A proposal of metaheuristics to schedule independent tasks in heterogeneous memory-constrained systems

Javier Cuenca ¹, Domingo Giménez ², José-Juan López ³, Juan-Pedro Martínez-Gallar ⁴
¹Dpto. de Ingeniería y Tecnología de Computadores, Universidad de Murcia, Spain
²Dpto. de Informática y Sistemas, Universidad de Murcia, Spain
³Dpto. de Estadística, Matemáticas e Informática, Universidad Miguel Hernández, Elche, Spain
⁴jlopez@uhm.es ⁴jp.martinez@uhm.es

Abstract—This paper proposes some metaheuristics for the solution of a tasks scheduling problem. Independent tasks with different computational costs and memory requirements are scheduled in a heterogeneous system with computational and communication heterogeneity and memory constraints. Versions of the scheduling problem with and without communications and with constant and variable computation and communication costs are considered.

I. INTRODUCTION

The problem of mapping independent tasks to the processors in a heterogeneous system is considered. The tasks are generated by a processor and sent to other processors which solve them and return the solutions to the initial processor. So, a master-slave scheme is used [1]. Each task has a computational cost and a memory requirement. The processors in the system have different speeds and the communications between each pair of processors have different costs. Furthermore, each processor has a certain amount of memory, which imposes a restriction on the tasks which it can be assigned. This situation corresponds to real problems to be solved in heterogeneous clusters and distributed systems (i.e., triatomic molecules simulation [2] and solution of simultaneous equation models [3]). The goal is to obtain a tasks mapping which leads to a low execution time. To obtain the optimum mapping in the general case is an NP problem [4], and heuristic methods may be preferable. This paper analyses different metaheuristics (genetic algorithms, scatter search, tabu search and GRASP) [5] to approach the mapping problem.

The paper is organized in the following way: section 2 explains the basic scheduling problem together with some variants; in section 3 some metaheuristics for the solution of the proposed scheduling problems are analysed; and finally, section 4 summarizes the conclusions and outlines future research.

II. SCHEDULING PROBLEMS

If the system has \( p \) processors, numbered from 0 to \( p - 1 \), an array \( s = (s_0, s_1, \ldots, s_{p-1}) \) represents the speeds of the processors, another array \( m = (m_0, m_1, \ldots, m_{p-1}) \) represents the maximum memory available in each processor, and two-dimensional arrays \( t_s \) and \( t_w \) the start-up and word-sending times, with \( t_{s_{i,j}} \) and \( t_{w_{i,j}} \), the times for messages from processor \( i \) to processor \( j \).

For a processor with speed \( s \) the cost of a basic arithmetic operation is \( a = \frac{s}{2} \). The cost of a basic arithmetic operation in each one of the processors in the system is stored in an array \( a = (a_0, a_1, \ldots, a_{p-1}) \), and when a problem of arithmetic cost \( n^3 \) is solved in processor \( i \) the cost is \( a_i n^3 \).

The cost of communicating \( N \) data from the master processor to processor \( i \) is represented by \( t_{s_{M,i}} + t_{w_{M,i}} N \), and from processor \( i \) to the master by \( t_{s_{i,M}} + t_{w_{i,M}} N \). Each one of the \( t \) independent problems to solve has a volume of data represented in an array \( i = (i_0, i_1, \ldots, i_{t-1}) \), an arithmetic cost represented by \( c = (c_0, c_1, \ldots, c_{t-1}) \), and the size of the solutions (which are returned to the master) are \( o = (o_0, o_1, \ldots, o_{t-1}) \).

