Evolutionary Scheduling of Parallel Tasks Graphs onto Homogeneous Clusters

Sascha Hunold
LIG Laboratory
Grenoble, France
sascha.hunold@imag.fr

Joachim Lepping
Robotics Research Institute
TU Dortmund University, Germany
joachim.lepping@udo.edu

***Abstract*—Parallel task graphs (PTGs) arise when parallel programs are combined to larger applications, e.g., scientific workflows. Scheduling these PTGs onto clusters is a challenging problem due to the additional degree of parallelism stemming from moldable tasks. Most algorithms are based on the assumption that the execution time of a parallel task is monotonically decreasing as the number of processors increases. But this assumption does not hold in practice since parallel programs often perform better if the number of processors is a multiple of internally used block sizes. In this article, we introduce the Evolutionary Moldable Task Scheduling (EMTS) algorithm for scheduling static PTGs onto homogeneous clusters. We apply an evolutionary approach to determine the processor allocation of each task. The evolutionary strategy ensures that EMTS can be used with any underlying model for predicting the execution time of moldable tasks. With the purpose of finding solutions quickly, EMTS considers results of other heuristics (e.g., HCPA, MCPA) as starting solutions. The experimental results show that EMTS significantly reduces the makespan of PTGs compared to other heuristics for both non-monotonically and monotonically decreasing models.**

***Keywords*-task scheduling; parallel tasks; task graphs; evolutionary algorithm; cluster**

I. Introduction

Scientific workflows are an important type of parallel task graphs and are often processed on computational grids. Many scientific workflows contain only a few parallel tasks. Yet, as Cirne *et al.* stated, almost 98% of parallel jobs submitted to computational clusters are moldable [1]. The number of processors of a moldable task is determined before its execution and stays unchanged during execution [2]. If these parallel tasks are combined, a parallel task graph (PTG) arises. Several algorithms can be expressed as PTGs such as Strassen's matrix multiplication or the Fast Fourier Transformation (FFT) [3]. The nodes of a PTG denote the computations and the edges denote data or control dependencies. Executing a PTG leads to a mixed-parallel schedule as nodes are implemented in a data-parallel way and independent tasks can be executed concurrently.

Let us examine the execution time of the parallel matrix multiplication routine PDGEMM from ScaLAPACK [4] for two matrix sizes, which is shown in Figure 1. As we can see, the execution time is not monotonically decreasing, but most scheduling algorithms assume a monotonically decreasing model of the execution time for computing the schedule. Hence, applying a non-monotonically decreasing model can lead to inefficient decisions of these algorithms.

For that reason, we focus on the question whether an evolutionary algorithm (EA) for scheduling PTGs can cope with irregularities in the execution time model for parallel tasks. We introduce the algorithm EMTS, which can be used with an *arbitrary execution time model*. We show that EMTS, when compared to other heuristics, produces better schedules with respect to the makespan objective. This holds for non-monotonically and monotonically decreasing execution time models.

The paper is organized as follows. Section II details the problem and discusses related approaches. In Section III the evolutionary method EMTS is presented. The methodology and the setup of the experiments is described in Section IV. The experimental results are discussed in Section V and Section VI draws conclusions.

II. Problem Description and Related Work

A. Application and Platform Model

A PTG (mixed-parallel application) can be represented by a directed acyclic graph (DAG) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_i \,|\, i = 1, \dots, V\}$ is a set of nodes that represent the tasks and $\mathcal{E} = \{e_{i,j} \,|\, (i,j) \in \{1, \dots, V\} \times \{1, \dots, V\}\}$ is a set of edges representing task interdependencies. A PTG can be executed on P identical processors interconnected by a network, so that each pair of processors can communicate. It is assumed that the parallel tasks are *moldable*, i.e., executable by an arbitrary number of processors ($1 \leq p \leq P$).

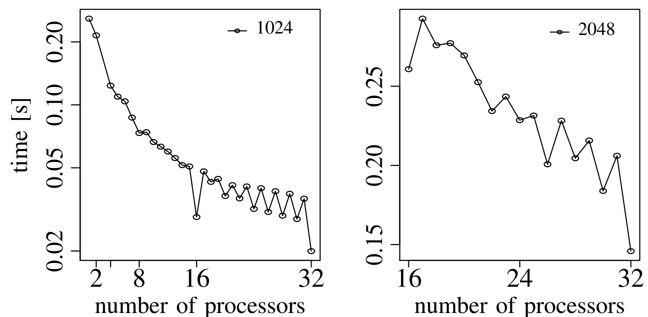


Figure 1. PDGEMM timings on the Cray XT4 of LBNL.

This number p is specified at a job's start time and is non-adjustable for the duration of its execution.

In the present article, we focus on the overall makespan as optimization objective, which is justified by considering the following scenario: To execute a PTG on a cluster, the user first requests a time slot from the local job scheduler (e.g., PBS). After the application has been granted several processors, the PTG scheduler computes a schedule while trying to minimize the overall execution time of the job.

