of merging v with each of the parents, p_i, on the earliest start time of v is computed.

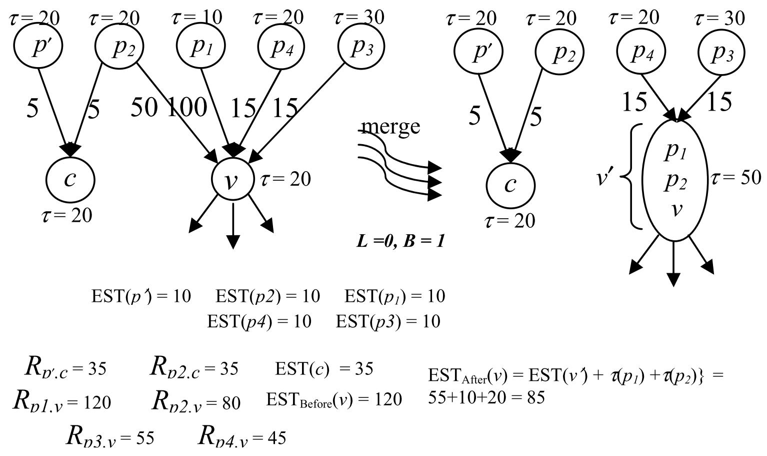
τ= 20
τ= 20
τ= 10
τ= 20
τ= 30
p′
p2
p1
p4
p3
5
5
50
100
15
15
c
τ= 20
v
τ= 20
merge
L =0, B = 1
τ= 20
τ= 20
τ= 20
τ= 30
p′
p2
p4
p3
5
5
15
15
c
τ= 20
v′
p1
p2
v
τ= 50
EST(p′) = 10 EST(p2) = 10 EST(p1) = 10
EST(p4) = 10 EST(p3) = 10
Rp′.c = 35 Rp2.c = 35 EST(c) = 35
Rp1.v = 120 Rp2.v = 80 ESTBefore(v) = 120
Rp3.v = 55 Rp4.v = 45
ESTAfter(v) = EST(v′) + τ(p1) + τ(p2)} =
55+10+20 = 85

Fig. 3 Efficacy of earliest start time

Therefore, node v is merged with the subset of its parents, which reduces its earliest start time, and their number does not precede the number of available processors. The benefit of the merge is described in the following section. As an example, in Fig. 3, the earliest time to collect data from parent nodes is computed by considering the execution time, τ, earliest start time, EST, of each parent node, p_1 to p_4, and the communication costs the earliest start time of node v. Also, applying relation (7), it is decided to merge v with p_1 and p_2. To avoid any increase in the earliest start time of the node c, the node p_2 is duplicated.

3.2.3 Compute the benefit of task merging

When merging a task, v, with one of its parents, p_k, the earliest start time of the other children of p_k may be increased. To resolve the difficulty, p_k may be duplicated. However, the duplication may increase the total execution time of the task graph if after the duplication the number of parallel tasks exceeds the number of available processors. To determine whether to merge a task with a subset or all of its parents, relation (8) can be applied. This relation computes the benefit, Q, of combining a task with a subset of its parents.

$$Q = \rho \times \Delta T - (1 - \rho) \times \Delta E \tag{8}$$

In relation (8), ΔT indicates the amount of reduction in earliest start time of the node v, ΔE is the sum of the execution time of the duplicated parents of v, and finally ρ is a parameter that depends on the number of available processors. If there are enough processors to execute parallel tasks within the task graph, ρ will be equal to 100. Otherwise, if the number of available processors is less than the number of parallel tasks within the task graph, ρ will be a number between 0 and 100. In order to compute the value of ΔT for a node v, we apply relation (7) to compute the value $R_{pi,v}$ for each parent p_i of v. Then the parent nodes of v are sorted by applying relation (9):

$$\forall p_i, p_j \in \text{parents}(v) \ \wedge \ 1 \le i \le j \le n \Rightarrow R_{pi,v} \ge R_{pj,v}. \tag{9}$$

Afterward, starting with the p_1, relation (11) is applied to compute the $Q_{v,1}$. Then the value of $Q_{v,2}$ indicating the benefit of merging the node v with p_1 and p_2. This process is carried on until the maximum value of $Q_{v,k}$, indicating the benefit of merging v with p_1 to p_k is found. In relation (10), P_k indicates the subset p_1 to p_k for which $Q_{v,k}$ is maximum:

$$\begin{aligned} P_k = \big\{ p_i : P \mid 1 \le i \le j \le n \ \wedge \ \forall j < k \Rightarrow Q_{v,j} \le Q_{v,k} \wedge Q_{v,k} > 0 \\ \wedge \ \forall k < r \le n \Rightarrow Q_{v,r} < Q_{v,k} \big\}. \end{aligned} \tag{10}$$

In relation (10), n indicates the number of parents of v and $Q_{v,k}$ represents the benefit of merging v with its parents, p_1 to p_k. In relation (11), $Q_{v,k}$ computes the benefit of merging the task (v) with parents p_1 to p_k. Therefore, k means merging task v with p_1 to p_k where $1 \le k \le n$ and p_1 to p_k are parents that according EST are sorted descending. If the result of $Q_{v,k}$ for all k tasks is zero or less than zero, task (v) will not be merged with any of the parents. The value of $Q_{v,k}$ is computed as follows:

$$Q_{v,k} = \rho \times \Delta T_{v,p_k} - (1-\rho) \times \Delta E_{v,p_k}. \tag{11}$$

In relation (11), $\Delta T_{v,Pk}$ indicates the reduced amount in earliest start time of the node v, where $P_k = \{p_1, p_2, \dots, p_k\}$ and $\Delta E_{v,Pk}$ represents the total execution time of the duplicated parents. The value of $\Delta T_{v,Pk}$ is computed by applying the relation (12):

$$\Delta T_{v,P_k} = EST_v - EST_{v'} - \sum_{p \in P_k} \tau(p). \tag{12}$$

In the above relation, v' addresses the node created by merging v with the parent nodes in P_k and EST_v indicates the earliest start time of the node, v. The value of $EST_{v'}$ is computed by applying relation (13):

$$EST_{v'} = \begin{cases} \text{MAX}_{p \in P_k}(EST_p) & k = n, \\ R_{p_{k+1},v'} & k < n, \end{cases} \quad \text{where } n \text{ is the number of the parents.} \tag{13}$$