A. A problem with fixed arithmetic costs and no communications

Normally, communications are of lower order than computations. This occurs in the two examples mentioned. This means a scheduling which produces satisfactory results can be obtained without considering communications. In that case the problem is:

Given \( t \) tasks, with arithmetic costs \( c = (c_0, c_1, \ldots, c_{t-1}) \) and memory requirements \( i = (i_0, i_1, \ldots, i_{t-1}) \), and \( p \) processors with speeds \( s = (s_0, s_1, \ldots, s_{p-1}) \) and memory capacities \( m = (m_0, m_2, \ldots, m_{p-1}) \), from all the mappings of tasks to the processors, \( d = (d_0, d_1, \ldots, d_{t-1}) \) (\( d_k = j \) means task \( k \) is assigned to processor \( j \)), with constraints \( i_k \leq m_{d_k} \), find \( d \) with which the following minimum is obtained:

\[
\min_{\{d/ i_k \leq m_{d_k}\forall k=0,1,...,t-1\}} \max_{j=0,1,...,p-1} \left\{ \frac{1}{s_j} \sum_{l=0}^{t-1} c_l \right\}
\quad (1)
\]
There is a maximum of \( p^j \) assignations (with the memory constraints the number of possibilities may decrease), and it is not possible to solve the problem in a reasonable time by generating all the possible mappings. An alternative is to obtain an approximate solution using some heuristic method.

B. A problem with variable arithmetic costs and no communications

It has been observed that the cost depends on the problem size, because for different sizes the data are in different levels of the memory hierarchy. When the execution time is modelled taking these variations in consideration, the models are more accurate [6], [7]. The array of arithmetic costs of basic arithmetic operations is an array of functions \( a = (a_0(x), a_1(x), \ldots, a_{p-1}(x)) \). For the task \( j \) the cost of a basic operation in processor \( k \) would be \( a_k(i_j) \). So, the problem is:

Given \( t \) tasks, with arithmetic costs \( c = (c_0, c_1, \ldots, c_{t-1}) \) and memory requirements \( s = (s_0, s_1, \ldots, s_{t-1}) \), and \( p \) processors with basic arithmetic operation costs as function of the problem size, \( \alpha = (a_0(x), a_2(x), \ldots, a_{p-1}(x)) \), memory capacities \( m = (m_0, m_2, \ldots, m_{p-1}) \); from all the mappings of tasks to the processors, \( d = (d_0, d_1, \ldots, d_{t-1}) \), with the constraints \( i_k \leq m_{d_k} \), find \( d \) with which the following minimum is obtained:

\[
\min_{d/ i_k \leq m_{d_k} \forall k = 0, \ldots, t-1} \max_{j = 0, \ldots, p-1} \left\{ \sum_{l=0}^{t-1} a_j(i_l) c_l \right\} \quad (2)
\]

C. A problem with fixed arithmetic and communication costs

The cost of communications can become very important for different reasons: they introduce an additional cost to the computational cost; the work of some processors can be stalled while waiting for messages from other processors; the messages from one processor are not sent simultaneously to all the other processors, ... So, the communications must be taken into account to obtain a more accurate model and the order in which tasks are assigned to processors must be determined. The problem becomes a scheduling problem, where the order in which tasks are assigned must be decided and the processor to which each task is assigned must also be established. In this case the problem is:

Given \( t \) tasks, with arithmetic costs \( c = (c_0, c_1, \ldots, c_{t-1}) \) and memory requirements \( s = (s_0, s_1, \ldots, s_{t-1}) \), and \( p \) processors with speeds \( s = (s_0, s_2, \ldots, s_{p-1}) \) and memory capacities \( m = (m_0, m_2, \ldots, m_{p-1}) \); from all the mappings of tasks to the processors, \( d = (d_0, d_1, \ldots, d_{t-1}) \), with the constraints \( i_k \leq m_{d_k} \), find \( d \) with which the minimum parallel execution time is obtained.

No formula similar to those of equations 1 and 2 is available, and to obtain the execution time for each scheduling the execution can be simulated.