B. Related Work

Scheduling algorithms for PTGs have to determine (1) how many processors are allocated to a task and (2) to which processor set a task is mapped. Most algorithms can be categorized in two groups:

One-step algorithms answer both questions (allocation and mapping) for each task iteratively. The algorithms pick the next tasks to be executed, determine its allocation, and finally map it onto the platform. Examples of such algorithms are LoC-MPS [5] or the algorithm by Boudet *et al.* [6]. This method has the advantage of producing short schedules, but the drawback is the amount of time spent for computing the schedules. Since a final decision of placement is made for a task in each iteration, the algorithms can consider several optimization criteria such as reducing the data redistribution overhead between dependent tasks. However, to make a good decision for each task, the algorithms need to apply a look-ahead strategy, which makes them computationally expensive.

Two-step algorithms decouple allocation and mapping phase into two distinct steps. First, the allocations of all tasks are determined and second, these allocations are mapped onto the cluster. The advantage of this method is that it is (usually) less computationally demanding than a one-step approach, but it produces slightly longer schedules. The TSAS algorithm [7] (Two Step Allocation and Scheduling) was one of the first algorithms of that class. TSAS uses convex programming to solve the allocation problem, which entails high computational costs. CPR [8] (Critical Path Reduction) and CPA [9] (Critical Path and Area-based) determine the processor allocations by minimizing the length of the critical path of the PTG. Several extensions of CPA have been proposed as it provides a good trade-off between the computational costs and the resulting makespan. HCPA [10] (Heterogeneous CPA) extends CPA to multi-cluster platforms, while MCPA [11] (Modified CPA) and MCPA2 [12] make better use of the potential task parallelism by bounding the allocation size per DAG level. RATS [13] (Redistribution Aware Two Step) attempts to adjust previously computed allocations in the mapping step to reduce redistribution costs. The algorithm BiCPA [14] tries to optimize both, the completion time of the PTG and the amount of resources used.

Most scheduling algorithms use one of two different models to predict the execution time of parallel tasks. The first is based on the speed-up model of Downey [15], while the second uses Amdahl's law [16]. In both models, the execution time of a task decreases as the number of processors increases. In Downey's model, each task is characterized by a parameter to represent the average parallelism and a parameter to define the variance of parallelism.

The "monotonous penalty assumption" (i.e., the execution time of a task is non-increasing) has also been addressed by Günther *et al.* [17]. In this work, the authors prohibit the assignment of those processor allocations that would violate the monotonous penalty assumption.

Finding a good empirical model for predicting the execution time of a parallel application is challenging. Linear regression can help to provide such a function, but many experiments are required to obtain robust fits, see Pfeiffer and Wright [18] for a case study on this problem.

Further, evolutionary algorithms (EA) have been applied to the task scheduling problem in various ways. A good survey is provided by Wu *et al.* [19]. The two main approaches comprise methods that use an EA in combination with other list scheduling techniques and methods that use an EA to evolve the actual assignment and sequence of tasks onto processors [20]. Previous approaches mainly differ in the fact that single-processor tasks are considered, whereas we address the problem of scheduling a DAG of moldable tasks on a cluster. Due to the iterative search concept, evolutionary approaches produce improved solutions at the expense of longer execution times. In the present paper, we therefore limit the amount of computational time that is spent for the evolutionary meta-optimization to be applicable under real-world time constraints.

C. Problem Statement

To the best of our knowledge, most scheduling algorithms for PTGs of moldable tasks assume that the execution time of a task decreases as the number of processors increases. In this article, we tackle the question of how to overcome this limitation. One could design another heuristic that would take care of the fact that increasing an allocation could also increase the execution time. However, the algorithm designer would remain with the question how far he or she should look ahead to find a better size of an allocation. Since the allocation and mapping step are highly dependent on each other, adding k processors (let k be the maximum look-ahead) to an allocation might decrease the parallel execution time of the corresponding task, but it might, at the same time, increase the overall makespan as it may impact the packing of tasks.

Thus, we want to find a meta-heuristic for this scheduling problem that is independent of the model used to describe the run time of parallel tasks. For that purpose, we apply an evolutionary search strategy, which may overcome the

problem of being trapped in local optima, but which cannot be guaranteed. The main advantages of evolutionary search strategies are: (1) They can cope with large problem instances; (2) They can be applied to an unknown search space without having a precise mathematical model of its structure; (3) They are capable of optimizing a solution starting anywhere in search space. The downside is that evolutionary approaches tend to converge slowly to the optimum, and one has usually no measure of how close the current result is to the optimal solution.

While being aware of these properties, we want to design an evolutionary algorithm for this scheduling problem that provides a good trade-off between the time used to compute the solution and the resulting makespan. Since we can usually trade time for solution quality, we focus on a given time constraint.

III. Algorithm EMTS

The algorithm presented in this section is based on the application and platform models (PTG), which were introduced in Section II-A. In the context of this article, a homogeneous cluster comprises the same type of computational nodes, i.e., the same processor and the same amount of memory. In addition, communication costs between tasks are not considered. If communication or data redistributions are necessary, they need to be included in the execution time model of the parallel tasks.

A. Designing the Algorithm

The Evolutionary Moldable Task Scheduling (EMTS) utilizes a two-step approach for solving the scheduling problem. In the first step, the allocations of each task are computed while in the second step, the allocations are mapped onto the processors of the system.

Mapping function: As the execution time model of parallel tasks has no influence on the mapping step, we start with defining the mapping function of EMTS. Since the mapping function also evaluates the fitness of all individuals, it should be as fast as possible and produce short schedules. Previous work showed that a list scheduling approach leads to efficient schedules [9]. In the list scheduling algorithm used by EMTS, the ready nodes are sorted by decreasing bottom level and each ready node v is mapped to the first processor set that contains $s(v)$ available processors[1]. The fitness level of a set of allocations is defined as the resulting makespan of the PTG. A smaller makespan corresponds to a better fitness of an EA's individual.

