Asociación Argentina de Mecánica Computacional

Mecánica Computacional Vol XXIX, págs. 9563-9571 (artículo completo)
Eduardo Dvorkin, Marcela Goldschmit, Mario Storti (Eds.)
Buenos Aires, Argentina, 15-18 Noviembre 2010

A PARALLEL GREEDY RANDOMIZED ADAPTIVE SEARCH PROCEDURE METAHEURISTIC FOR THE RESOURCE-CONSTRAINED TASK SCHEDULING PROBLEM

Edelberto F. Silva, Bruno J. Dembogurski, Gustavo S. Semaan

Universidade Federal Fluminense, Rua Passo da Pátria 156 - Bloco E - 3o. andar São Domingos Niterói - RJ, {esilva, bdembogurski, gsemaan}@ic.uff.br, http://www.ic.uff.br

Keywords: Task Scheduling Problem, Multi-core, Metaheuristics, Parallel.

Abstract. The Task Scheduling Problem (TSP) consists, basically, of a group of tasks that should be executed respecting precedence constraints and minimizing the execution time of all tasks. For every task, processing units must be chosen to execute a particular task and at what moment that will happen. This work presents a parallel implementation, through multi-core architectures, to solve Resource-Constrained TSP, based on a known model called dynamic resource constrained task scheduling problem (DRCTSP) and using a Greedy Randomized Adaptive Search Procedure (GRASP) metaheuristic. In this approach, the moment a task is activated until the last considered time period, an amount of resources (called profit), associated with each activated task, is provided for each period. Thus, the amount of resources available in a given period will depend on what tasks have been activated by then and when this occurred. Based on the OpenMP standard, this work shows the benefits of using parallel methods to speed-up the overall time spent to find a feasible solution without compromising the its quality. Also, a comparative analysis is done based on the empirical probability distributions of the random variable time to target solutions.

1 INTRODUCTION

The Task Scheduling Problem (TSP) belongs to the NP-Complete class of combinatorial optimization problems, meaning that finding a solution using exhaustive methods can be inefficient. Metaheuristics, on the other hand, presents an interesting problem-resolution paradigm that is more suitable to this kind of problem. So, in this work the approach used was a Greedy Randomized Adaptive Search Procedure (GRASP), is a multi-start metaheuristic composed basically by two phases: Construction and Local Search Resende and Ribeiro (2003). The construction phase aiming to build a feasible solution, while the local search going to investigate the neighborhood, searching a way to increasing the solutions quality, and keeping the Best solution as the result. The Algorithm 1 shows the GRASP implementation.

This work is based on the Dynamic Resource-Constrained Task Scheduling Problem (DRCTSP) presented in Renato et al. (2006) which consists of a DAG (directed acyclic graph) $G = (V, A)$, where V is the set of vertexes (tasks) and A is the set of arcs (precedence among the tasks). Associated to each task t_i there is a cost c_i and a profit l_i (positive integer values). There is too a planning process (time interval composed by H time units). The objective of the DRCTSP is maximizing the available resources at the end of the planning process. This model has potential application on manufacture expansion projects, where the tasks are expansion steps that can be made separately Renato et al. (2006). Further detailment can be found in Renato et al. (2006) and the necessary mathematical basis will be presented in the following sections.

Nowadays, with the increasing number of cores available in modern processors, is natural the interest on researching and developing parallel algorithms that take advantage of these resources. With this objective in mind, this work implements the RCTS problem Freitas (2002) using the OpenMP standard in a multi-core approach. The section regarding this work approach will show a really straight forward parallel GRASP implementation, which showed good results considering its simplicity.

