Evaluation of Multi-Objective Decentralized Scheduling for Applications in Grid Environment

Florin Pop, Ciprian Dobre, Valentin Cristea
Faculty of Automatics and Computer Science, University "*Politehnica*" of Bucharest
Emails: {florinpop, cipsm, valentin}@cs.pub.ro

Abstract

In Grid environments, various applications require dynamic scheduling for optimized assignment of tasks. Optimization technique represents the key solution for scheduling. This paper presents evaluation of multi-objective decentralized scheduling models for the problem of task allocation. The paper presents a survey of existing optimization solution for Grid scheduling. The tested solutions are: Random and Best of n Random, Exhaustive Search, Simulated annealing, Game Theory, Ad-Hoc Greedy Scheduler, and Genetic Algorithm for Decentralized Scheduling. We carry out our experiments with complex scheduling scenarios and with heterogeneous input tasks and computation resources. We also use simulation methods to tests and validate scheduling methods. The experimental results offer a support for near-optimal algorithm selection.

1. Introduction

Developing high quality schedules for distributed applications on a Computational Grid is a challenging problem. To provide a faster scheduling algorithm the aproximation methods are required. The approximate algorithms use formal computational models, but instead of searching the entire solution space for an optimal solution, they are satisfied when a solution that is sufficiently "good" is found. In the case where a metric is available for evaluating a solution, this technique can be used to decrease the time taken to find an acceptable schedule. The factors which determine whether this approach is worthy of pursuit include [1]: availability of a function to evaluate a solution, the time required to evaluate a solution, the ability to judge the value of an optimal solution according to some metric, availability of a mechanism for intelligently pruning the solution space.

The other branch in the suboptimal category is called heuristic. This branch represents the class of algorithms which make the most realistic assumptions about a priori knowledge concerning process and system loading characteristics. It also represents the solutions to the scheduling problem which cannot give optimal answers but only require the most reasonable amount of cost and other system resources to perform their function. The evaluation of this kind of solution is usually based on experiments in the real world or on simulation. Not restricted by formal assumptions, heuristic algorithms are more adaptive to the Grid scenarios where both resources and applications are highly diverse and dynamic, so most of the algorithms to be further discussed are heuristics.

In Grid scheduling, in the case that all information regarding the state of resources and the jobs is known, an optimal assignment could be made based on some criterion function, such as minimum makespan and maximum resource utilization. But due to the NP-Complete nature of scheduling algorithms and the difficulty in Grid scenarios to make reasonable assumptions which are usually required to prove the optimality of an algorithm, current research tries to find suboptimal solutions, which can be further divided into the following two general categories: approximate and heuristic algorithms. Scheduling mechanisms for Grid environment use different methods: Random and Best of n Random, Exhasutive Search, Simulated annealing, Game Theory, Ad-Hoc Greedy, Genetic Algorithms, method for task dependencies. Optimization of Distributed Scheduling is based on some theoretical methods for local or global optimization (Multidimensional Optimization Methods): Simplex Algorithm, Simulated Annealing, Genetic Algorithms, Clustering Methods, Heuristic Methods.

All this methods consider Grid from two point of view: like a hole, and try to find a global opimum for scheduling problem, or, like a distributed system and try to find la local optimum for job execution. In or-

der for jobs to be executed a number of tasks need to be performed: appropriate locations for data and execution need to be identified, the data required for the task needs to be moved to the appropriate location, any binary executable files need to be transferred to their appropriate location, along with their parameters, each computer needs to perform authentication and authorisation that the job is allowed, the job needs to be executed and, finally, any output needs to be returned to the user. All these tasks could be subject of optimization methods.

This paper presents evaluation of multi-objective decentralized scheduling for applications in Grid environment. The paper is structured as follows: Section 2 is a general presentation of the optimization methods for decentralized dcheduling. Section 3 describes the near-optimal scheduling models. We describe and comment on the experimental results in the 4th section. Section 5 contains conclusions about "best" methods for grid scheduling optimization and directions for future research.