Allocation function: The EA modifies the processor allocations of tasks and its goal is to find the right allocation for each task so that the resulting fitness is optimized. The set of allocations is encoded as individual I. For a task

[1] $s(v)$ is the allocation size of node v, and the bottom level $bl(v)$ is the length of the longest path from a node v to the sink of the PTG including its own execution time.

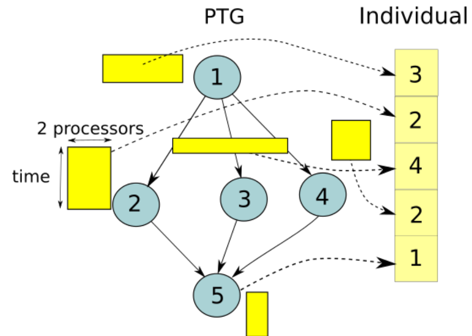


Figure 2. Encoding of individuals: the allocation $s(v_i)$ of node v_i is stored at position i.

v_i of PTG $\mathcal{G}_j$ the individual $I_j(i)$ holds the number of processors allocated to v_i at position i. Thus, $I_j(i) = s(v_i)$ for $1 \leq i \leq V$, where $s(v_i)$ denotes the current allocation of task v_i. Figure 2 illustrates this encoding of the allocations of a PTG. The depicted PTG contains five nodes and each node possesses a processor allocation, e.g., three processors are allocated to node 1. An individual is the collection of all allocations of a PTG. So, the number of processors of node 1 is stored at the first position of the individual.

The EA modifies the individuals randomly to optimize the schedule. Thus, the objective function can be defined as follows: For a given PTG $\mathcal{G}$ and fitness function $F : S \to \mathbb{R}$, the EA tries to find a set of allocations $S = \{s(v_0), \ldots, s(v_V)\}$ that minimizes F. If we consider PTGs of about 100 tasks and a platform with hundreds of processors, the search space to find the best solution S^* is enormous. Nonetheless, EMTS may find an S which is closer to S^* than the solutions produced by other heuristics. This leads to two main questions: (1) Where should the EA start searching? Or to put it in another way, how should the original individuals be initialized? (2) How to produce new individuals from existing ones?

B. Retrieving Starting Solutions

To obtain starting solutions, EMTS makes use of results produced by other heuristics. In the present work, we execute the allocation functions of MCPA [11] and HCPA [10] and encode their results as individuals in the initial population. These allocation functions were chosen because they produce sufficiently good solutions in a short amount of time.

Additionally, we designed another heuristic to create a starting individual: First, the bottom level of each task is computed assuming that each task is allocated to one processor. Having the bottom level, EMTS determines the critical path of the PTG. The idea is to assign all processors of the system to nodes on the critical path. Since several nodes on a precedence level have similar or equal criticality (bottom level), we consider several *almost critical paths* to make this heuristic effective. Hence, we separate the nodes by precedence level (depth of the nodes from the

source) and share all processors of the system among the Δ-critical nodes of a layer. The concept of Δ-critical tasks was successfully applied by Suter [21]. In the present work, the set of Δ-critical tasks of a precedence layer k includes all tasks v_k for which $\{v_k \,|\, bl(v_k) \geq \Delta \cdot \max_i(bl(v_i))\}$ holds. The parameter Δ, $0 \leq \Delta \leq 1$, defines the minimum bottom level of a task to be considered critical. By applying this heuristic, tasks on the critical path in one precedence level receive P/c_l processors and non-critical ones receive 1 processor (c_l is the number of almost critical tasks of level l).

C. Creating New Individuals

The reproduction of new individuals in each EA iteration is driven by the requirement to obtain a good solution quickly. In many EAs, new individuals are produced by mutation *and* recombination (e.g., crossover). EMTS has a particular encoding, where alleles represent processor allocations, which belong to dependent tasks. The chances are therefore small to improve the solution quality when performing a crossover on random individuals. Additionally, it has been shown for several combinational problems that relying only on mutation operators can be sufficient [22]. Thus, we decided to apply a mutation-only strategy and are also aware of the fact that specially tailored recombination methods might improve the EA performance. Yet, such fine tuning of the EA is not in the scope of the current paper.

The mutation strategy applied to EMTS works as follows: First, we pick the scaling factor $f_m \in]0, 1]$ that influences how many allocations of an individual should be changed. Intuitively, it is better to change many allocations in the beginning of the EA and to decrease this factor when better solutions have been already found. In this way, we adapt the mutation step size in each iteration of the EA, which allows a good search space exploitation at the beginning and favors a concise convergence in further generations. So, EMTS changes fewer allocations when the evolutionary step u increases. Let the PTG contain V nodes and let U be the number of generations to be executed. The number of allocations that are modified in generation $u \leq U$ is given by $m = (1 - \frac{u}{U}) \cdot f_m \cdot V$.

D. Mutation Operator

In the previous section, we defined how many alleles per individual are mutated in each step, but it needs to be clarified by how many processors they are modified. As the mutation process is purely random, any uniform distribution could be applied. Nevertheless, a uniform distribution might not converge to the optimum very quickly. The reason is that, the probability of changing an allocation by 1 or k processors is the same for such distribution. The number k could be very large or very small (even negative). However, in reality, it is less likely that changing an allocation by a large number of processors improves the schedule length.