D. A problem with variable arithmetic and communication costs

In this case, the communications are considered, as in the previous case, and the arithmetic costs and the start-up and word-sending times are considered as function of the problem size or the volume of data to be sent. This problem is also a scheduling problem, with the number of possible configurations \( !p^j \), and where the cost of each configuration can be estimated by simulation. In this case the problem is:

Given \( t \) tasks, with arithmetic costs \( c = (c_0, c_1, \ldots, c_{t-1}) \) and memory requirements \( s = (s_0, s_1, \ldots, s_{t-1}) \), and \( p \) processors with functions of basic arithmetic cost \( a = (a_0(x), a_2(x), \ldots, a_{p-1}(x)) \), functions of start-up and word-sending times from the master to the slave processors, \( t_{SM} = (t_{w_{0,M}}, t_{w_{1,M}}, \ldots, t_{w_{p-1,M}}) \), and functions of start-up and word-sending times from the slaves to the master, \( t_{SW} = (t_{s_{0,M}}, t_{s_{1,M}}, \ldots, t_{s_{p-1,M}}) \), and with functions of start-up and word-sending times from the slaves to the master, \( t_{SW} = (t_{s_{0,M}}, t_{s_{1,M}}, \ldots, t_{s_{p-1,M}}) \), and with memory capacities \( m = (m_0, m_2, \ldots, m_{p-1}) \); from all the mappings of tasks to the processors \( d = (d_0, d_1, \ldots, d_{t-1}) \), with the constraints \( i_k \leq m_{d_k} \), find \( d \) with which the minimum parallel execution time is obtained.

E. Other possibilities

It could be convenient to have models which give a good reflection of the reality. The previous models can be satisfactory depending on how well the values of the parameters have been estimated. They can be estimated at installation time or when the routine is executed. At installation time it would not be a problem to spend a long time on the estimation, but at running time the estimation represents an overhead which can make the method impracticable. Likewise, the optimization problem to obtain an optimum scheduling is NP, and the application of heuristics to obtain approximate solutions is compulsory, but it can also mean a large execution time. There are alternatives to the solution of a scheduling problem.

1) Dynamic assignation of tasks: Master-slave schemes can be implemented using dynamic assignation of tasks. The master manages a pool of pending tasks, and deals with the requests of the slaves when they finish the task they have assigned. But it is easy to find some examples in which this dynamic strategy does not give the best result. For example, with two tasks of costs 100 and 50, and two slaves with basic arithmetic operation costs 1 and 4, the first task could be assigned to the first processor and the second task to the second processor, and the execution time would be 200 (communications are not considered). If the two tasks are assigned to the first processor the execution time is 150.

The memory constraints also means that the dynamic approach is non optimal. Let us take two tasks, each with cost 100 and memory requirements 80 and 100, and two processors with memory constraints 100 and 80 and costs of basic operations 2 and 3. The first task could be assigned to the first processor, and the second can not be assigned to the second processor due to memory constraints. The execution time is 400, but if the first task is assigned to processor two and task two is assigned to processor one the time would be 300.
2) **Adaptative metaheuristics:** One way of avoiding excessive delays in beginning the computation in the slaves is to obtain an initial mapping quickly with a metaheuristic which could run only a low number of iterations. The slaves would work on the solution of their tasks, and the master could continue to improve the initial solution, while waiting asynchronously for messages from the slaves. When the solution of a task is received, the following task is assigned according to the best mapping solution the master has found so far.

3) **Problems with slave multiprocessors:** Until now we have supposed a task is assigned to one processor in the system, and each processor has its own local memory. But this is not the only possibility. Other possible cases are:

- The system could be a shared memory processor and there could be a limit in the total memory, but not in the memory of each processor.
- There could be a network of processors and the tasks could be assigned to a group of processors which would solve the task in parallel, with a memory limit, which could be the sum of the local memories of each processor.
- The system could be a distributed grid, with monoprocessors, shared memory multiprocessors, clusters of processors, constellations, ...

Let us consider briefly the simplest case, with a network of processors, with fixed arithmetic costs and no communications.