In order to compute the value of $\Delta E_{v,Pk}$, the set of siblings of v, $\Psi_{v,Pk}$, whose earliest start time increases after the merge is selected, firstly. Then the nodes $p_i \in P_k$, which are also parents of one or more nodes in $\Psi_{v,Pk}$, are duplicated. The value of $\Delta E_{v,Pk}$ is computed by applying relation (14) as the sum of the execution time of the nodes p_i:

$$\Delta E_{v,Pk} = \sum \tau(p_i), \quad \forall p_i \in \big\{ P_k \cap \text{Parents}(\Psi_{v,Pk}) \big\}. \tag{14}$$

The proposed algorithm attempts to increase parallelism in the execution of a given task graph by reducing the earliest start time of the tasks within the task graph. The earliest start time of a task is dependent on the completion time of its parent nodes and the time it takes to receive the results from the parents.

3.3 Applying game theory

There are two noncooperator games. The first game is for determining appropriate number of processors and the second game for determining merging tasks.

3.3.1 Game for determining appropriate number of processors

First, we make categories (group) for tasks according to their level and find the number of tasks for every level (categories or group) and assume every level as a player. The strategies that every player can choose are in range of 1 to $N_{\max}$. In other words, in order to execute its tasks, every level can choose from 1 to $N_{\max}$ processors. So there are $N_{\max}$ choices for finding the number of processor is explained in detail in Sect. 3.1.2.

The algorithm of determining the appropriate number of processors based on game theory:

1. For all tasks in the graph (G), find the level (L) of every task where n is the number of tasks, $i = 1, 2, \dots, n$. Consider the level of root is one as every child is at the next level of parent and level for current task increases one unit.
2. Find the number of tasks in every level and put them in the variable N_{Li}, as L is the number of levels.
3. Consider every level as one player in a way that $L = \{L_1, L_2, \dots, L_L\}$, $1 \le i \le n$.
4. Consider that S is set of strategies that every player can choose it. In other words every player can choose one of strategies (number of processors) in a way that the range is from 1 to $N_{\max}$ $S = \{1, 2, \dots, N_{\max}\}$.
5. Calculating utility that for every player (level) is as follow: $U(L_i, L_{-i}) = (N_L - \mu)^* L(\mu)$.
6. The best answer will obtain if $U(L_i^*, L_{-i}) \ge U(L_i, L_{-i}^*)$.

   In other words, the most appropriate number of processors for the level i faced to the in relation to best choice of opponents that have the most utility for i level (level of $-i$).
7. Calculating the maximum of utility as $\max U(L_i, L_{-i}) = \max[(N_i - \mu)^* L(\mu)]$.
8. According to calculations in Sect. 3.1.2: $N_i = (\mu^2 + N_{\max}^2)/(2\mu)$, $N^* = N_k^*(\mu/N_{\max})$.

For executing the graph tasks, N^* is the appropriate number of processors that can reach to the balance to reduce CPU from being idle.

3.3.2 Game for determining merging tasks

Suppose there are n tasks v_1 to v_n in a task graph G. Assume each task in a task graph as a player, all the players attempt to reduce their earliest start time by merging with one or more of their parents. If a player v_i has k parents, as shown in Fig. 3, it may apply $k+1$ different strategies $s_{i,0}$ to $s_{i,k}$ to merge with its parent nodes. The parents from p_1 to p_k of each node v_i are sorted in descending order of their communication time with v_i. The strategy $s_{i,j}$ indicates the merge node v_i with the parents p_1 to p_k $(1 \le i \le n,\ 1 \le j \le k)$. When merging a node v_i, with each of its parents p_r, if the earliest start time of the sibling of v_i increases, then the merge is made with the copy of p_r. Considering relation (11), the benefit, $Q_{vi,j}$, of merging the node (player) v_i with parents (from 1 to j) the utility, $U(v_i, v_{-i})$ of the player v_i is computed as follows:

Merging tasks algorithm based on game theory:

1. For all tasks of v_i in graph G $(1 \le i \le n)$ the following drill must be done:
2. Let $V = \{v_1, v_2, \ldots, v_n\}$ as a set (group) of players where $1 \le i \le n$ and n is the number of tasks in graph.
3. There are several strategies for every task. In a way that S be set of strategies for tasks: $S = \{S_1, S_2, \ldots, S_n\}$, $1 \le i \le n$, where $S_i = \{s_{i,0}, s_{i,1}, \ldots, s_{i,k}\}$ indicates the set of strategies of the player v_i for merge with selfish parents ($s_{i,0}$ indicate do not merge, $s_{i,1}$ indicate to merge v_i with $p_{i,1}$, $s_{i,k}$ indicate to merge v_i with $p_{i,1}, p_{i,2}, \ldots, p_{i,k}$). Therefore, $P_i = \{p_{i,1}, p_{i,2}, \ldots, p_{i,k}\}$ the set of parents of v_i, sorted on their descending order of earliest start time (EST).

Apply relation (11):

$$U(v_i, v_{-i}) = Q_{vi,j} = \rho^* \Delta T v_i, s_{i,j} - (1-\rho)^* \Delta E v_i, s_{i,j}.$$

The highest utility is achieved:

$$U\big(v_i^*, v_{-i}^*\big) \ge U\big(v_i, v_{-i}^*\big).$$

To achieve this:

$$\mathrm{Max} U\big(v_i, v_{-i}^*\big) = \mathrm{Max}(Q_{vi,j}) = \mathrm{Max}\big(\rho^* \Delta T v_{i,s_{i,j}} - (1-\rho)^* \Delta E v_{i,s_{i,j}}\big).$$

3.4 PSGT algorithm

In this section, our algorithm, PSGT (Prescheduling and Scheduling using Game Theory), is presented. The algorithm accepts a task graph, G, as its input and outputs a graph, G', which is the appropriate algorithm for scheduling and Total Finish Time (TFT).