A traditional GRASP pseudo-code can be seen in Algorithm 1. Basically, the construction phase will generate an initial guess, a seed, to the algorithm. After, it will proceed into a local search to improve the actual solution. That might be limited by a maximum number of iterations or a time limit, or maybe both, depending of the problem being solved.

```
**Procedure** $GRASP(Max\_Iterations, Seed)$;
**for** $k = 1$ *to* $Max\_Iterations$ **do**
    $Solution = Greedy\_Randomized\_Construction(Seed)$;
    $Solution = Local\_Search(Solution)$;
    $Update\_Solution(Solution, Best\_Solution)$;
**end**
**return** $Best\_Solution$;
**End GRASP()**;
```

Algorithm 1: Pseudo-code of a GRASP procedure.

The GRASP implementation in this work is based in Freitas (2002); Festa and Resende (2002); Holland (1992); Renato et al. (2006). As a greed randomized construction it utilizes a randomized ADD heuristic. Its local searches are presented in three different forms: The first one will remove jobs that don't return any profit at all. The second one will remove sub-branches of jobs that don't return profit. The third local search, does not remove any jobs from the solution, in Freitas (2002) different local searches are presented. Actually, this last approach tries to guess their initial time. For instance, if a job i is (with associated profit of 3) can be activated in some instant before its actual activation time, it will return a liquid profit higher (in this case it will be 3 plus each time unit anticipated).

2 INTERGER MATHEMATICAL FORMULATION

Given a directed graph $G = (V, A)$, each vertex i of V represents a task to be scheduled and is associated with a positive cost c_i and a profit l_i. The edges (i, j) indicate that the vertices i are the predecessors of task j, and may be represented by $P(j)$. The time interval in which the tasks must be scheduled is represented in T units of time. The goal is to maximize the amount of resources available through the scheduling of tasks. For each scheduled task i, it is necessary to pay a cost c_i associated with it. To achieve this, the task cost is removed from the amount of resources currently available. With the activation of task i, the available resources are increased by l_i units at each subsequent time unit.

In the formulation presented below, the binary variable x receives the value 1 if a task i is activated at time t, otherwise it will recieve zero. The variables Q_t and L_t respectively indicate the amount of resources available at the beginning instant of time t and the profit that will be added to the resources in the next time instant $(t + 1)$.

Maximize

$$Q_{T+1}$$

Subject to

$$\begin{aligned}
&|P(i)|x_{i1} = 0 \forall i = 1, .., n \\
&|P(i)|x_{i1} \leq \sum_{j \in P(i)} \sum_{t=1}^{t-1} x_{jt} \forall i = 1, .., n \forall t = 2, .., T \\
&\sum_{i=1}^{n} c_i x_{it} \leq Q_t \forall t = 1, .., T \\
&Q_{T+1} = Q_T - \sum_{i=1}^{n} c_i x_{it} + L_t \forall t = 1, .., T \\
&L_t = L_{t-1} + \sum_{i=1}^{n} l_i x_{it} \forall t = 1, .., T \\
&\sum_{t=1}^{T} x_{it} \leq 1 \forall i = 1, .., n \\
&x_{it} \in \{0, 1\} \forall i = 1, .., n \forall t = 1, .., T \\
&Q_t, L_t \in \forall t = 1, .., T + 1
\end{aligned}$$

3 OPENMP STANDARD

OpenMP (Open Multi-Processing) is an application programming interface (API)(Chandra et al. (2001); Quinn (2003); Rabbani et al. (2007); Remy (2004))that is a valuable tool to create and design parallel applications. With easy and simple preprocessor directives it is possible to spread the workload among the processors. The standard is an implementation of multithreading, where the main thread divides the work among slave threads.

All threads run concurrently and the runtime environment allocates them to each processor. The main reasons to use this standard are: portability and simple implementations. It supports multiplatform shared memory multiprocessing in C, C++ and Fortran, on many architectures. To create a parallel program, the section of the code that is meant to run in parallel is marked with a directive that will ensure the threads creation before that section is actually executed. Each thread executes the parallelized section of code independently. Work sharing is also possible, by using appropriate directives. This way, both task parallelism and data parallelism can be achieved using OpenMP.