**Mapping problem with parallel slaves, fixed computational cost and no communications.** Given $t$ tasks, which can be solved by a variable number of processors in a system of $p$ processors, and with memory requirements $i = (i_0, i_1, \ldots, i_{t-1})$, and the $p$ processors with memory capacities $m = (m_0, m_2, \ldots, m_{p-1})$; from all the possible arrangements of tasks $l = (l_0, l_1, \ldots, l_{t-1})$ and all the possible mappings of tasks to sets of processors $d = (d_0, d_1, \ldots, d_{t-1})$ (each $d_k$ is an array $d_k = (d_{k,0}, d_{k,1}, \ldots, d_{k,t-1})$, with each $d_{k,j} = 0, 1$, where $d_{k,j} = 0$ means processor $j$ is not in the set of processors to which task $k$ is assigned), with the restrictions $\sum_{j=0}^{p-1} d_{k,j} \leq m_j$, and with the cost of the tasks in the set of all the possible processors subsets given by an array of functions $c = (c_0, c_1, \ldots, c_{t-1})$ where each $c_j$ is a function $c_j : \{0,1\}^{p-1} \rightarrow \mathbb{R}^+$; find $l$ and $d$ with which the minimum parallel execution time is obtained.

### III. Application of metaheuristics to the scheduling problem

The application of metaheuristic methods to a simplified version of the scheduling problem is analyzed. The problem is that with fixed cost and no communications. For $t$ tasks, the mapping $d = (d_0, d_1, \ldots, d_{t-1})$, $\forall i = 0, 1, \ldots, t-1$, $0 \leq d_i \leq p - 1$, with the lowest associated parallel execution time is approximated.

The methods considered are: genetic algorithms, scatter search, tabu search and GRASP. The four metaheuristics are analysed from the same perspective, identifying common routines and element representations. The goal is to obtain a mapping with an associated modelled time close to the optimum, but with a low assignment time. A general metaheuristic scheme is studied [8]. One such scheme is shown in algorithm 1.

#### Algorithm 1: General scheme of a metaheuristic method

1. **Inicilice($S$);**
2. **while not EndCondition($S$) do**
   3. **SS = ObtainSubset($S$);**
   4. **if [SS] > 1 then**
      5. **SS1 = Combine(SS);**
   6. **else**
      7. **SS1 = SS;**
   8. **end**
   9. **SS2 = Improve(SS1);**
   10. **S = IncludeSolutions(SS2);**
11. **end**

#### A. Genetic algorithms

We analyse the characteristics of genetic algorithms by describing how the sets, elements and routines in the general metaheuristics scheme (algorithm 1).

- Genetic algorithms work with a population (set $S$) formed by individuals.
- In ObtainSubset some of the individuals are selected randomly, with more probability for the individuals of better fitness function (equation 1).
- The selected individuals are crossed (routine Combine). One possibility is to cross pairs of individuals to obtain some decendants which inherit some of the characteristics of the two parents. So, a new group of individuals is obtained ($SS1$).
- Some of the individuals are selected (routine Improve), with a low probability, to obtain other individuals which can differ greatly from the original ones. This is done with mutation operands. The new set obtained is $SS2$.
- The new generation is obtained using some selection technique in routine IncludeSolutions.

To tune the genetic scheme to the mapping we are working with it is necessary to decide the value of some parameters and how the routines in the scheme work. There are a lot of possibilities. Some of these are considered:

- The size of the population in a genetic algorithm is normally large, but too many individuals could mean a large execution time. To decide the influence of this parameter, experiments with varying sizes of the population must be carried out.
- Good initial values could contribute to accelerating the search. So, the generation of the initial population could use some criteria to include potentially good elements. One possibility is to select for each task $j$ the processor to which it will be assigned ($d_j$), assigning more probability to processors of higher speeds. Only processors which fit the memory constraints can be selected.
- The crossover operator must be designed in such a way that the generated individual inherits some characteristics (the best ones) from their parents. Because
each individual represents a mapping \( d \), where the order of the components has no relevance, an easy way of combining two mappings could be to exchange half of the mappings: when two mappings \( d^{(1)} \) and \( d^{(2)} \) are combined, two new mappings are generated with the forms 

\[
\begin{align*}
d^{e,1} &= \left( a^{e,1}_0, \ldots, a^{e,1}_l, a^{e,1}_{l+1}, \ldots, a^{e,1}_{l+2k-1} \right) \\
d^{e,2} &= \left( a^{e,2}_0, \ldots, a^{e,2}_l, a^{e,2}_{l+1}, \ldots, a^{e,2}_{l+2k-1} \right)
\end{align*}
\]

- The mutation operator must contribute to diversify the search. With a low probability an individual would be selected to mutate, and a new individual whose characteristics greatly differ from those of the initial individual would be generated. One possibility is to generate a permutation of \( p \) elements and to modify the mapping according to the permutation. It is also possible to select one component of the mapping and to assign a new value between the processors which satisfy the memory constraints.

- The convergence criterion is normally that the best fitness value from the individuals in the population does not change over a number of iterations. If the number of iterations is large, better values could be obtained, but more time would be necessary. Because the time must be reduced, the number of iterations to converge must be low.

When the mapping problem to solve takes into consideration the communications, the characteristics of the genetic algorithm are the same as in the previous case, the only difference being the way in which fitness is obtained, which is done by simulation.

Where the order of the tasks \( l = (l_0, l_1, \ldots, l_{t-1}) \) and the assignments \( d = (d_0, d_1, \ldots, d_{t-1}) \) must be obtained, the fitness for each individual is obtained with equation 2. Some modifications are necessary to tackle a scheduling problems:

- To obtain good initial values the most expensive tasks could be selected first, and assigned to the fastest processors.

- In the crossover operator the assignments could be selected as previously, but it is less clear how the orders can be combined to inherit part of the characteristics of the parents. One possibility is to combine \( l \) as \( d \) is combined, and to eliminate the possible repetitions \( (T_{k,0}^{(j)} = T_{k,0}^{(j)}) \) by simply assigning randomly the values not included in \( T^{(j)} \) to one of them. The memory constraints could be violated, and the process repeated until they are fulfilled.

- The mutation can include a permutation of \( l \), of the selection of two elements to exchange. Memory constraints must be taken in account.

### B. Scatter Search

Scatter Search is a very aggressive search method that attempts to find high quality solutions fast. A description of the sets, elements and routines in the general metaheuristic scheme (algorithm 1) is given:

- It works with a set \( S \) with a reduced number of elements. Typically, the reference set has 20 solutions or less.

- In ObtainSubset, certain elements of the reference set are selected to be combined. It is possible to select all the elements for combination, or to select the best elements to be combined with the worst ones.

- The combination process could consider all pairs of solutions in the reference set, and it is necessary to keep the cardinality of the set small.

- The elements are improved (routine Improve) in some way, possibly by searching for a better element in its neighbourhood. The new set obtained is \( SS2 \).

- The new reference set is obtained using some selection technique in routine IncludeSolutions. The best elements are selected, as well as some elements which are scattered with respect to them.

- The convergence criterion is the same as in genetic algorithms, but in this case less iterations are normally needed due to the improvement of the elements at each step.

To tune Scatter Search to the problem we are working with, it is necessary to decide the value of some parameters and how the routines in the general scheme work:

- Experiments to analyse how the size of the population influences the behaviour of the method will be carried out.

- Good initial values could contribute to accelerate the search. The improvement method could be used in the initialization. The influence of the goodness of the initial set in the behaviour of the search will be studied.

- The combination could be that of the genetic method, although other combinations will be analysed. For example, vectors \( d^{(1)} \) and \( d^{(2)} \) could be combined by substituting components \( d^{(1)}_i \) and \( d^{(2)}_i \) in the other vector, and including in the new elements \( d^{(1)}_i \) and \( d^{(2)}_i \) the component with which best values are obtained.

