A Parallel Approach to Resource-Constrained Task Scheduling Problem
Bruno J Dembogurski, Edelberto F Silva, Gustavo S Semaan
Universidade Federal Fluminense - UFF, Niterói, Rio de Janeiro, Brazil
{bdembogurski, esilva, gsemaan}@ic.uff.br

Abstract
The task scheduling problem (TSP or SP) is a well known topic. Basically it consists of a group of tasks that should be executed (activated, started) somehow, spending the least time possible. The classic problem consider, besides tasks, the precedence and the processing units (PU) in which those tasks will be executed. These units might have similar characteristics (homogeneous environment) or not (heterogeneous). For every task, a PU must be chosen to execute a particular task and at what moment that will happen.

This work presents a parallel implementation to solve resource-constrained task scheduling problems. This approach is based on a well known technique where from the moment when a task is activated until the last considered period, a quantity of resources called profit (associated to the activated task) is available at each period. Thus, the quantity of available resources, at a given period, will depend on what tasks were activated until this period and when it happened. Some heuristics were used to handle this problem including concepts of GRASP and Evolutionary Algorithms.

Based on the OpenMP and Message Passing Interface (MPI) patterns, this work showed the benefits of using parallel methods to speed-up the overall time spent to find a solution without compromising the its quality. Also, a comparative analysis was done, between a standalone OpenMP application, a standalone MPI application and a hybrid implementation using MPI and OpenMP.

Several experiments analyses were done with a variety of cases. These consist in a different amount of tasks and different levels of dependency between them. Tests showed satisfactory results, which, in some cases, the total speed-up obtained is proportional to the number of cores and/or machines (network nodes or cluster nodes) available. This way, the computational results showed that the proposed approach is an efficient way to solve the TSP.

Keywords: Parallel Programming, metaheuristic, multicore, optimization.

1. Introduction
This work focuses the use of parallel architectures in order to solve optimization problems, specifically the Dynamic Resource-Constrained Task Scheduling Problem (DRCTS). For this matter, the study of patterns such as OpenMP and MPI (Message Passing Interface) are needed in order to explore the parallel potential of modern architectures.

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 RCTSP problem [1] with two parallel approaches: a standalone OpenMP version and hybrid OpenMP/MPI version.

The solving of the RCTS problem is done by two main metaheuristics. One version is solved by a Greedy Randomized Adaptive Search Procedures (GRASP) approach [2, 4] and a second version is solved by Evolutive Algorithm (EA) [1, 2], a detailed explanation will be presented in the following sections. Both algorithms are suited for parallel implementations since they allow splitting of the iterations throughout the cores with minimum information exchange needed. Also, the execution of independent instances, based on the number of machines available in the network, make easier to find better results since each instance will have its own seed.

This work is based on the implementations presented in [9] and the use of these two metaheuristics is justified by their own characteristic, where GRASP is a restart type and EAs are population heuristics.

2. Metaheuristic Algorithms
This section presents the proposed metaheuristic, Evolutionary Algorithm (EA) and Greedy Randomized Adaptive Search Procedures (GRASP), aiming to solve the RCTSP.
According to [1, 2], the EA paradigm consists of stochastic search algorithms that are based on biological theories of evolution. In this way, concepts of survival of the best fitted individual and hereditary transference of genetic characteristics are used, like reproduction, mutation and selection. The basic idea is that each individual represents a solution to a given problem. The Genetic Algorithms are a particular class of evolutionary algorithms, and this term was first used by John Holland [3]. The Figure 1 shows a simple template for the operation of a genetic algorithm present by [2].

```plaintext
Procedure GA();
    Build_Initial_Population();
    while termination condition not satisfied do
        repeat
            Crossover();
            Mutation();
            Evaluate_Offsprings();
        until sufficient offspring created;
        Select(newpopulation);
    end
End GA();
```