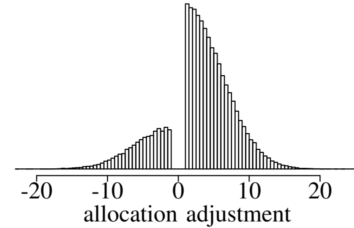


Figure 3. Probability density function of the mutation operator, $\sigma_1 = \sigma_2 = 5$ and $a = 0.2$.

Hence, we would like to decrease the probability of changing an allocation by k processors as k increases.

So, the definition of the mutation operator is driven by two main objectives. The first objective is that the operator should support the shrinking and stretching of allocations. The shrinking is necessary to improve the potential packing of allocations, e.g., to enable backfilling of tasks. The second objective is that it should be more likely to change an allocation by a few processors than by many, which helps to reduce oscillations during the convergence process. Thus, we add the constraint that the shrinking of allocations is less likely than the stretching of allocations.

After specifying the constraints, we can define the mutation operator to adapt the size of an allocation randomly. Let L be a Bernoulli variable with parameter a, $\mathbb{P}[L = 0] = a$ and $\mathbb{P}[L = 1] = 1 - a$. The number of processors C that is added or removed from an allocation is computed as follows:

$$C = \begin{cases} -|X_1| - 1 & \text{if } L = 1 \\ |X_2| + 1 & \text{if } L = 0 \end{cases} \quad (1)$$

where $X_1 \sim \mathcal{N}(0, \sigma_1)$ and $X_2 \sim \mathcal{N}(0, \sigma_2)$.

Figure 3 shows an example of the probability density function of C with $\sigma_1 = \sigma_2 = 5$ and $a = 0.2$. The value $a = 0.2$ means that the number of processors allocated to a task decreases with a probability of 20%.

E. Complexity Analysis

Let us assume that EMTS evolves over U generations. The EA of EMTS takes t starting solutions as input. The complexity of producing the starting solutions depends on the algorithms that provide the allocation procedures. For example, the complexity of HCPA's allocation procedure is the same as for CPA, namely $O(V(V + E)P)$. The complexity of their mapping functions for V nodes and P processors is $O(E + V \log V + VP)$ [9]. Let C_{alloc} be the highest complexity of all allocation procedures and let C_{map} be the complexity of the mapping procedure. Thus, the complexity to generate the starting solutions is $O(t \cdot (C_{alloc} + C_{map}))$.

In each generation, we select μ parent individuals and generate λ offspring from the selected parent set. Hence, the

overall complexity of optimizing the processor allocations using the EA is $O(U \cdot \mu \cdot \lambda \cdot C_{map})$. Altogether, the complexity of EMTS is $O(t \cdot (C_{alloc} + C_{map}) + U \cdot \mu \cdot \lambda \cdot C_{map})$.

IV. Experimental setup

We implemented a simulator to evaluate EMTS. The simulator reads a platform file, containing the processors' speed, and builds a platform model according to Section II-A. In addition, the simulator reads the description of the PTG and executes the scheduling algorithm. Every task of the PTG has associated costs, measured in number of floating point operations (FLOP). Each task has an execution time function assigned to it, which returns the run time of that task for a given number of processors. Data communication costs between tasks are not considered in the simulation. Furthermore, it is assumed that a processor only executes one task at a time.

A. Platforms

For the experiments, we have used the platform models of two clusters of Grid'5000. These clusters have been chosen for two reasons: First, they have been used in previous articles [12], [13] and therefore, the results of the scheduling algorithms are better comparable. Second, both are production systems that could potentially run the parallel workflows. The smaller cluster named *Chti* is located in Lille and comprises 20 computational nodes with a computing speed of 4.3 GFLOPS. The other, called *Grelon*, is located in Nancy and composed of 120 nodes, each of computing speed of 3.1 GFLOPS. The peak performances have been measured with the HP-LinPACK benchmark using the AMD Core Math Library (ACML).

B. Execution Time Models

We have conducted the experiments with two execution time models of parallel tasks. The first model uses Amdahl's law and gives us insight of how EMTS performs compared to algorithms that were originally designed for a monotonically decreasing model. The second model is synthetic and applies a non-monotonically decreasing function that imitates the execution time characteristics shown in Figure 1. For both models it is assumed that the processor speed (GFLOPS) and the number of operations per task (FLOP) are known.

Model 1 – Amdahl's Law: Let $T(v, 1)$ be the sequential execution time of task v. Amdahl's law states that the parallel execution time for p processors is bounded by the amount of code that is non-parallelizable. This fraction of code is denoted by parameter α, $0 \leq \alpha \leq 1$. Hence, the overall execution time of a task using p processors can be estimated as: $T(v,p) = (\alpha + \frac{1-\alpha}{p}) \cdot T(v, 1)$.

To predict the execution time in the simulator, each node of a PTG must define its own value for the parameter α. If two nodes have different values of α, their performance models differ as well.