4 PARALLEL APPROACH

Our approach is entirely based on the OpenMP standard. It is also possible to extend or combine it with different methods and/or standards, but that will be presented in the conclusion section.

The main idea is to create many instances of the code using OpenMP in order to have a bigger pool of solutions. Since every execution starts with a different seed, the more instances, the higher is the chance to find a good solution. In other words, the whole metaheuristic implementation is based in the possibility of splitting the main loop iterations throughout the available cores and collecting all the results after the processing.

To create a parallel version of the GRASP algorithm, this work focused in a simple approach that does not deal with data dependency. In order to achieve faster and better results, four instances of the algorithm are executed in parallel, one in each available core. Also, it is possible to extend this approach to any number of cores and also apply some other parallelization methods (such as MPI).

A pseudo-code of the GRASP approach is presented in the Algorithm 2, also the OpenMP directives necessary to create each thread.

```
Procedure $GRASP(Max\_Iterations, Seed)$;
//generate threads
#pragma omp parallel
.
//initializations
.
//splitting the for
#pragma omp for
for $k = 1$ to $Max\_Iterations$ do
    $Solution = Greedy\_Randomized\_Construction(Seed)$;
    $Solution = Local\_Search(Solution)$;
    $Update\_Solution(Solution, Best\_Solution)$;
end
return $Best\_Solution$;
End GRASP();
```

Algorithm 2: pseudo-code of an OpenMP version of a GRASP procedure.

Algorithm 2 shows how simple and straightforward is an OpenMP implementation. The directive *"#pragma omp for"* defines that the next *"for"* command will be split into different threads, one for each loop iteration. If necessary, it is possible with simple directives, to force an order, in which, the loop is being executed and/or verify in what thread each iteration is being executed.

In this algorithm each thread will run an instance of the code which means that each thread will run one greedy construction, one local search and will update the solution. The idea is to obtain a variety results from a single execution, with more instances of the same code but each with its own seed. This will increase the chance to find a feasible solution.

5 TESTS AND RESULTS

All tests were executed on an AMD Phenon 9850 x4 with 4GB RAM memory; also each and every execution had an upper bound for its fitness function and/or execution time limit to find a possible solution.

The first test, represented in Figure 1, shows that the parallel implementation has little impact in the fitness result, meaning that, the gap between the parallel and sequential implementations is acceptable and does not compromise the final result.