**Algorithm 1:** a Genetic Algorithms (GA) template.

The GRASP is a multi-start metaheuristic composed basically by two phases: construction and local search [2, 4]. 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 Figure 2 shows the main block of GRASP procedure.

```plaintext
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 2:** Pseudo-code of a GRASP procedure.

3. Parallel Architectures

It is possible to name two main approaches to parallel implementations. One utilizes distributed memory architecture what refers to a multiple-processor computer system in which each processor has its own private memory. Computational tasks can only operate on local data, and if remote data is required, the computational task must communicate with one or more remote processors. In this context MPI is the most used model and will be explained in the following section.

In contrast, a shared memory multi processor offers a single memory space used by all processors. Processors do not have to be aware where data resides, except that there may be performance penalties, and that race conditions are to be avoided. For this matter the OpenMP pattern is used and its main properties are described in section 3.2.

3.1. MPI

MPI is a language-independent communications protocol used to program parallel computers. Both point-to-point and collective communication are supported. MPI "is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation" [5]. MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today.

MPI is not sanctioned by any major standards body; nevertheless, it has become a de facto standard for communication among processes that model a parallel program running on a distributed memory system. Actual distributed memory supercomputers such as computer clusters often run these programs.
The principal MPI-1 model has no shared memory concept, and MPI-2 has only a limited distributed shared memory concept. Nonetheless, MPI programs are regularly run on shared memory computers. Designing programs around the MPI model (contrary to explicit shared memory models) has advantages over NUMA architectures since MPI encourages memory locality.

3.2. OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran on many architectures, including Unix and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

OpenMP is an implementation of multithreading, a method of parallelization whereby the master "thread" (a series of instructions executed consecutively) "forks" a specified number of slave "threads" and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

The section of code that is meant to run in parallel is marked accordingly, with a preprocessor directive that will cause the threads to form before the section is executed. Each thread has an "id" attached to it which can be obtained using a function (called `omp_get_thread_num()` in C/C++ and `OMP_GET_THREAD_NUM()` in Fortran). The thread id is an integer, and the master thread has an id of "0". After the execution of the parallelized code, the threads "join" back into the master thread, which continues onward to the end of the program.

By default, each thread executes the parallelized section of code independently. "Work-sharing constructs" can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both Task parallelism and Data parallelism can be achieved using OpenMP in this way.

4. Implementation

The implementation of the metaheuristics focused on the idea that it is possible to split the iterations of the main loop throughout the cores of a machine and collecting all results after the processing. Also, the use of the MPI model is interesting to maximize the variety of the output, since one instance of the application is created in each machine in the network and each one of them will create its own seed.

Processing the metaheuristic in a multicore environment is pretty useful and the use of OpenMP makes the implementation really fast and straightforward. In the Results section it will be described how this can increase the processing of the metaheuristic.

```
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 3: pseudo-code of an OpenMP version of a GRASP procedure.

In 3 the OpenMP version of the GRASP implementation is described. The routine `omp parallel` is used to fork additional threads to carry out the work enclosed in the construct in parallel; also the original process will be denoted as master thread, with a thread ID 0. Following, the routine `omp for` is used to
split loop iterations among threads (which are going to execute in multiple cores), this routine is also called loop constructs. In this work the \textit{Max\_iterations} variable was set to match the maximum number of cores available, in the Results section this is discussed more deeply.

The OpenMP Evolutative Algorithm follows the same idea, but in this case using a template with a while loop instead of for iterations. Anyways, the omp do will consecutively split the loop in threads to be executed in parallel. It's possible to see that the syntax for this code is almost the same as the GRASP implementation and that is part of the motivation behind using OpenMP for this class of algorithms.

\begin{algorithm}
\begin{procedure}{GA()}
  //generate threads
  #pragma omp parallel
  .
  //initializations
  .
  //splitting the while
  #pragma omp do
  \textbf{Build\_Initial\_Population()};
  \textbf{while} termination condition not satisfied \textbf{do}
    \textbf{repeat}
      \textbf{Crossover()};
      \textbf{Mutation()};
      \textbf{Evaluate\_Offspring()};
    \textbf{until} sufficient offspring created;
    \textbf{Select(newpopulation)};
  \textbf{end}
\end{procedure}
\end{algorithm}

Algorithm 4: an OpenMP Genetic Algorithms template.