Algorithm 1 Parallel Execution Time of *Model 2*

```
T(v,p) = apply Model 1
if p > 1 then
    if p%2 == 1 then
        T(v,p) = 1.3 · T(v,p)
    else
        if √p is an integer then
            T(v,p) = 1.1 · T(v,p)
```

Model 2 – Synthetic Model: This model is based on *Model 1* but slightly increases the execution time of a parallel task if the number of processors is not a multiple of 2 or if this number has no integer square root. The pseudo code of this model is presented in Algorithm 1. The model emulates the run time characteristics of PDGEMM, which were shown in Figure 1. We note that the timings recorded on a Cray were not directly transferred to our Grid'5000 clusters.

C. Applications

Selecting PTGs for comparing the performance of scheduling algorithms is difficult. The PTGs should be free of any bias that would favor a particular scheduling algorithm. As it is often better to reuse PTGs than generating a new set that has to be justified, we decided to base our evaluation on PTGs that were used in previous articles.

Choosing Task Complexities: It is assumed that a task operates on a dataset of d doubles (8 bytes), e.g., a $\sqrt{d} \times \sqrt{d}$ matrix. We also assume that all processors have at least 1 GB of memory. So, the upper bound for d is $125E6$. The number of FLOP per task follows one of three computational patterns[2]: (1) $a \cdot d$, (2) $a \cdot d \log d$ and (3) $d^{3/2}$. The parameter a is randomly chosen between 2^6 and 2^9 to model multiple iterations of these tasks. In order to determine the execution time of a parallel task, the scheduling algorithm needs information about the α value (fraction of non-parallelizable code) of a task. Thus, a random number α is picked uniformly between 0 and 0.25, which leads to very scalable tasks.

Application Task Graphs: We consider PTGs that can be extracted from the Strassen's matrix multiplication algorithm and from the Fast Fourier Transform (FFT) application. These PTGs are of regular shape with several layers of concurrent parallel tasks. For more information about the shape of these PTGs we refer the reader to Hall *et al.* [3] and Cormen *et al.* [23]. Depending on the depth level, the FFT PTG comprises a different number of tasks. We use FFT PTGs with 2, 4, 8, and 16 levels, which lead to 5, 15, 39, or 95 tasks respectively.

We use a DAG generator that takes the shape of a DAG (Strassen or FFT) and assigns each task a random data size d

[2](1) stencil computation, (2) sorting an array, or (3) multiplication of $\sqrt{d} \times \sqrt{d}$ matrices

and a random parallelization factor α. So, the generated PTGs for the FFT and the Strassen's matrix multiplication have the same shape but differ in their task complexities. We have created 400 FFT and 100 Strassen PTGs.

Synthetic PTGs: To evaluate EMTS over a broader range of PTGs we have also generated random PTGs with DAGGEN [24]. The random task graphs contain 20, 50, or 100 data-parallel tasks. Four parameters define the shape of the PTGs, which are: *width*, *regularity*, *density*, and *jumps*. The *width* defines the maximum task parallelism of the PTG. A small value leads to a chain of tasks and large values lead to fork-join graphs. The *regularity* denotes the uniformity of the number of tasks per level. The *density* parameter changes the number of edges between two levels of the PTG. The *jump* parameter controls if edges can span several precedence levels of the PTG. More information about the PTG generation process can be found in the literature [12], [13], [14], [24].

The following parameters were used to create the synthetic PTGs: *width*=$\{0.2, 0.5, 0.8\}$, *regularity*=$\{0.2, 0.8\}$, *density*=$\{0.2, 0.8\}$. For the parameter *jump* we used the values $\{1, 2, 4\}$ to generate *irregular* PTGs and $\{0\}$ for *layered* ones. In total, we have generated 108 layered and 324 irregular PTGs. *Layered PTGs* only contain edges between adjacent precedence levels (*jump*=0) and the number of operations of tasks in one layer is similar. Such restrictions do not exist for *irregular* PTGs.

V. Experimental results

Like many other evolutionary algorithms EMTS has several tunable parameters. However, as stated above, we are not primarily interested in finding the best parameters for each case. In fact, the main purpose of our experiments is to reveal whether an EA can tune given schedules in a short amount of time. Hence, we set the parameters to reasonable values. The parameter Δ defines the criticality level per precedence layer and has been set to $\Delta = 0.9$, meaning that tasks whose criticality is only 10% smaller than the maximum value are also considered critical. The parameter f_m defines the number of allocations that are initially changed by the mutation operation. We chose $f_m = 0.33$ so that 33% of an original individual are randomly changed in the first iteration.

Finally, we specify the common parameters that influence the evolutionary optimization process. We apply a $(\mu + \lambda)$-EA, where μ is the number of parents that are kept in the population while λ specifies the number of offspring generated per generation. We use a "Plus-Strategy" where the best individuals of a pool of parent and offspring individuals are selected. For such a setup, the best solution that has been found is definitely conserved and will exist in the next generation's population. Therefore, the population can never become worse while the generations proceed [25]. In detail, we perform our evaluation with a $(5 + 25)$-EA and a $(10 + 100)$-EA, denoted as EMTS5 and EMTS10, respectively. We run EMTS5 for a relatively small number of 5 and EMTS10 for 10 generations.