Input: a task graph $G(V, E, \tau, c)$
where V: Set of tasks, E: Set of task interconnection lines
τ: A function to compute the execution cost of each task
c: A function to compute the communication costs
Output: New graph $G'(V', E', \tau', c')$ with the merging tasks and appropriate number of processors & Total Finish Time (TFT)

```
Begin (main)
   Compute $\rho = N^*/N_{\max})^*100$ based on relation (4)
   Begin merge
L1: For each $v$ in $V$ do
      Apply relation (6) to compute $EST(v)$
      For each $p_i$ in parents$(v)$ do
         Apply relation (7), to compute the earliest time, $R_{pi,v}$, to collect data
         from parent, $p_i$, on $v$
         Sort the parents, $p_i$, on the value of $R_{pi,v}$, descending, Giving the
         sequence $P = (p_1, p_2, \dots, p_n)$;
      End for;
      Apply relation (11) to compute the benefit $Q_{v,k}$, of merging $v$ with its
      parents $p_1$ to $p_k$ for $1 \le k \le n$;
      If there are no $Q_{v,k} > 0$ then it is not possible to merge $v$ with any subset
      of its parents
         go to L1;
      End if;
      Find the subsequence $P_k = \{(p_1, p_2, \dots, p_k)$ in $P$, $1 \le k \le n\}$ such that
      $Q_{v,k}$ is maximum;
      Let $\Psi_{v,Pk}$ be the set of siblings of $v$ whose earliest start time increases
      Let $D_{v,pk}$ be the set of parent nodes $p_i$ in $P_k$ with at least one child in
      $\Psi_{v,Pk}$
      For each node $p$ in $D_{v,pk}$ do
         Let $p'$ be a copy of $p$
         Remove any edges from $p$ to nodes in $\Psi_{v,Pk}$
         Remove any edges from $p'$ to nodes that not in $\Psi_{v,Pk}$
      End For;
    End For;
   End (merge)
   Apply scheduling algorithm on new graph by $N^*$ processors
End (main).
```

4 Experimental results

In this section, the second step of our proposed algorithm (prescheduling), PSGT, is firstl compared with a known algorithm, GRS [7, 8]. Then the length of the critical paths and degree of parallelism in the resultant task graphs are compared. The experiments have been carried out on a microcomputer with Intel(R) Core(TM) i5 CPU M480, 4 GB of RAM (64-bit). All the programs are in C# and developed within the Microsoft Visual Studio 2010.

4.1 First step of PSGT (processors cost)

The applicability of PSGT in predicting the appropriate number of processors for scheduling 6 benchmark task graphs is shown in Table 1. As shown in Table 1, on

Table 1 The effect of applying PSGT to estimate the appropriate number of processors for 6 shown benchmarks

Task graph	Completion time with Max. number of processors are available	Completion time when the number of processors are computed by PSGT	Percentage of reduction in the number of processors	Percentage of increase in execution time of the task graph
FFT1	148	149	0.25	0.007
FFT2	205	255	0.25	0.243
FFT3	2350	2360	0.25	0.004
FFT4	710	730	0.25	0.028
IRR	761	785	0.29	0.031
Standard	44	44	0.34	0.0

Table 2 The effect of applying PSGT to estimate the appropriate number of processors for random graphs and new graphs

Task graph	Completion time with Max. number of processors are available	Completion time when the number of processors are computed by PSGT	Percentage of reduction in the number of processors	Percentage of increase execution time of the task graph
20 Random graphs	10205	9813	0.31	0.04
20 New graphs	9173	8906	0.28	0.03

average, the execution time of the task graphs are increased about 5.2 % while the number of parallel processors is reduced by 27.2 %.

The applicability of PSGT in predicting the appropriate number of processors for scheduling 20 random task graphs and 20 new task graphs is shown in Table 2. As shown in Table 2, on average, the execution time of the task graphs are increased about 3.5 % while the number of parallel processors is reduced by 29.5 %.

In Fig. 4, the STD (standard) benchmark task graph is presented. This task graph has been prepared for scheduling by applying the PSGT algorithm with two different values of the parameter ρ for two different numbers of processors. The resultant task graphs are shown in Figs. 5 and 7. The resultant scheduling of task graphs are shown in Figs. 6 and 8.

4.2 Second step of PSGT (merging cost)

In Table 3, the critical paths of 6 benchmark task graphs[18], restructured by applying the PSGT algorithm are compared with their critical paths resulted from applying the GRS algorithm, and in Table 4 the degree of parallelism of the resultant task graphs are compared. To compute the parallelism degree of a task graph, the sum of the