2. Optimization Methods for Decentralized Scheduling

Optimization methods for decentralized scheduling in Grid environment use heuristic (multi-objective) approaches. We present in this section the opportunistic load balancing heuristic, methods based on minimul execution time, minimum completion time, min-min, max-min, duplex, genetic algorithms, simulating annealiing, A*.

Opportunistic Load Balancing (OLB). The Opportunistic Load Balancing heuristic picks one task arbitrarily from the group of tasks and assigns it to the next machine that is expected to be available [2][3][4]. It does not consider the task's expected execution time on that machine, which may lead to very poor maxspans. The advantages of this heuristic are its simplicity and the intention of keeping all the machines as busy as possible, which means a high processor utilization. In tasks that come one at a time, rather than in groups of tasks, the Opportunistic Load Balancing heuristic is also named First Come First Served strategy.

Minimum Execution Time (MET). The Minimum Execution Time heuristic assigns each task picked arbitrarily to the machine with the least expected execution time for that task, and is not concerned with the time the machine becomes available [5]. The result can be severe load imbalance across machines, although MET gives each task to its best machine.

Minimum Completion Time (MCT). The Minimum Completion Time heuristic assigns each task, in arbitrary order, to the machine with the minimum expected completion time for that task [5]. The MCT combines the benefits of OLB and MET, and tries to avoid the circumstances in which OLB and MET perform poorly.

Min-min. The Min-min heuristic begins with the set T of all unmapped tasks. Then, the set C of minimum possible completion times of all tasks on any of the machines is computed $t = \min_{1 \leq n \leq N_t, 1 \leq j \leq N_m}(t_{i,j})$, where N_t is the total number of tasks, and N_m is the total number of machines. $t_{i,j}$ represents the time of execution of task i on machine j. The task with the minimum possible execution time is then assigned on the respective processor, after which the process continues in the same way with the remaining unmapped tasks.

The major difference between Min-min and MCT is that Min-min considers all unmapped tasks during each mapping decision and MCT only considers one task at a time.

The machine that finishes the earliest, is also the machine that executes the task the fastest. The percentage of tasks assigned to their first choice (on the basis of execution time) is likely to be very high, and therefore a smaller maxspan can be obtained.

Max-min. The Max-min heuristic is very similar to Min-min. The Max-min heuristic also begins with the set T of all unmapped tasks. Then, the set C of minimum completion times is found. The difference from Min-min comes at the next step, when the task with the overall maximum completion time from C is selected and assigned to the corresponding machine. Last, the newly mapped task is removed from C, and the process repeats until C is empty [5][6].

Max-min tries to perform tasks with longer execution times first, which usually leads to a better balanced allocation of tasks, and prevents that some processors stay idle for a long time, while others are overloaded.

Duplex. The Duplex heuristic is a combination of the Min-min and Max-min heuristics. The Duplex heuristic performs both of the Min-min and Max-min heuristics and then uses the better solution [5][6][3]. Duplex exploits the conditions in which either Min-min or Max-min performs better.

Genetic Algorithms (GA). Genetic Algorithms are a technique used for searching large solution spaces. Multiple possible mappings of the metatask are computed, which are considered chromosomes in the population. Each chromosome has a fitness value, which is the result of an objective function designed in accordance with the performance criteria of the problem (for example maxpan). At each iteration, all of the

chromosomes in the population are evaluated based on their fitness value, and only the best of them survive in the next population, where new allocations are generated based on crossover and mutation operators. The algorithm usually stops when a predefined number of steps is performed, or all chromosomes converge to the same mapping.