A. Performance with Model 1

The first experiment should reveal whether EMTS can optimize the schedule of other heuristics when the execution time model is monotonically decreasing. Figure 4 presents the average relative makespan obtained with EMTS5 compared to the schedules produced by HCPA and MCPA (e.g., for MCPA: $T_{\text{MCPA}}/T_{\text{EMTS5}}$). The results are shown for each class of PTGs (FFT, Strassen, layered, irregular) for both the *Chti* (two left bars) and the *Grelon* cluster (two right bars). For regular PTGs (FFT, Strassen, layered), EMTS only slightly improves the schedules produced by MCPA. That is not surprising as MCPA takes special care of regularly shaped PTGs and attempts to exploit maximum task parallelism. Nevertheless, EMTS improves the makespan significantly when compared to HCPA or if irregular PTGs are scheduled. We can also observe that EMTS performs comparatively better for larger platforms (*Grelon*). This is obvious as the probability of finding a better allocation increases when the size of the platform increases. So, we can state that EMTS can be applied as meta-heuristic to optimize schedules if parallel tasks exhibit a monotonically decreasing execution time model.

B. Performance with Model 2

Figure 5 presents the experimental results where the non-monotonically decreasing model *Model 2* has been applied to predict the run time of parallel tasks. The upper row shows the results for EMTS5 and the lower row the results for EMTS10. We can see that EMTS can reduce the makespan more if the system is comprised of more processors (*Grelon*). We point out that when applying *Model 2*, the allocation routine of MCPA or HCPA does not stop with 1-processor allocations. Often allocations will grow up to a size of 4–8 processors before the allocation procedure stops. An allocation between 4 and 8 processors does not lead to huge performance differences for the smaller cluster (*Chti*) that comprises 20 processors. That explains the outcome for the smaller platform and helps to clarify the results for the larger platform (the two right bars in each plot), where EMTS5 significantly reduces the makespan in all cases.

The scheduling performance improves if more individuals are created and tested. This can be seen in the bottom half of Figure 5 as EMTS10 shows superior results over EMTS5. Nonetheless, the performance of EMTS10 is just slightly better than the one of EMTS5 for regular PTGs (FFT, Strassen, layered) on the larger cluster (*Grelon*). There are two reasons for that. First, the random generator uses the same (random) seed for all experiments. Thus, the solutions found by EMTS5 will also be found by EMTS10. Second, the schedule created by EMTS5 is already efficient, so that

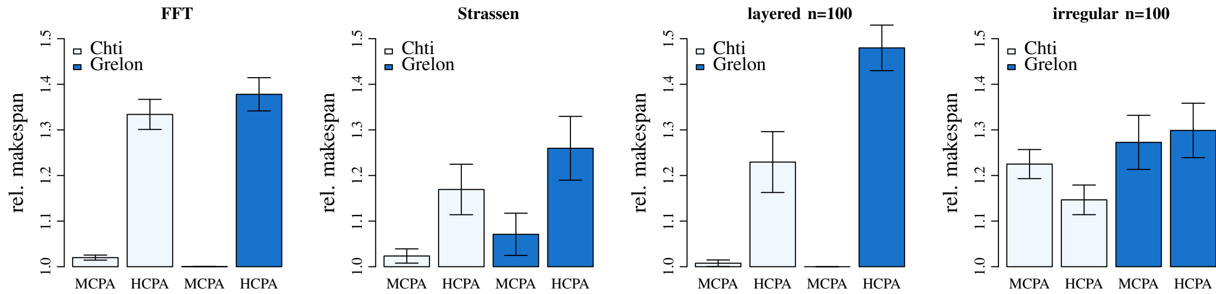


Figure 4. Average relative makespan of MCPA and HCPA compared to EMTS5 for different types of PTGs on *Chti* and *Grelon* (with 95% confidence intervals), ***Model 1***.

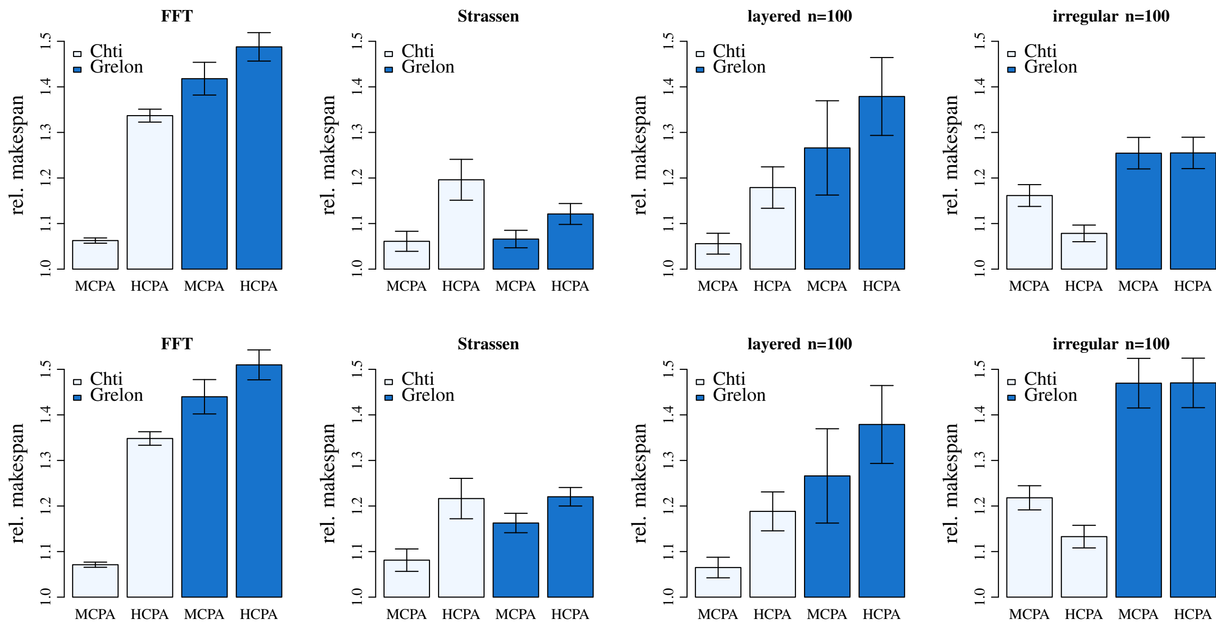
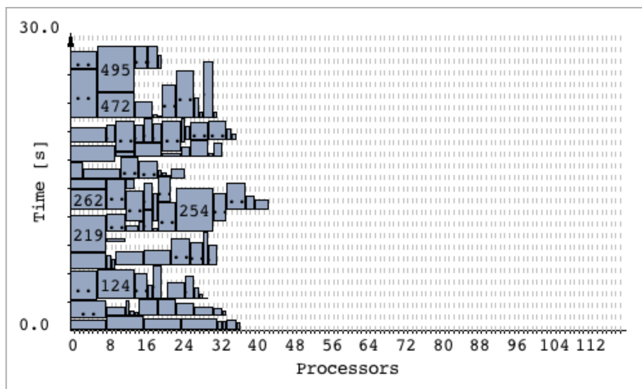