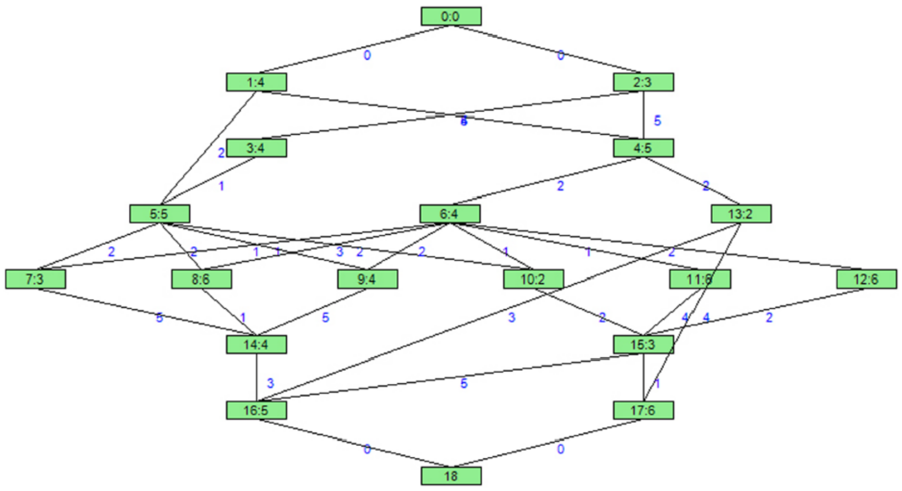

Fig. 4 STD (standard), a task graph benchmark

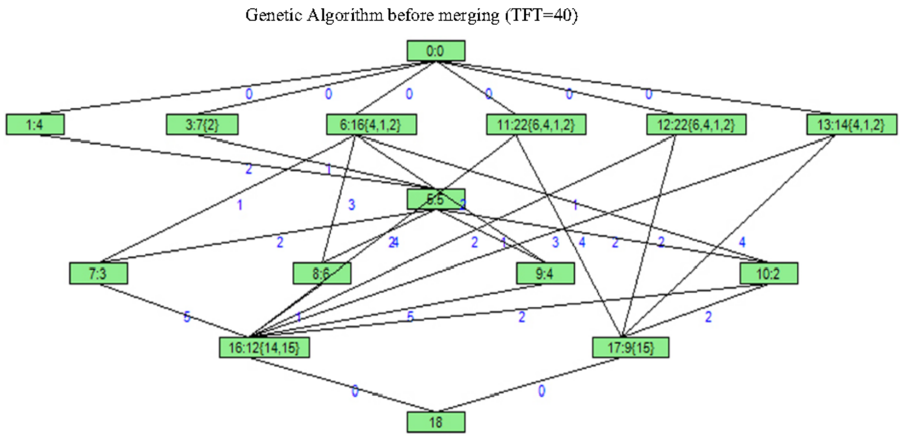

Fig. 5 STD (standard), a task graph benchmark with maximum processors ($\rho = 100$)

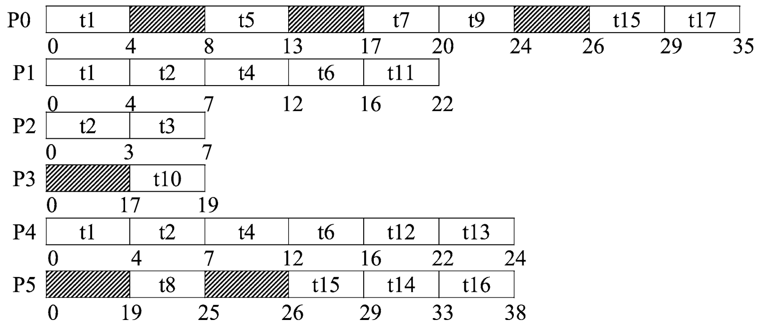

Fig. 6 Standard (STD) graph with maximum processors ($\rho = 100$). Genetic algorithm after merging RTM (TFT = 38)

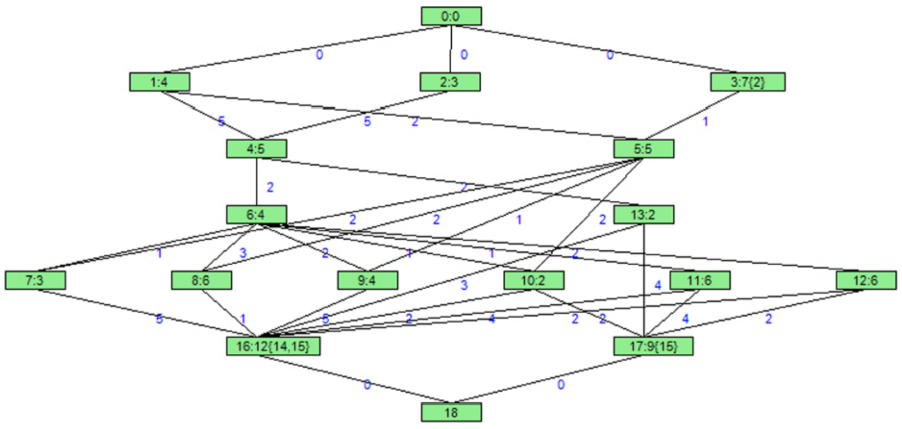

Fig. 7 STD (standard), a task graph benchmark with appropriate processors ($\rho = 66$)

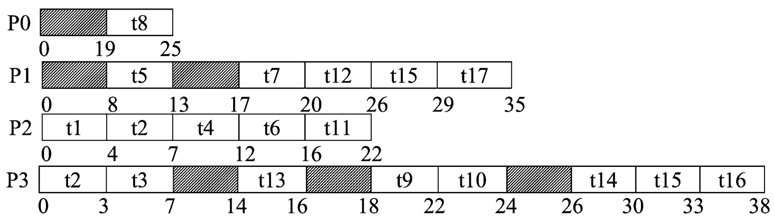

Fig. 8 Standard (STD) graph with appropriate processors ($\rho = 66$). Genetic algorithm after merging PSGT (TFT = 38)

Table 3 The length of critical paths resulted form three prescheduling algorithms

Graph	FFT1	FFT2	FFT3	FFT4	IRR	STD
Input T.G.	178	225	2838	710	796	44
GRS	123	184	1848	130	696	44
PSGT	**120**	**180**	**1640**	**125**	**621**	**39**

execution time of all the tasks within the task graph is divided into the critical path length of the task graph.

The GRS algorithm uses three separate rules for preparing a task graph. Applying these rules in different orders may result different in task graphs. To compute the critical path length and parallelism of the resultant task graphs, a brute-force algorithm has been applied. The PSGT algorithm uses one rule (merge rule) for preparing a task graph. Therefore, comparing GRS with PSGT, the GRS takes a long time to prepare task graphs for scheduling. Also, as shown in Tables 3 and 4, PSGT produces task graphs with shorter critical path length and higher parallelism. As an example, the FFT4 graph modified by PSGT is presented in Fig. 9.

Table 4 The degree of parallelism achieved by applying three prescheduling algorithms

Graph	FFT1	FFT2	FFT3	FFT4	IRR	STD
Input T.G.	1.72	3.3	1.2	0.6	1.6	1.6
GRS	3.05	4.66	2.47	4.32	2.61	1.71
PSGT	**3.86**	**4.66**	**3.47**	**5.28**	**3.07**	**3.24**