Simulated Annealing (SA). Simulated Annealing (SA) is an iterative technique that considers only one possible solution (mapping) for each metatask at a time. This solution uses the same representation as the chromosome for the GA. SA uses a procedure that probabilistically allows poorer solutions to be accepted to attempt to obtain a better search of the solution space. This probability is based on a system temperature that decreases for each iteration. As the system temperature decreases, poorer solutions are less likely to be accepted. The initial temperature of the system is the maxspan of the initial mapping, which is randomly determined. At each iteration, the mapping is transformed in the same manner as the GA, and the new maxspan is evaluated. If the new maxspan is better (lower), the new mapping replaces the old one. If the new maxspan is larger, a uniform random number z in the $[0, 1)$ interval is generated. Then, z is compared with y, which is determined according to the following formula: $y = \frac{1}{1+\exp \frac{OLDmaxspan - NEWmaxspan}{Temperature}}$

If $z > y$ the new and poorer mapping is accepted, otherwise it is rejected, and the old mapping is preserved. The following cases may occur:

- $OLDmaxspan \approx NEWmaxspan$ or the system temperature is very large $\Rightarrow y = 0.5$ and poorer solutions are accepted with a high probability
- $OLDmaxspan$ very different from $NEWmaxspan$ or the system temperature is very small $\Rightarrow y = 1$ and poorer solutions are rejected with a high probability.

After each mutation, the system temperature is reduced by the cooling rate (usually considered 900 of its current value) and one iteration of SA is completed. Stopping conditions: when there is no change in the maxspan for a number of generations (for example 150 iterations) or, the system temperature approaches zero.

A*. A* heuristic is a search technique based on a μ-ary tree, which has been applied in various task allocation problems. The A* heuristic begins at a root node that is a null solution. As the tree grows, nodes represent partial mappings (a subset of tasks is assigned to machines). With each child added, a new task t is mapped. Each parent node generates μ children, where μ is the number of possible mappings for task t. After a parent node has done this, the parent node becomes inactive. A pruning process is performed to limit the maximum number of active nodes in the tree at any one time. Moreover, each node, n, has a cost function, $f(n)$, associated with it. $f(n)$ represents the maxspan of the partial solution of node n plus a lower-bound estimate of the time to execute the rest of the (unmapped) tasks in the metatask. After the root node generates μ nodes for the first task, t_0 (which means the mapping of the task on the available machines), the node with the minimum $f(n)$ generates its μ children, and so on until a predefined number of nodes are created. From that point on, any time a node is added, the node with the largest $f(n)$ is deactivated, pruning the tree. This process continues until a complete mapping is reached.

3. Near-Optimal Scheduling Models

3.1. Random and Best of n Random

The task of scheduling an application is a complex one often taking a significant amount of time. When the complexity of scheduling the application becomes sufficiently large it may be the case that the scheduling phase will be responsible for a major proportion of the total application duration. In extreme cases it is possible that the execution time saved by selecting the optimal schedule will be less than the increase in time required to find that schedule.

In order to avoid this problem we consider a random scheduling algorithm which simply chooses a resource at random for each component in the application. Other than checking that each component is capable of executing on its chosen resource, no form of active selection takes place. Producing a schedule this way is very fast and will not produce an excessive delay in selecting a schedule. An optimisation of this algorithm is also presented. The Best of n Random scheduler randomly generates n schedules and then returns the one with the highest benefit value.

This method maintains the speed of the random scheduler while increasing the chance of selecting a high quality schedule.

3.2. Exhasutive Search

The exhaustive search algorithm presented here servers to discover both the optimal solution to the scheduling problem, and to define an upper bound on the computation time taken to select a schedule. The exhaustive search performs a depth first search on the tree of all possible schedulers, calculating the benefit of

each feasible schedule. It then returns the optimal result. Any scheduler which takes longer to return then the exhaustive search can be improved upon by replacing it with an exhaustive search, and so does not need to be considered.

3.3. Simulated annealing

Simulated annealing, as we presented bellow, is a generalization of the Monte Carlo method, used for optimisation of multi-variable problems. Possible solutions are generated randomly and then accepted or discarded based on the difference in their benefit compared to a currently selected solution. Simulated annealing has been used previously in order to select how many resources a Grid application should be split over, but not to select which resources are used. If each resource is managed by one agant, this method can be consider a decentralized one.