Figure 5. Average relative makespan of MCPA and HCPA compared to EMTS5 (top) and EMTS10 (bottom) for different types of PTGs on *Chti* and *Grelon* (with 95% confidence intervals), ***Model 2***.

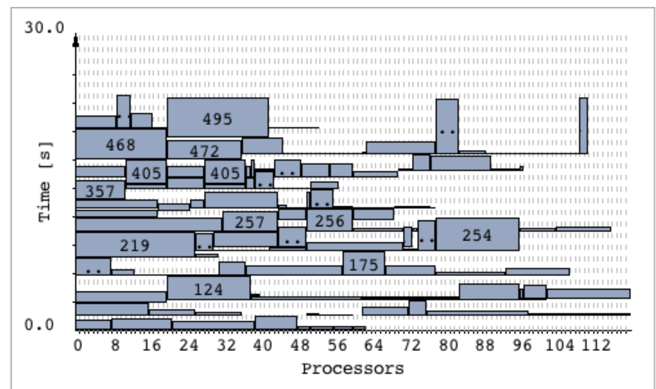
improving this solution would require many more evolutionary generations. In the synthetic case of irregular PTGs, EMTS10 improves the scheduler's performance significantly as there is a higher probability that modifying an allocation leads to a shorter schedule than for regularly shaped PTGs.

Figure 6 shows a side-by-side comparison of schedules obtained from MCPA and EMTS10 for a PTG with 100 nodes on *Grelon* using *Model 2*. The details of the graphics (e.g., task identifiers) are not important to understand the overall statement of this visualization. We can see in the left graphic that the processor allocations produced by MCPA are very small, which leads to a poor resource utilization. In contrast to this, EMTS finds a shorter schedule by stretching the big tasks from the left across multiple processors. Thus, EMTS utilizes the cluster resources more efficiently.

One question still remains unanswered: How much time is spent on optimizing the schedules with EMTS? The run time of EMTS5 varies between 0.45 s (SD 0.01 s) for smaller PTGs (Strassen) and 2.7 s (SD 1.1 s) for larger PTGs (100 nodes). These timings are mean values obtained for the *Chti* platform model on an Intel Core i5 (2.53 GHz). For the larger platform *Grelon*, the optimization with EMTS5 requires between 1.3 s (SD 0.07 s) and 5.5 s (SD 1.7 s) on average. The mean execution time of EMTS10 on *Grelon* is between 9.6 s (SD 0.5 s) and 38.1 s (SD 9.5 s). Considering these run times, one can say that EMTS5 could be applied in practice. However, EMTS10 should be applied when scheduling larger PTGs to find a good solution. The current prototype is written in Python. So, we can expect a reduction of the run time by a factor of 10 for an optimized C program.



MCPA



EMTS

Figure 6. Example schedules produced by MCPA and EMTS10 for an irregular PTG with 100 nodes on *Grelon*.

VI. Conclusions

In the present paper, we have proposed a hybrid method for scheduling PTGs onto homogeneous clusters. EMTS optimizes the processor allocation of each parallel task to minimize the makespan of the schedule. A major advantage of EMTS is its independence of the execution time model used to predict the run time of parallel moldable tasks. EMTS applies an evolutionary algorithm to obtain a good solution to the scheduling problem. EMTS encodes scheduling results produced by heuristics such as MCPA and HCPA as individuals of the EA, which significantly reduces the time to find efficient schedules. The experimental results show that EMTS significantly reduces the makespan of the schedules for the cases considered in this work.

Several extensions of EMTS could be part of future work. It is our main objective to reduce the time spent for the evolutionary search. In this context, different evolutionary methods could be compared to each other with respect to scheduling performance and speed. The execution time of the EA is mainly determined by the mapping function as it evaluates the fitness of individuals. Since the list scheduling algorithm can hardly be reduced in complexity, it would be beneficial to design heuristics that reject solutions if the current schedule does not meet certain conditions while the algorithm is still in the mapping phase. With such a rejection strategy, the construction of the whole schedule for inefficient solutions could be avoided.