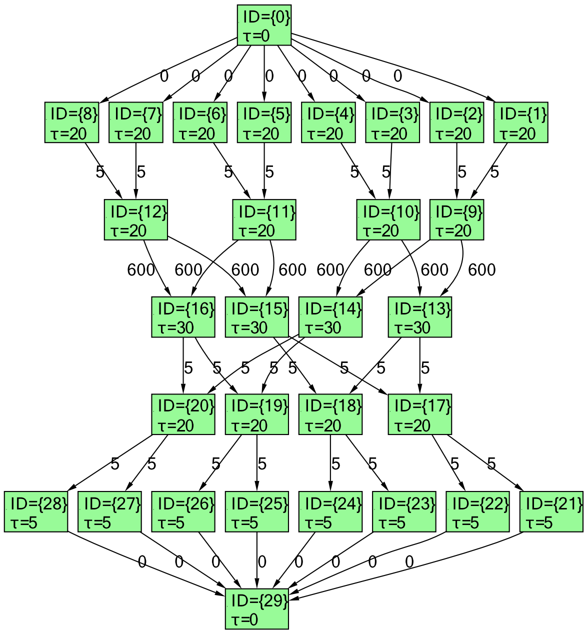

Fig. 9 FFT4, a task graph benchmark with communication cost (CPL = 710, Parallelism = 0.6)

4.3 PSGT algorithm vs. other algorithms

Applying PSGT algorithm to schedule five benchmark task graphs, the parallel execution times of the resultant scheduling were compared with the execution times resulted by applying 7 known scheduling algorithms. These scheduling algorithms are CLANS [9], DSC [10], MCP [11], LC [12], LAST [13], SR [14] and Genetic [17] and the benchmark task graphs are FFT1-4 [15] and IRR [14]. In Table 5, the comparison results are presented.

The applicability of PSGT in predicting the merging of tasks for scheduling 6 benchmarks task graphs is shown in Table 6. As shown in Table 6, on average, the execution time of the task graphs after merging is reduced by 100.6 %.

In Fig. 9, the FFT4 benchmark task graph is presented. This task graph has been prepared for scheduling by applying the PSGT algorithm with two different values of the parameter ρ for two different numbers of processors. The resultant task graphs are shown in Figs. 10 and 12. Figure 11 illustrates the scheduling of the modified task graphs.

Table 5 The effect of PSGT on preparing 5 benchmark task graphs for optimal scheduling

Graph	CLANS	DSC	MCP	LC	LAST	SR	Genetic	PSGT
FFT1	124	124	148	127	146	146	173	**120**
	4	4	8	8	1	–	3	**10**
FFT2	200	205	205	225	240	215	225	**180**
	8	8	8	8	8	–	20	**8**
FFT3	1860	1860	2350	2838	2220	–	1992	**1640**
	4	4	8	8	2	–	2	**8**
FFT4	405	710	710	710	170	160	160	**125**
	2	12	8	8	8	–	8	**8**
IRR	725	605	605	710	840	680	730	**546**
	7	12	7	8	3	–	3	**10**

Table 6 The effect of applying PSGT to merging of tasks for 6 shown benchmarks

Task graph	Completion time with Max. number of processors after merging	Completion time with Max. number of processors before merging	Percentage of reduction execution time of the task graph
FFT1	120	148	0.233
FFT2	180	205	0.139
FFT3	1640	2350	0.433
FFT4	125	710	4.680
IRR	546	761	0.394
Standard	38	44	0.158

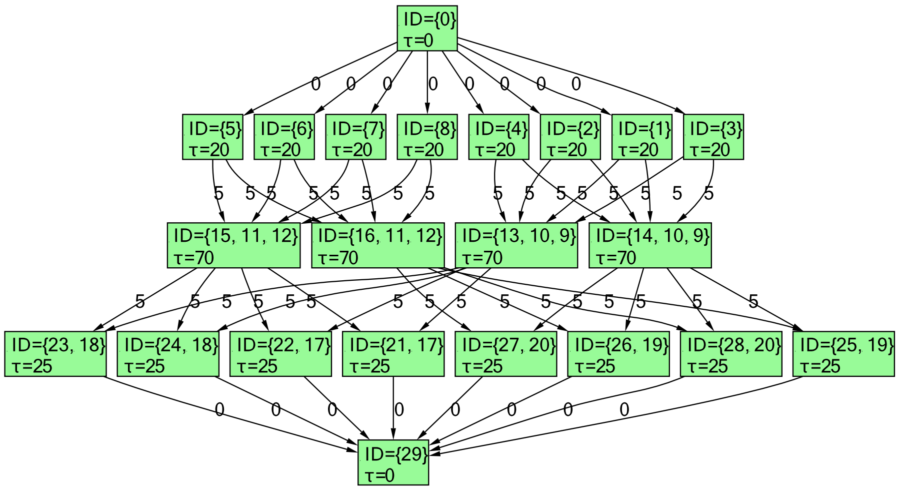

Fig. 10 Optimized FFT4 task graph by applying PSGT algorithm with $\rho = 100$ (CPL = 125, Parallelism = 5.28)

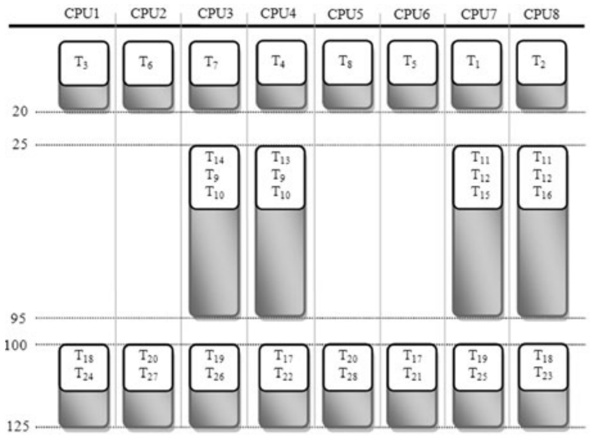

Fig. 11 Scheduling of optimized FFT4 on eight processors

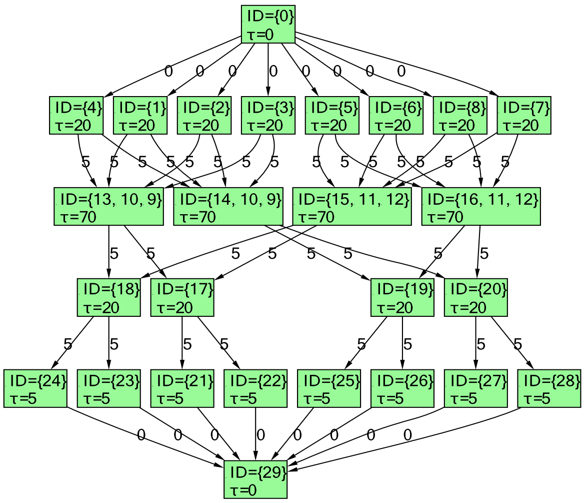

Fig. 12 FFT4 task graph re-structured by applying the PSGT algorithm with $\rho = 25$ (CPL = 130, Parallelism = 4.30)

Suppose there are eight parallel processors ($\rho = 100$) and the interprocessors communication baud rate is 1 and the communication latency equals zero. Comparing the modified graph with the original FFT4 graph, it is observed that the critical path length is reduced about 200 %. Also, the parallelism of the modified graph is about 88 % higher than the original one. In Fig. 11, the result of scheduling the modified task graph is presented. The parallel execution time of this scheduling is 125. As shown in Table 5, this execution time is much less than the execution times resulted from other known scheduling algorithms.