Algorithm 1 Simulated Annealing Algorithm

```
1: currentSolution ← generateNewSolution()
2: currentBenefit ← getBenefit(currentSolution)
3: while noAcceptedSolutions > 0 do
4:    noAcceptedSolutions ← 0
5:    for i = 0 to maxNoOfTrialSolutions do
6:       trailSolution ← generateTrialSolution()
7:       trialBenefit ← getBenefit(trailSolution)
8:       if acceptTrialSolution() then
9:          currentSolution ← trailSolutionl
10:         currentBenefit ← trialBenefitt
11:         noAcceptedSolutions++
12:         if noAcceptedSolutions ≥ maxAcceptedSolutions then
13:            break out of for loop
14:         end if
15:      end if
16:   end for
17:   reduce T
18: end while
```

An initial schedule is generated at random and its benefit calculated. A new schedule, a permutation of the previous solution, generated by moving one component onto a different resource is then created. The new schedule is either accepted or discarded as the new solution through the Metropolis algorithm. If the new solution has greater benefit value than the current solution it is accepted as the new selection. However, if the new solution has a lower benefit then it is accepted with a probability proportional to $e^{-d\beta/T}$, where $d\beta$ is the difference in benefit value between the two solutions, and T is a control parameter.

3.4. Game Theory

Grid scheduling can also be modelled using game theory, a technique commonly used to solve economic problems. In game theory a number of players each attempt to optimise their own payoff by selecting one of many strategies. This is a dedicated strategies for a decentralized system.

To apply this to scheduling each component is modelled as a player in the set $P = p_1, p_2, ..., p_n$ and each resource is a strategy in the set $S = s_1, s_2, ..., s_m$. There exists a set of strategies $S_i \subset S \forall p_i \in P$ which describes the strategies available to the i^{th} player (the resources that component is capable of executing on). We then define the payoff of a given strategy s_j for a player p_i, given all other players have chosen a strategy, to be the sum of the benefit function applied to both the i^{th} component executing on the j^{th} resource, and the communications being sent to the ith component. Hence for player p_i the payoff is dependent on s_j and the stratagies chosen by P_i' , the set of players preceeding p_i.

Each player is unable to influence the strategy selected by other players. Thus this is the branch of game theory classed as non-cooperative.

Algorithm 2 Game Theory Algorithm

```
1: for all candidateStrategies in StrategyList do
2:    strictlyDominated ← true
3:    for all alternateStrategy in StrategyList do
4:       if candidateStrategy is not dominated by alternativeStrategy then
5:          strictlyDominated ← false
6:       end if
7:    end for
8:    if strictlyDominated then
9:       remove candidateStrategy from StrategyList
10:   end if
11: end for
```

Game theory examines the existence of a stable solution in which players are unable to improve their payoff by changing only their strategy. The game used here is a static game of complete information. This is a class of game in which the payoff for all players is known for all combinations of strategies and each player has to make one single choice of strategy. The game is solved using elimination of strictly dominated strategies, in which the least optimal solutions are continually discarded. All players simultaneously look at their list of potential strategies and for each strategy attempts to determine if it is strictly dominated. Player p_i can say that strategy s_j is strictly dominated by s_k if for all

possible strategies available to the players in $P_i^{'}$, the payoff for choosing s_j is less than that for s_k. Once a strategy has been identified as strictly dominated it is guaranteed not to be an optimal solution, so it can be removed from all further consideration.

This scheduler never considers the benefit to the total application, instead only focusing on the optimisation of individual components. This may lead to suboptimal schedules for the entire application, just as an optimal schedule for the application may lead to a suboptimal placement for an individual component.

3.5. Ad-Hoc Greedy Scheduler

The scheduling algorithm based on Ad-Hoc greedy technique in conjunction with a hand-crafted Performance Model, select the machines on which to schedule the execution. The list of all qualified, currently available machines is obtained from the Grid Middleware (Globus Metacomputing Directory Service - MDS); it may contain machines from several geographically distributed clusters. The Monitoring System is contacted to obtain details pertaining to each machine (i.e., the CPU load, the available memory) and the latency and bandwidth between machines. This detailed resource information is used by the Performance Model to estimate the execution time. The Ad-Hoc scheduling algorithm can be approximated as in Algorithm 3.