The \textit{omp do} and \textit{omp for} belongs to the same class of routines and their execution is almost the same, splitting up the loop into threads. Note that it is also possible to create a Genetic Algorithm using for loops for a fixed set of iterations.

Moving into the MPI implementation, it is necessary to say that every machine in the network must have the same version installed and running. Since MPI use SSH to communicate, it is necessary some configuration before actually using the MPI, a fine explanation of this can be found in [5]. Also is important to mention that it is possible to have a heterogeneous network with different machines with different types of hardware, again in the Results section a description of the test ambient will show this more clearly.

To distribute the execution of many instances through the network, every machine must have a copy of the executable and be previously configured and mentioned before. Since the objective is to run different instance and not to split the work of a single one, it is not needed to introduce many MPI routines for message exchanging, it is only needed to distribute the work, start it in every machine available and wait for the output.

5. Results
The experiment environment consists of a LAN (Local Area Network) with three machines, as shown Figure 1. The machine configuration can be seen in Table 1. It may be noted that the computers configurations are heterogeneous, which proves that MPI can be used for a wide range of environments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Processor</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc1</td>
<td>Intel Pentium M processor 1.70GHz</td>
<td>1GB RAM</td>
</tr>
<tr>
<td>pc2</td>
<td>AMD Athlon II Dual-Core M300</td>
<td>4GB RAM</td>
</tr>
<tr>
<td>pc3</td>
<td>AMD Phenom 9850 Quad-Core Processor</td>
<td>4GB RAM</td>
</tr>
</tbody>
</table>

During the experiments, pc1 ran the sequential algorithm, as well as the OpenMP implementation.
Now, during the parallel processing using MPI and OpenMP, all three computers had the full program running and pc1 was the Master while pc2 and pc3 were Slaves.

Figure 1: The environment of experiments.

Results main focus is to show that a parallel implementation of such algorithms does not impact on the final solution. Also, that simply using the OpenMP pattern can speed up the process of the metaheuristic and combined with the MPI model can increase the possibility of finding better solutions.

Figure 2: EA’s final result comparison.

In Figure 2 are shown all results according to the best fitness according to [9], the main objective is to show that the parallel implementation can achieve close or same results of the sequential algorithm. This is also true for the GRASP metaheuristic presented in Figure 3.

Figure 3: GRASP final result comparison.
Following, the time comparison is presented in Figure 4 and Figure 5.

In Figure 4 the time comparison between the sequential version and the OpenMP implementation of the Evolutionary Algorithm. The time gained in the parallel implementation is visible and this is due the distribution of the metaheuristic iterations. This is also true to the GRASP presented in Figure 5.

For the time measure, each instance of each algorithm was executed over than 20 times to obtain the average time of the metaheuristic. This proves that a simple parallel implementation, with basic task distribution can speed up the metaheuristic. But it is necessary to say that for a more complex environment, with a population pool for instance or more optimizations may need some message exchanging controls and further configurations with MPI and OpenMP routines.

6. Future Work

The next steps are towards exploring the MPI, not only to create new instances but also helping to process the metaheuristics themselves. Since it is possible to split the process with a multicore approach a next natural step is to use a machine to process one or more iteration and use the cores to process inner parts of the metaheuristic separately.

Also, since two different metaheuristics are used and their converging speeds are different, a interest way can be to developer and analyze the use of other metaheuristics, such as: Iterated Local Search (ILS), Variable Neighborhood Search (VNS), Tabu Search or a hybrid metaheuristic version [2]. Besides this, in order to integrate intensification and diversification in search for new best solutions, using the Path Relinking technique, that consists in to explore trajectories between high quality solutions [11].

Acknowledgements

This work has been supported by CAPES. The authors are grateful for the assistance of Eugene Francis
References