Acknowledgments

We would like to thank NERSC of the Lawrence Berkeley National Laboratory for providing access to their Cray XT4. This work is partially supported by the ANR project USS SimGrid (08-ANR-SEGI-022).

References

[1] W. Cirne and F. Berman, “A Model for Moldable Supercomputer Jobs,” in *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS’01)*. IEEE Computer Society, 2001.

[2] J. Leung, L. Kelly, and J. H. Anderson, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Boca Raton, FL, USA: CRC Press, Inc., 2004.

[3] R. Hall, A. L. Rosenberg, and A. Venkataramani, “A Comparison of Dag-Scheduling Strategies for Internet-Based Computing,” in *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS’07)*, 2007, pp. 1–9.

[4] L. S. Blackford *et al.*, *ScaLAPACK Users’ Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997.

[5] N. Vydyanathan, S. Krishnamoorthy, G. M. Sabin, Ü. V. Çatalyürek, T. M. Kurç, P. Sadayappan, and J. H. Saltz, “An Integrated Approach to Locality-Conscious Processor Allocation and Scheduling of Mixed-Parallel Applications,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 8, pp. 1158–1172, 2009.

[6] V. Boudet, F. Desprez, and F. Suter, “One-Step Algorithm for Mixed Data and Task Parallel Scheduling without Data Replication,” in *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS’03)*, 2003, p. 41.

[7] S. Ramaswamy, S. Sapatnekar, and P. Banerjee, “A Framework for Exploiting Task and Data Parallelism on Distributed Memory Multicomputers,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, pp. 1098–1116, November 1997.

[8] A. Rădulescu, C. Nicolescu, A. J. C. van Gemund, and P. Jonker, “CPR: Mixed Task and Data Parallel Scheduling for Distributed Systems.” in *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS’01)*. IEEE Computer Society, 2001, p. 39.

[9] A. Rădulescu and A. J. C. van Gemund, "A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling," in *Proceedings of the 2001 International Conference on Parallel Processing (ICPP '02)*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 69–76.

[10] T. N'Takpé and F. Suter, "Critical Path and Area Based Scheduling of Parallel Task Graphs on Heterogeneous Platforms," in *Proceedings of the 12th International Conference on Parallel and Distributed Systems (ICPADS 2006)*, 2006, pp. 3–10.

[11] S. Bansal, P. Kumar, and K. Singh, "An Improved Two-Step Algorithm for Task and Data Parallel Scheduling in Distributed Memory Machines," *Parallel Computing*, vol. 32, no. 10, pp. 759–774, 2006.

[12] S. Hunold, "Low-Cost Tuning of Two-Step Algorithms for Scheduling Mixed-Parallel Applications onto Homogeneous Clusters," in *Proceedings of the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2010)*, 2010.

[13] S. Hunold, T. Rauber, and F. Suter, "Redistribution Aware Two-Step Scheduling for Mixed-Parallel Applications," in *Proceedings of the 10th IEEE International Conference on Cluster Comp.*, 2008.

[14] F. Desprez and F. Suter, "A Bi-criteria Algorithm for Scheduling Parallel Task Graphs on Clusters," in *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid 2010)*, 2010, pp. 243–252.

[15] A. B. Downey, "A Model for Speedup of Parallel Programs," U.C. Berkeley, Tech. Rep. CSD-97-933, 1997.

[16] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the Spring Joint Computer Conference*. ACM, 1967, pp. 483–485.

[17] E. Günther, F. König, and N. Megow, "Scheduling and Packing Malleable Tasks with Precedence Constraints of Bounded Width," in *Approximation and Online Algorithms*, ser. Lecture Notes in Computer Science, E. Bampis and K. Jansen, Eds. Springer Berlin / Heidelberg, 2010, vol. 5893, pp. 170–181.

[18] W. Pfeiffer and N. Wright, "Modeling and Predicting Application Performance on Parallel Computers Using HPC Challenge Benchmarks," in *Proceedings of the 22nd International Parallel and Distributed Processing Symposium (IPDPS'08)*, 2008, pp. 1–12.

[19] A. S. Wu, H. Yu, S. Jin, K. chi Lin, and G. Schiavone, "An Incremental Genetic Algorithm Approach to Multiprocessor Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 2, pp. 824–834, 2004.

[20] C. Franke, J. Lepping, and U. Schwiegelshohn, "Greedy Scheduling with Custom-made Objectives," *Annals of Operations Research*, vol. 180, no. 1, pp. 145–164, November 2010.

[21] F. Suter, "Scheduling Δ-Critical Tasks in Mixed-Parallel Applications on a National Grid," in *Proceedings of the 8th IEEE/ACM International Conference on Grid Comp. (GRID 2007)*. IEEE, 2007, pp. 2–9.

[22] F. Rothlauf, *Representations for Genetic and Evolutionary Algorithms*, 2nd ed. Springer-Verlag, 2006.

[23] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[24] DAG Generation Program, http://www.loria.fr/~suter/dags.html.

[25] H.-P. Schwefel and G. Rudolph, "Contemporary Evolution Strategies," in *Advances in Artificial Life – Proceedings of the Third European Conference on Artificial Life (ECAL'95)*, F. Morán, A. Moreno, J. J.Merelo, and P. Chacón, Eds. Berlin: Springer, 1995, pp. 893–907.