Algorithm 3 Ad-Hoc Greedy Algorithm

```
1: for each cluster, starting a new search in the cluster do
2:    select fastest machine in the cluster to initialize
3:    repeat
4:       find a new machine which has highest average
         bandwidth to the machines that are already
         selected and add it to the selected machines
5:       ensure that memory constraints are met by all
         machines (details omitted here)
6:       use the Performance Model with detailed machine
         and network information to predict the
         execution time
7:    until the Performance Model shows that execution
      time is no longer decreasing
8:    track the best solution
9: end for
```

In this algorithm, new machines are ordered by their average bandwidth with the machines that have already been selected and they are added to the selection in this order. This technique returns a good set of machines for the application, but it assumes that communication is the major factor determining the execution time of the algorithm. Other greedy techniques using different orderings have been implemented within the resource selection process, for example, using CPU load to order the machines.

3.6. Genetic Algorithm for Decentralized Scheduling

In [7] a genetic algorithm was proposed. The description of the scheduling algorithm in a logical flow of activities is described below (algorithm 4).

Fitness function. The fitness function is essential for the evaluation of chromosomes. It represents a numerical value which measures the quality of each individual in the population. With scheduling objectives in mind, fitness functions have been developed in order to achieve algorithm convergence on a near optimal solution.

The main objective is to obtain a well-balanced assignment of tasks on processors, that implies low overall completion time on processors. Many researchers have oriented their study towards genetic schedulers with optimal execution times. This means minimization of maxspan, which is defined as: $t_M = \max_{1 \leq i \leq n}\{t_i\}$, where n is the number of processors and t_i is the total execution time for processor i, computed as the sum of processing times for all tasks assigned to this processor in the current or previous schedules: $t_i = t_i^p + t_i^c = \sum_{j=1}^{T_i}(t_{i,j})$. If we have considered T_i to be the total number of tasks assigned to processor i for execution and $t_{i,j}$ the running time of task j on processor i. t_i^p is the execution time for previously assigned tasks to processor i, and t_i^c is the processing time for currently assigned tasks.

A mapping of maxspan in the [0, 1] interval leads to the factor $\frac{1}{t_M}$ that is often considered for fitness computing. To optimized this factor for a more efficient search of well-balanced schedules is neccesary to reduce the difference between the minimum and the maximum processing times is a factor worth considering for fitness, in terms of load-balancing optimization. Therefore, one factor introduced for fitness computation is:

$$f_1 = \frac{t_m}{t_M} = \frac{\min_{1 \leq i \leq n}\{t_i\}}{\max_{1 \leq j \leq n}\{t_j\}}, 0 \leq f_1 \leq 1$$

The factor converges to 1 when t_m approaches t_M, and the schedule is perfectly balanced.

The second factor considered fitness computation is the average utilization of processors:

$$f_2 = \frac{1}{n}\sum_{i=1}^{n}\frac{t_i}{t_M}, 0 \leq f_2 \leq 1$$

Algorithm 4 Genetic Algorithm

1: A user requests that one or more tasks are scheduled. His request has as a parameter the name of the file containing a description of the tasks. The file has a standard XML format and presents requirements for each task relative to memory, cpu usage, execution time, etc.
2: The input file is processed and a "batch of tasks" (group of tasks) object is constructed.
3: The batch of tasks is broadcast to all the nodes in the cluster.
4: The nodes receive the group of tasks to be scheduled. The tasks are inserted sorted in a queue according to a sorting criteria like arriving time or scheduling priority. If the number of tasks in the queue is less than a predefined length of the chromosome, they wait for T units of time before starting the genetic algorithm. If the chromosome is still not complete at the end of the waiting period, a non-influential padding is added. On the contrary, if the length of an arriving group of tasks exceeds the predefined dimension of the chromosome, some tasks are saved in the waiting queue and will be scheduled at the next time.
5: On each node, a daemon keeps an up-to-date status of the computers in the Grid on which tasks are sent for execution, by constantly interrogating a monitoring system. The Grid is a dynamic environment in which nodes' characteristics such as free memory or cpu utilization vary over time. The current configuration information is necessary for an accurate scheduling process. The nodes interrogates the daemon for at the beginning of the algorithm, to find out the current status of the Grid.
6: The nodes in the cluster run the GA. Each node starts with a different, specific initialization of the genetic algorithm. The subsequent steps of the GA are similar for all the nodes in the cluster, and so is the fitness formula. In this way, the clients will compute different optima from which the best one will be chosen.
7: The migration of the best current solutions is performed after each step of the GA, thus ensuring that the population finds a better optima. The nodes exchange the fittest individuals and insert them in the next generation.
8: The generation of populations stops after a finite, predefined number of steps. At this point, each client in the cluster computes its optimal individual.
9: The same communication procedure as above is used for the final step of the GA. Each node sends its optimum to all the other nodes in the cluster and the final optimal individual is decided by each one. The fittest chromosome is selected from the optimal individuals only. The result is the same on every node, because the computing procedure and the individuals at the last step of the GA are the same.
10: The scheduling obtained is saved in a history file

Zomaya pointed out its utility of reducing idle times by keeping processors busy. Division by maxspan is pursued in order to map the fitness values to the interval $[0, 1]$. In the ideal case, the total execution times on the processors are equal and equal to maxspan, which leads to a value of 1 for average processor utilization.

Another considered factor represents meeting the imposed restrictions. In a realistic scenario, task scheduling must meet both deadline and resource limitations (in terms of memory, cpu power). In deadline computation for a task t, we must consider the execution times for each of the tasks assigned to run before task t on the same processor. These tasks occupy a previous slot on the respective processor in our encoding of a chromosome. The fitness factor is subsequently defined as:

$$f_3 = \frac{T_s}{T}, 0 \leq f_3 \leq 1$$

where T_s denotes the number of tasks which satisfy deadline and computation resource requirements, and T represents the total number of tasks in the current schedule. This factor acts like a contract penalty on the fitness. Its value varies reaching 1 when all the requirements are satisfied and proportionally decreases with each requirement that is not met. The chance of being selected for future generations is reduced due to the penalty introduced by this factor. Still, the schedule is not dismissed, but may be used in subsequent reproduction stages that lead to valid chromosomes.

The fitness function applied consists of the contribution of the factors presented ($0 \leq F \leq 1$):

$$F = f_1 \times f_2 \times f_3 = \left(\frac{t_m}{t_M}\right) \times \left(\frac{1}{n}\sum_{i=1}^{n}\frac{t_i}{t_M}\right) \times \left(\frac{T_s}{T}\right)$$

3.7. Experimental results

We carry out our experiments with complex scheduling scenarios and with heterogeneous input tasks and computation resources.We improve upon centralized genetic approaches with respect to scalability and robustness.

We compared the performances of the presented optimization methods Figures 1, 2 and 3 show the execution time for scheduling algorithm for different number of thasks.

The Ad-Hoc scheduler has very little overhead associated with it. The number of times it has to evaluate the Performance Model for different machine configurations is at most (number of clusters x total number of available machines). This takes a negligible amount of time given the size of the current testbeds.

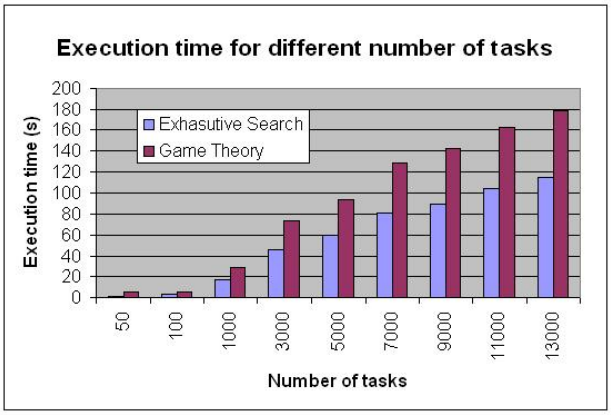

Figure 1. Compariosn between Exhausitve Search and Game Theory for different number of tasks