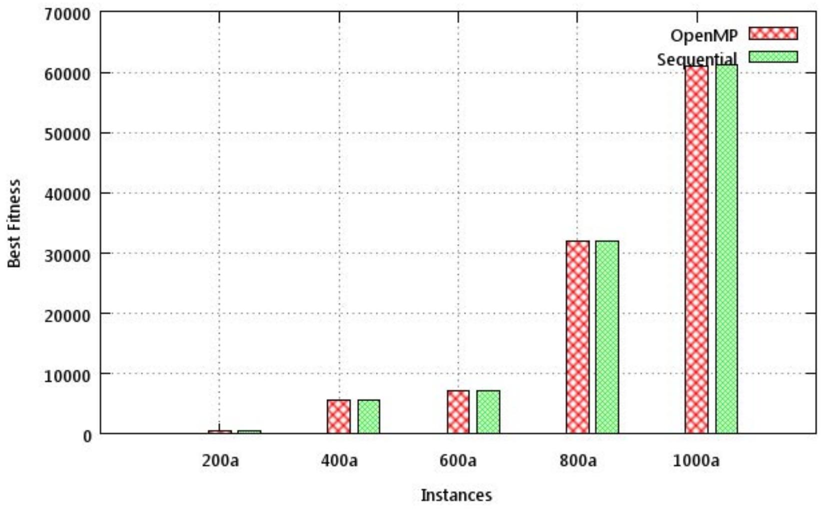

Figure 1: Fitness comparison: OpenMP and Sequential versions

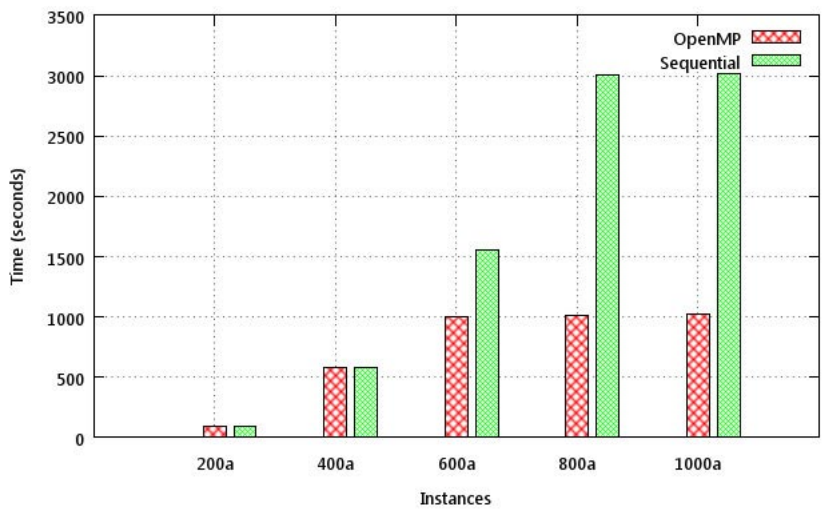

Figure 2: Time comparison: OpenMP and Sequential versions

In Figure 2 it is possible see the importance of a parallel approach to the TSP problem. The computational time required to process instances, especially those with a higher dependency between tasks (600a, 800a and 1000a), is, in some cases, proportional to the number of cores available. In this particular figure, an example would be the 1000a instance, which represents around 90% of dependency between tasks; the processing time is reduced to one fourth of the sequential implementation.

In order to obtain more detailed results about the use of the several cores quantities, the algorithms were submitted to a new experiment, the empirical probability distributions of the random variable time to target solutions Aiex et al. (2006). In this way, the best fitness solutions values were used as targets and each algorithm was run one hundred times over the selected instances for each core quantity. This experiment presents results obtained executing three instances and considering four colors.

As we can see in Figures 3, 4 and 5, parallel algorithms have a higher probability to achieve a desirable fitness target. That is explained by the fact that each core has its own instance of the implementation, with a different initial seed, which can be proven really efficient to this scenario. To make it more clear once a core finds a desirable fitness value (or an optimal value) all execution is halted.

This paper presents the results of experiments using the instances 600, 800 and 1000, that have a higher number of tasks and dependency between them. Figures 3, 4 and 5 shows that the use of a multi-core approach lowers the processing time to achieve the targets. For the instance 600, for example, the probability of versions using one, two, three and four cores reach the target at the 200 seconds were 28%, 46%, 68% and 98%, respectively. This way, observing the results, the use of idle cores through parallel metaheuristics can be an interesting way to solve this and other optimizations problems without both use other computers or compromising the final result.