Assuming that the execution environment has just two processors and the interprocessors communication baud rate is 1, and the communication latency equals zero. The value of ρ for the FFT4 task graph will be 25. The resultant modified FFT4 task graph is shown in Fig. 12.

Table 7 Experimental setup

Problem	No. of tasks	Computation time	Communication time
Gauss–Jordan elimination	15, 21, 28, 36	40 s/task	100 s
LU factorization	14, 20, 27, 35	10 s bottom layer task, plus 10 s for every layer	80 s

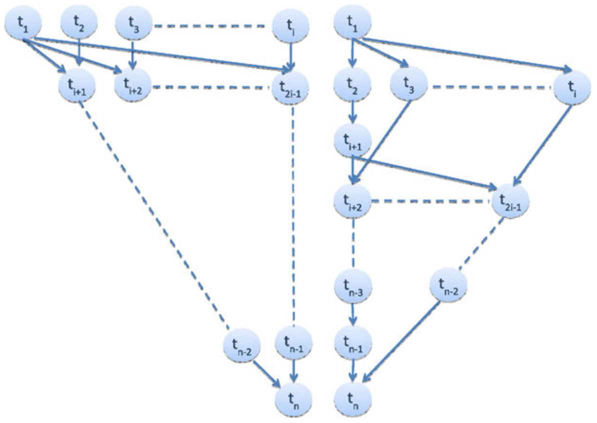

Fig. 13 The task graph for the Gauss–Jordan elimination algorithm (*left side*). The task graph for the LU decomposition algorithm (*right side*)

4.4 PSGT vs. other algorithms

Recently, the challenge is about the many number of tasks and processors. In order to be sure that the proposed algorithm presented a good result for much number of tasks and processors. We have run all ten algorithms on five instances of various sizes of both the Gauss–Jordan elimination problem and the LU factorization problem. In our simulation, we assume the number of processors is four. The problem sizes, the computation, and communication time considered were summarized in Table 7. For the nondeterministic scheduling algorithms, the simulations were run 100 times and the average values reported. The number of tasks we chose for LU and GJ algorithms is based on their “layered” structure diagrams shown in Fig. 13, respectively. Let us explain the reasoning behind the number of tasks chosen for our performance study. We use a number of dashed lines to help explain the chosen tasks’ numbers. Figure 14 is an example of the LU algorithm with 35 tasks. Each layer starts with one task followed by a number of tasks. The LU diagram increments from layer L1 into L2, L3, L4, L5, and L6, thus the number of task increasing from 9 into 14, 20, 27, and 35, respectively. Also, Fig. 14 shows an example of the GJ algorithm with 36 tasks. The numbers of tasks we chose in the GJ algorithm is more straightforward. Layers L0, L1, L2, L3, L4, L5, L6, L7 contain 3, 6, 10, 15, 21, 28, and 36 tasks, respectively. Therefore, the total numbers of tasks between layer L0 and L1, L2, L3, L4, L5, L6, and L7 are 3, 10, 15, 21, 28, and 36.

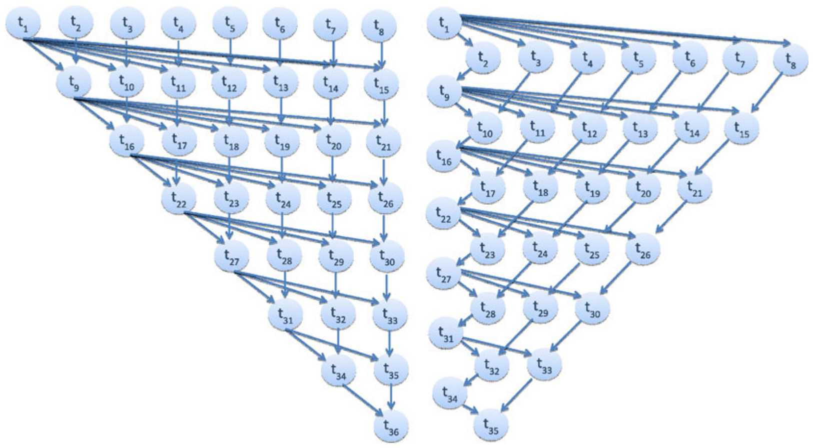

Fig. 14 The task graph for GJ algorithm with 36 tasks (*left side*). The task graph for LU decomposition algorithm with 35 tasks (*right side*)

Table 8 GA parameters

Parameters	Value
Population size	30
Mutation rate	0.01
Crossover rate	0.7
Stopping criteria	Stops when it converges with in a predefined threshold or when the max. no. of generation exceeds 6,000

Considering our assumption of four processors, a number of tasks longer than 35 would not yield qualitative differences, especially considering that the additional tasks are the same type as the previous ones. As discussed thus far, PSGT is an effective way to decrease a makespan. We use elitist selection, meaning that the best individual always survives in one generation, to keep the best result into the next generation. All GA parameters of this simulation are shown in Table 8.

4.4.1 Simulation metrics

The quality of results is measured by two metrics: makespan and computational cost. The makespan is represented by the time required to execute all tasks; the computational cost is the execution time of an algorithm. A good scheduling algorithm should yield short makespan and low computational cost. In our simulation, we evaluate Gauss–Jordan elimination and LU factorization algorithms based on different number of tasks, with the former increasing from 15, 21, 28 to 36 and the latter from 14, 20, 27 to 35. Other parameters such as task computation time and intercommunication delays are given in Table 7. An ideal algorithm must perform well on different problems with different sizes.