The Simulated Annealing scheduler is has a higher overhead; however, it is still relatively small in comparison to the time required for most of the problems that would solved in a Grid environment. The time taken by simulated annealing is dependent on some of algorithm parameters, like the temperature reduction schedule and the desired number of iterations at each temperature, as well as the size of the space that is being searched. In our experiments, when the testbed consists of 11 machines, we got the maximum overhead of 9 seconds from the Simulated Annealing scheduling process for 13000.

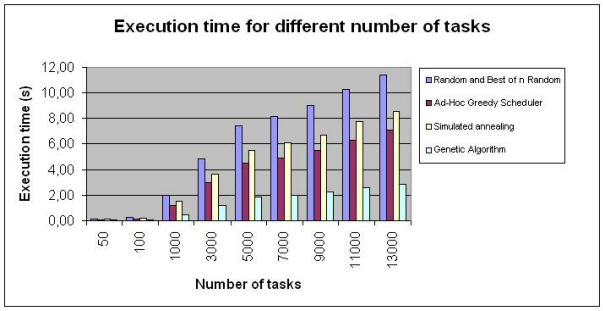

Figure 2. Compariosn between Random and Best of n Random, Ad-hoc Greedy, Simulating Annealing and Genetinc Algorithm for different number of tasks

We also compared the performances of the Decentralized Cooperative Genetic Algorithm with three other strategies: OLB, Centralized GA and Decentralized Non-cooperative GA. It is shown that the algorithm clearly outperforms these methods. Decentralization and cooperation provide significantly better results of load-balancing and average processor utilization increase, as well as of total execution time minimization.

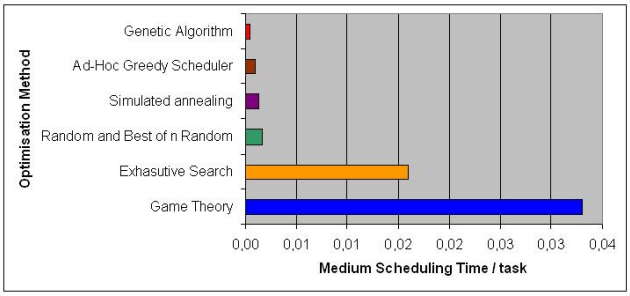

Figure 3. Medium scheduling time per task for different scheduling methods

3.8. Conclusions

If we analtze all presented results, we can conclude that the "best" algorithm for decentralized scheduling is recomanded to be based on genetic algorithm.

References

[1] T. Casavant, and J. Kuhl, A Taxonomy of Scheduling in General-purpose Distributed Computing Systems, in IEEE Trans. on Software Engineering Vol. 14, No.2, pp. 141-154, February 1988.Scheduling Algorithms for Grid Computing

[2] R. Armstrong, D. Hensgen, and T. Kidd, The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions, in 7th IEEE Heterogeneous Computing Workshop (HCW '98),' pp. 79-87, 1998.

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

[4] R. F. Freund, H. J. Siegel, Heterogeneous processing, IEEE Comput. 26, 6 (June 1993), pp. 13-17.

[5] I. Foster and C. Kesselman, Globus: A Metacomputing Infrastructure Toolkit, International Jour-

nal of Supercomputer Applications, 11(2): 115-128, 1997.

[6] Weizhao, K., Ramamritham, K. and Stankovic, J. A., Preemptive Scheduling Under Time and Resource Constraints, IEEE Transactions on Computers, Vol. 36, No. 8, 1987, pp. 949-960.

[7] George V. Iordache, Marcela S. Boboila, Florin Pop, Corina Stratan, Valentin Cristea - A Decentralized Strategy for Genetic Scheduling in Heterogeneous Environments, GADA 2006 Montpellier, France, Nov 2 - Nov 3, 2006