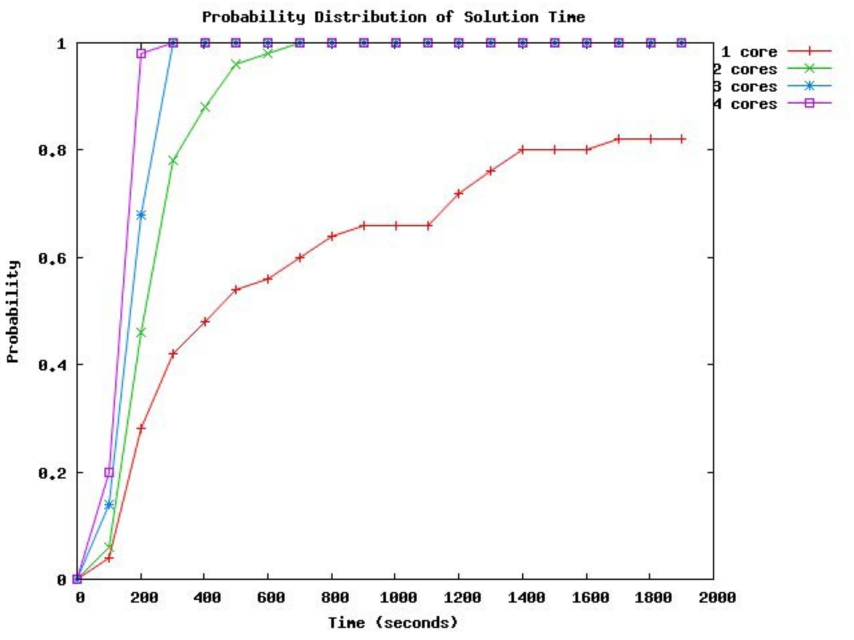

Figure 3: TTTPlot: instance 600

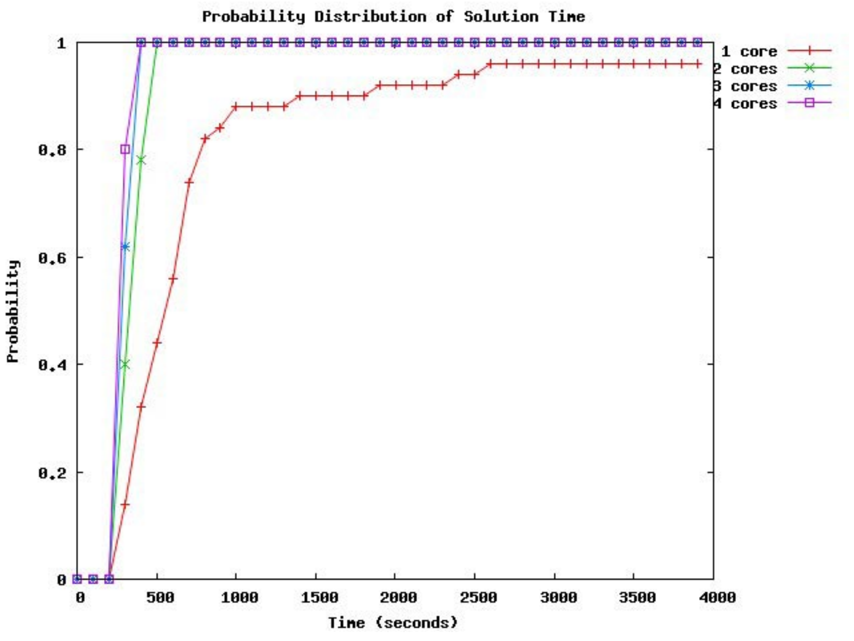

Figure 4: TTTPlot: instance 800

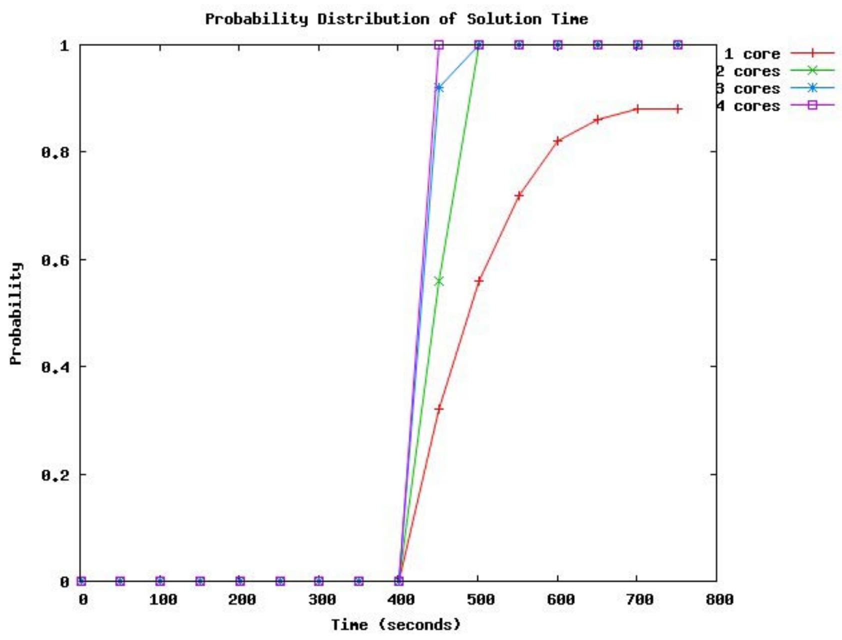

Figure 5: TTTPlot: instance 1000

6 CONCLUSIONS AND FUTURE STEPS

Computational results showed that the proposed parallel algorithm is an efficient way to solve the Resource-Constrained TSP as showed in the Results section. Also, it shows that with simple parallelization methods it is possible to achieve a considerable speed up regarding the time to find a good solution.

The empirical probability distributions of the random variable time to target solutions proved to be a powerful analysis in this work. It is easier to see that a parallel approach has a higher chance to achieve target fitness values and validate such implementations.

The main inspiration to this work is the crescent number of cores in recent processors. This tendency will only grow. To take good advantage of such architectures is becoming a must in this kind of problem and the graphs presented in the results section indicate that the use of available cores, that usually stay idle during the processing, can increase the algorithm efficiency in finding good solutions in less time.

Based on the experiments and searches, this paper proposes as future work new ways that can help to solve the problem, such as:

- The use of different parallel approaches can be really useful, like MPI protocol. With this in mind, it is possible to expand this approach to a higher level, using multiple machines with multiple cores, each executing instances of the same application.

- To parallelize different parts of the metaheuristic, creating a speed-up of a different kind, this time making the metaheuristic to actually run faster. The main issue is to avoid classic problems, such as: data dependency and bottlenecks.