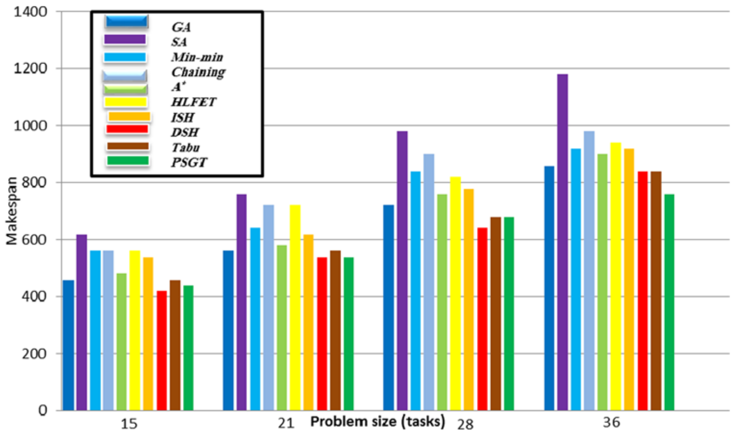

Fig. 15 Makespan for Gauss–Jordan elimination problem for variable task sizes and the ten scheduling algorithms

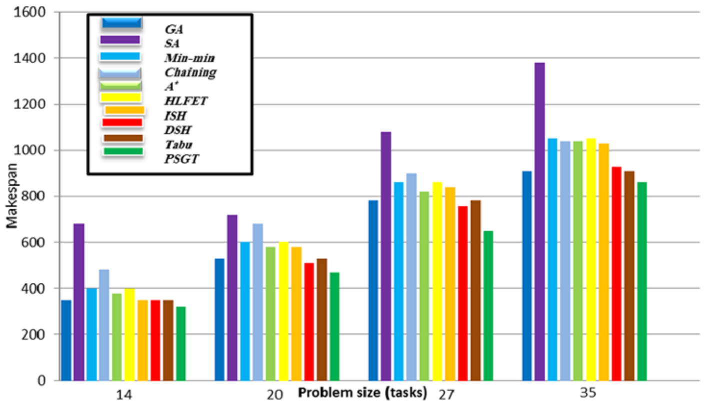

Fig. 16 Makespan for LU factorization problem for variable task sizes and the ten scheduling algorithms

Simulation results The makespan of the obtained solutions are represented on Fig. 15 for the Gauss–Jordan elimination and on Fig. 16 for the LU factorization. The execution time of the algorithms is presented in Fig. 17 for the Gauss–Jordan elimination, and in Fig. 18 for the LU factorization.

5 Conclusion

Multiprocessors systems were the start of a revolution in computation with high utility, and caused remarkable change in computation. In this article, we scheduled for homogenous processors and task graphs are assumed to have only one starting and one ending node. There could be different strategies to reach from the starting point to the ending point. Attention to two aims of minimizing energy consumption by minimizing the number of idle processors and minimizing scheduler length makespan

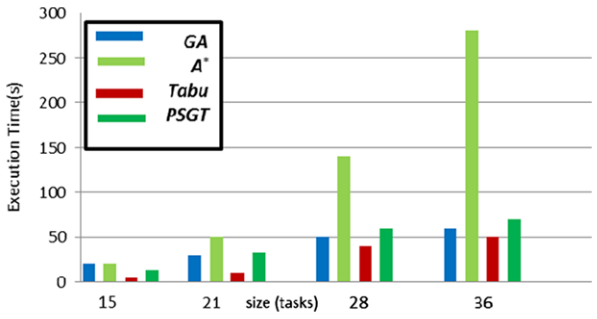

Fig. 17 Execution time of the four scheduling algorithms for the Gauss–Jordan elimination problem of variable task sizes

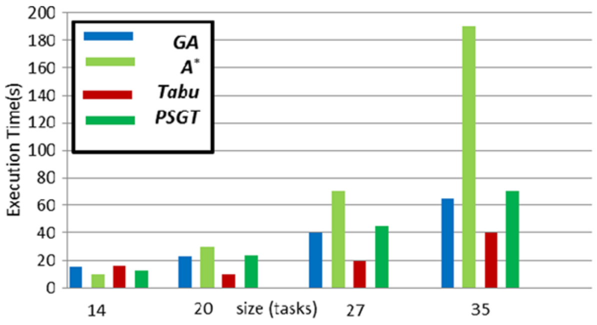

Fig. 18 Execution time of the four scheduling algorithms for the LU factorization problem of variable task sizes

that a specific strategy will choose. Therefore, the purpose of prescheduling is that our suggested solution according to games theory is minimizing schedule length and determining the number of processors for minimizing idle processors. Recently, researchers and workmen despite multiprocessor systems can reach higher goals.

Prescheduling of task graphs could highly affect the scheduling of the tasks on parallel processors. Pre-scheduling begins with determining the appropriate number of processors for executing the tasks. The idea of Nash equilibrium in game theory could be very vital for determining the appropriate number of parallel processors to execute the tasks at each level of the task graph in parallel. Task merging techniques can be applied to minimize the length of the critical paths while maximizing the parallelism. In this article, a new algorithm is presented. The algorithm suggests the appropriate number of processors at each level in a way that the idle time of the processor reduces and merges each task with a subset of its parents in a way that the earliest start time of the task reduces and the start time of the siblings of the task does not increase after the merge. To avoid any increase in the start time of the siblings, those parent nodes which are supposed to be merged with their children, are duplicated. When duplicating a parent node, the number of tasks apparently increases. Therefore, there should be an adequate number of available processors to execute the parallel tasks, otherwise the duplication, and thereby the merge will not be beneficial.