- The improvement phase could be implemented with a greedy method which works evaluating the fitness value of the elements obtained with the \( p \) possible values (with memory constraints) in each component.

- The most promising elements are included in the reference set, as are those which are most scattered with respect to the most promising ones. The elements with longest Euclidean distance to the most promisings of the set, or those which are more “different” (element by element) with respect to the most promising ones, can be considered as most scattered.

Some modifications are necessary to tackle a scheduling problem where the order of the tasks \( l = (l_0, l_1, \ldots, l_{t-1}) \) and the assignments \( d = (d_0, d_1, \ldots, d_{t-1}) \) must be obtained:

- The most expensive tasks could be selected first, and assigned to the fastest processors. After generating these initial values, they can be improved with a greedy method which could work by evaluating all the possible tasks \( l \) and processors \( d \) for each pair \( (l_i, d_i) \) changed to
all the possible tasks and processors. The corresponding exchange of elements in \( l \) (if \( l_i^k = l_j \) then \( l_j^k = l_i \)) must be taken into account.

- In the combination the assignations could be selected as in the mapping problem.
- The improvement phase would be like that explained for the initialization.
- The most scattered elements are selected as in the mapping problem, but now considering distances between vectors \( l \) and \( d \).

C. Tabu search

Tabu search is a local search technique which uses memory structures to guide the search. A description of the sets, elements and routines in the general metaheuristic scheme (algorithm 1) is given:

- The set to work with has only one element (\(|S| = 1\)), and the search is done in the neighbourhood of this element (ObtainSubset and Combine are not necessary).
- Some elements in the neighbourhood of the actual element are analysed, excluding those elements in a list of tabu elements previously analysed. The best elements from those analysed is taken as the next solution (routine IncludeSolutions).
- A number of iterations without improving the solution is used as convergence criterion.

To tune Tabu Search to the problem we are working with, it is necessary to decide the value of some parameters and how the routines in the general scheme work:

- A good initial value could contribute to accelerating the search. Each task can be assigned to a processor randomly, with the same probability for each processor, or with the probability proportional to the processor speed, or a greedy method can be used. Some random and greedy methods are compared.
- A large neighbourhood will produce a large execution time for each iteration, but the number of iterations could reduce.
- The tabu list also influences the execution time and the speed of convergence of the method. Large lists require more time to be evaluated, but avoid re-evaluating more elements, and so the search is more guided.
- The number of iterations without improving the solution as convergence criterion will also be studied.

Some modifications are necessary to tackle a scheduling problem where the order of the tasks and the assignations must be obtained:

- The most expensive tasks could be selected first, and assigned to the processors like in the mapping problem. The greedy method to improve these initial solutions could be that of scatter search.
- The neighbourhood could include the actual order of tasks, and all the orders obtained by exchanging two components of \( l \). For each of these orders the possible values of \( d \) could be those considered in the mapping problem.

- The tabu list could store the same information as in the previous case (the task and the processor to which it has been assigned) or more information (the task, its position and the processor).

D. GRASP method

The GRASP method (Greedy Randomized Adaptative Search Procedure) works by multiple iterations. In each iteration a feasible solution is obtained (random part of the method), and in a second phase the best solution is improved by a local search (greedy part). The characteristics of the method are analysed:

- In each iteration the cost of each possible candidate is evaluated, and a number of candidates (parameter to be tuned) are selected to be included in a set of solutions. These candidates could be selected with some degree of randomness by assigning more probability to elements with better fitness function, or the best ones could be selected. The set of candidates could be the same at each iteration, so it would not be necessary to re-evaluate them, or the set could change, for example eliminating in an iteration the candidates selected in the previous ones. One element from the set of solutions is selected to constitute the set \( SS (|SS| = 1) \). At this step, the goodness of the elements can also be used for the selection. The way in which they are selected can vary with the iteration.
- Routine Improve consists of a local search to improve the element selected. Some greedy method can be used, or all the elements in the a neighbourhood of the selected one can be analysed.