- To use Integer programming Formulation aiming to analyze and compare the results.

- To developer and analyze the use of other metaheuristics, such as Iterated Local Search (ILS), Variable Neighborhood Search (VNS) or a hybrid heuristic version Resende and Ribeiro (2003).

REFERENCES

Aiex R.M., Resende M.G.C., Celso, and Ribeiro C. Tttplots: A perl program to create time-to-target plots. *Optimization Letters*, 1:10–1007, 2006.

Chandra R., Dagum L., Kohr D., Maydan D., McDonald J., and Menon R. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. ISBN 1-55860-671-8.

Festa P. and Resende M. GRASP: An annotated bibliography. In C. Ribeiro and P. Hansen, editors, *Essays and surveys in metaheuristics*, pages 325–367. Kluwer Academic Publishers, 2002.

Freitas A.A. Evolutionary computation. In *In Handbook of Data Mining and Knowledge Discovery*. University Press, 2002.

Holland J.H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0262082136.

Quinn M.J. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003. ISBN 0071232656.

Rabbani M., Fatemi Ghomi S., Jolai F., and Lahiji N. A new heuristic for resource-constrained project scheduling in stochastic networks using critical chain concept. *European Journal of Operational Research*, 176(2):794–808, 2007.

Remy J. Resource constrained scheduling on multiple machines. *Inf. Process. Lett.*, 91(4):177–182, 2004. ISSN 0020-0190. doi:http://dx.doi.org/10.1016/j.ipl.2004.04.009.

Renato A., Silva V., and Ochi L.S. A dynamic resource constrained task scheduling problem. In *CLAIO:Congreso Latino-Iberoamericano de Investigación Operativa*. 2006.

Resende M. and Ribeiro C. Greedy randomized adaptive search procedures. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 219–249. Kluwer Academic Publishers, 2003.