References

1. Jin S, Schiavone G, Turgut D (2008) A performance study of multiprocessor task scheduling algorithms. J Supercomput 43:77–97. doi:10.1007/s11227-007-0139-z

2. ChoonLee Y, Zomaya A, Siegel H (2010) Robust task scheduling for volunteer computing systems. J Supercomput 53:163–181. doi:10.1007/s11227-009-0326-1
3. Afgan E, Bangalore P, Skala T (2012) Scheduling and planning job execution of loosely coupled applications. J Supercomput 59:1431–1454. doi:10.1007/s11227-011-0555-y
4. Ababneh I, Bani-Mohammad S, Ould-Khaoua M (2010) An adaptive job scheduling scheme for mesh-connected multi computers. J Supercomput 53:5–25. doi:10.1007/s11227-009-0333-2
5. Niemi T, Hameri A (2012) Memory-based scheduling of scientific computing clusters. J Supercomput 61:520–544. doi:10.1007/s11227-011-0612-6
6. Qureshi K, Majeed B, Kazmi JH, Madani SA (2012) Task partitioning, scheduling and load balancing strategy for mixed nature of tasks. J Supercomput 59:1348–1359. doi:10.1007/s11227-010-0539-3
7. Aronsson P, Fritzson P (2005) A task merging technique for parallelization of modelica models. In: 4th international modelica conference, Hamburg
8. Aronsson P, Fritzson P (2003) Task merging and replication using graph rewriting. In: 2nd international modelica conference, Germany
9. Aronsson P, Fritzson P (2002) Multiprocessor scheduling of simulation code from modelica models
10. Parsa S, Lotfi S, Lotfi N (2007) An evolutionary approach to task graph scheduling. In: Lecture notes in computer science, vol 4431, p 110
11. Kim S, Browne J (1988) A general approach to mapping of parallel computation upon multiprocessor architectures. In: International conference on parallel processing
12. McCreary C, Gill H (1989) Automatic determination of grain size for efficient parallel processing. Commun ACM 32(9):1073–1078
13. Yang T, Gerasoulis A (1994) DSC: scheduling parallel tasks on an unbounded number of processors. IEEE Trans Parallel Distrib Syst 5(9):951–967
14. Wu M, Gajski D (1990) Hypertool: a programming aid for message-passing systems. IEEE Trans Parallel Distrib Syst 1(3):330–343
15. Baxter J, Patel J (1989) The LAST algorithm—a heuristic-based static task allocation algorithm. In: International conference on parallel processing
16. Parsa S, Reza Soltan N, Shariati S (2010) Task merging for better scheduling. In: Lecture notes in computer science, pp 311–316
17. Abdeyazdan M, Rahmani AM (2008) Multiprocessor task scheduling using a new prioritizing genetic algorithm based on number of task children. In: Distributed and parallel system, pp 105–114
18. Kwok YK, Ahmad I (1999) Benchmarking and comparison of the task graph scheduling algorithms. J Parallel Distrib Comput 59:381–422
19. Ahmad I, Ranka S (2008) Using game theory for scheduling tasks on multi-core processors for simultaneous optimization of performance and energy. IEEE 27(2):177–194
20. ChouLai K, Yang CT (2008) A dominant predecessor duplication scheduling algorithm for heterogeneous systems. J Supercomput 44:126–145. doi:10.1007/s11227-007-0152-2
21. Gairing M, Lücking T, Mavronicolas M, Monien B (2010) Computing Nash equilibria for scheduling on restricted parallel links. Theory Comput Syst 47(2):405–432
22. Dummler J, Kunis R, Runger G (2012) SEParAT: scheduling support environment for parallel application task graphs. Clust Comput, online first
23. Baskiyar S, Abdel-Kader R (2010) Energy aware DAG scheduling on heterogeneous systems. Clust Comput 13(4):373–383
24. Wang W, Mok AK, Fohler G (2005) Pre-scheduling. Real-Time Syst 30(1–2):83–103
25. Bonyadi MR, Moghaddam ME (2009) A bipartite genetic algorithm for multi-processor task scheduling. Int J Parallel Program 37(5):462–487
26. Li K (2012) Energy efficient scheduling of parallel tasks on multiprocessor computers. J Supercomput 60(2):223–247
27. Kwok YK (2001) Fault-tolerant parallel scheduling of tasks on a heterogeneous high-performance workstation cluster. J Supercomput 19(3):299–314
28. Tosun S (2011) Energy- and reliability-aware task scheduling onto heterogeneous MPSoC architectures. J Supercomput, online first
29. Nisan N, Roughgarden T, Tardos E, Vazirani V (2007) In: Algorithmic game theory. Cambridge University, New York, pp 301–330
30. Sinnen O, Sousa L (2004) On task scheduling accuracy: evaluation methodology and results. J Supercomput 27(2):177–194
31. Mc Kelvey RD, Mc Lennan A (1996) Computation of equilibria in finite games. In: Computational economics, vol 1, pp 87–142

32. Nash JF (1951) Non-cooperative games. Ann Math 54(2):286–295
33. Papadimitriou CH (2001) Algorithms, games and the Internet. In: Proceedings of the 33rd annual ACM symposiumon theory of computing, pp 749–753
34. Aydin H, Melhem R, Moss D, Meja-Alvarez P (2004) Power-aware scheduling for periodic real-time tasks. IEEE Trans Comput 53(5):584–600
35. Deb K (2001) Multi-objective optimization using evolutionary algorithms. Wiley, New York
36. Kang J, Ranka S (2008) Dynamic algorithms for energy minimization on parallel machines. In: Euromicro international conference on parallel, distributed and network-based processing
37. Khan SU, Ahmad I (2006) Non-cooperative, semi-cooperative and cooperative games-based grid resource allocation. In: International parallel and distributed processing symposium
38. Schmitz MT, Al-Hashimi BM (2001) Considering power variations of DVS processing elements for energy minimisation in distributed systems. In: International symposium on system synthesis, pp 250–255
39. Weichen L, Zonghua G, Jiang X, Xiaowen W, Yaoyao Y (2010) Satisfiability modulo graph theory for task mapping and scheduling on multiprocessor systems. Digital object indentifier. IEEE, New York
40. Agarwal A, Kumar P (2009) Economical duplication based task scheduling for heterogeneous and homogeneous computing systems. In: WEE international advance comnputing conference (IACC)
41. Thang NK (2010) NP-hardness of pure Nash equilibrium in scheduling and connection games. In: Proceedings of the 35th international conference on current, trends in theory and practice of computer science (SOFSEM), 2009