To tune the GRASP method to the problem, it is necessary to decide the value of some parameters and how the routines in the general scheme work:

- Experiments with a fixed number of iterations and with a number of iterations without improving the best solution will be carried out.
- The possible candidates are all the tasks to processors mappings (\( p_i^t \)) and it is not possible to evaluate all them. Another possibility is to generate some promising candidates. The cardinality of the set of candidates is a parameter to be tuned.
- Once the set of candidates has been formed, the cost of each possible candidate is evaluated, and a number of candidates (parameter to be tuned) are selected to be included in a set of solutions.
- To summarize, there are three sets: that of all the mappings (\( M \)), the candidates set (\( C \)) and the solutions set (\( S \)); and to pass from one set to the following, four schemes can be used: select all (\( All \)), make a random uniform selection (\( RaUn \)), use an exact greedy method (\( GrEx \)) or a random greedy method (\( GrRa \)). Some of the combinations are more meaningful than others, but all could be tested, as well as some variants of the greedy and random selections. The cost for each combination is shown in table I.
Some modifications are necessary to tackle a scheduling problem where the order of the tasks and the assignments must be obtained:

- The selection can be made as in the previous problem, but with the fitness functions calculated by simulation, which could mean larger execution times are necessary, and the sizes of the sets must be reduced.
- The greedy methods to improve the selected element could be those of scatter search.

E. Preliminary experimental results

Some preliminary experimental results obtained with GRASP for the simplest problem are shown. The method has been compared with a backtracking. Experiments for different numbers of tasks have been carried out. The size of each task has been randomly generated between 1000 and 2000, the arithmetic cost is \( n^3 \), and the memory requirement \( n^2 \) ([2]). The number of processors in the system coincides with the number of tasks. The costs of basic arithmetic operations have been randomly generated between 0.1 and 0.2 \( \mu \text{sec} \), and the memory of each processor is between half the memory needed by the biggest task and one and a half times this memory. GRASP has been tested with different values of the parameters and different versions of the basic routines. Satisfactory results are obtained when:

- The reference set \( S \) has 10 elements. This number is small to avoid large execution times.
- The elements in \( S \) have been generated randomly assigning the tasks to the processors, with the probability proportional to the processor speed.
- The number of iterations has been fixed at 10.
- The element selected from \( S \) has been chosen randomly, with more probability for the elements with better objective function.
- The selected element is improved using a greedy method, which works by selecting for the processor with highest execution time a task which could be assigned to another processor, and so obtains a lower objective function.

Table II compares the results of backtracking and GRASP. The time to obtain the mapping and the value obtained are shown. Mapping times are the means of five executions. For small problems backtracking obtains the optimum mapping in a low time, but the time increases with the number of tasks. For 15 and 16 tasks the times are not shown because the program did not finish in one hour. The times of GRASP are always low, and it can be used to decide the mapping at running time.

Only for the problem with 14 tasks does GRASP not obtain the optimum mapping. To obtain a better mapping, the value of some parameters can be varied: with 20 elements on \( S \) and 20 iterations the simulated time is 1135, and with 10 elements and 100 iterations the time is 1124, which is the optimum value.

**IV. Conclusions and possible future studies**

The characteristics of metaheuristics of different types and how to tune them to the mapping problems are analysed. Once the parameters and the routines to tune have been identified, and indications of how to tune them have been given, the experiments to obtain satisfactory versions of the metaheuristics will be carried out. After that, the metaheuristics will be applied to other mapping and scheduling problems, some of which have been indicated here.

**Acknowledgment**

This work has been funded in part by the Consejería de Educación de la Comunidad de Murcia, Fundación Séneca, project number 02973/PI/05.

**References**


