Comparison of Static and Dynamic Resource Allocation Strategies for Matrix Multiplication

Olivier Beaumont, Lionel Eyraud-Dubois, Abdou Guermouche, Thomas Lambert

Comparison of Static and Dynamic Resource Allocation Strategies for Matrix Multiplication

Olivier Beaumont
Lionel Eyraud-Dubois
Abdou Guermouche
Thomas Lambert
Inria and University of Bordeaux – Talence, France
olivier.beaumont@inria.fr, lionel.eyraud-dubois@inria.fr
abdou.guermouche@labri.fr,thomas.lambert@inria.fr

***Abstract*—The tremendous increase in the size and heterogeneity of supercomputers makes it very difficult to predict the performance of a scheduling algorithm. In this context, relying on purely static scheduling and resource allocation strategies, that make scheduling and allocation decisions based on the dependency graph and the platform description, is expected to lead to large and unpredictable makespans whenever the behavior of the platform does not match the predictions. For this reason, the common practice in most runtime libraries is to rely on purely dynamic scheduling strategies, that make short-sighted scheduling decisions at runtime based on the estimations of the duration of the different tasks on the different available resources and on the state of the machine. In this paper, we consider the special case of Matrix Multiplication, for which a number of static allocation algorithms to minimize the amount of communications have been proposed. Through a set of extensive simulations, we analyze the behavior of static, dynamic, and hybrid strategies, and we assess the possible benefits of introducing more static knowledge and allocation decisions in runtime libraries.**

Keywords: Scheduling; Resource Allocation; Dynamic Scheduling Strategies; Independent Tasks Scheduling Sharing Input Files; Outer Product;

I. Introduction

In this paper, we compare the use of static, dynamic and hybrid scheduling strategies for modern heterogeneous platforms.

In static strategies, scheduling and resource allocation decisions are made based on the description of the dependency graph of the application and on the description of the execution platform. This includes the execution times of all types of tasks on all types of resources, the communication times between any two resources and the congestion between any set of communications.

In dynamic strategies, scheduling and resource allocation decisions are made at runtime based on the state of the platform (which computing and communication resources are available) and the set of available tasks (whose dependencies have all been resolved) and the current location of input data. Dynamic strategies may be either task driven (allocate a task to a resource as soon as a task becomes ready) or resource driven (allocate a task to a resource as soon as a resource becomes idle). Of course, in both cases, sophisticated mechanisms in order to overlap communications with computations and to determine the relative priorities among ready tasks are added.

At last, hybrid strategies start with a static allocation strategy together with a dynamic strategy to adapt to changes in timing predictions, that may come either from bad predictions, concurrent applications running on the platform or even resource failures. Basically, hybrid strategies come altogether with an initial static mapping and a policy to cope with non-determinism of both processing and communication times.

In general, computing the optimal static schedule is known to be very difficult and greedy strategies may work poorly. The scheduling literature abounds in NP-Completeness [1] and inapproximability results [2]. Therefore, many static strategies rely on list-based techniques. For instance, HEFT [3] defines for each task its priority as its upward rank, which is defined as the estimated time before the scheduling time of the task and the execution time of the last task, based on average values of processing and communication times. If all communication and processing times are precisely known, a dynamic strategy can be seen as a basic greedy static strategy. On the other hand, dynamic schedulers can benefit from static information such as HEFT priorities, in order to decide which task should be scheduled first when several tasks are available.

On the other hand, many practical problems exhibit a specific structure of dependencies that make the scheduling problem easier. An extreme application model in term of dependencies is that of independent tasks with no task synchronizations and no inter-task communications. In this case, the scheduling problems are akin to off-line and on-line bin-packing and a number of heuristics have been proposed in the literature (see [4], [5] for surveys in off-line and on-line contexts).

Besides the intrinsic difficulty of the optimization problem, static schedulers face another problem. Indeed, recently, there has been a very important change in the scale of parallel platforms, and both Cloud and Supercomputers involve more and more computing resources. This scale change poses many problems related to unpredictability and failures. In this context, relying on purely static solutions is very questionable, even for basic and regular task graphs. Indeed, a single failure

of a resource may lead to arbitrarily long execution times, and replication must be used in order to cope with failures.

In this paper, following [6], [7] where the simpler context of static and dynamic strategies for the outer product computation is considered, we concentrate on the design and analysis of static, dynamic and hybrid for an intermediate settings, namely Matrix Multiplication $C = C + AB$. Besides its practical interest, this application is of particular interest in our context. Indeed, it can be written as a sequence of phases of independent tasks, but where those tasks share input files.

This context is of particular interest in order to compare static and dynamic strategies. Indeed, on the one hand, the simple structure of the application makes it amenable to both analytical study and simulations. Indeed, even obtaining the analysis through simulations of the performance of an allocation strategy under failures and unpredictable execution times described as probability distributions is hard due to the huge space of parameters. On the other hand, the results presented in [6], [7] in the simpler context of the outer product show that it is very difficult to design dynamic strategies that take data reuse efficiently into account, whereas it is possible to design efficient static distribution strategies. This makes the problem of designing and analyzing the behavior of hybrid strategies crucial. In sharp contrast with previous works [6], [7], we consider a more general and realistic model, where processor speeds are not only unknown in advance but can vary over time. Under this model, it is of interest to consider hybrid strategies, that take advantage of (possibly erroneous) information on the resource performance to build static clever allocation scheme and that can modify allocation decisions at runtime based on the state of both the system and the application. We will evaluate the behavior of all the considered strategies in the context of a simulated environment where experimental scenario can be fully reproducible.

This paper is organized as follows. Section II states the problem formally and presents the experimental setting which is used throughout the paper. Section III gives a review of related works. In Section IV we present the different algorithms we study in this paper, the static ones in Section IV-A and the dynamic and hybrid ones in Section IV-C. Finally Section VI gives concluding remarks.

II. Problem Statement and Notations

A. Targeted Application

In the case of independent tasks working on independent data, what corresponds to the map phase of Hadoop [8], replication strategies have been designed so as to achieve a good makespan while not forcing the multiple execution of too many tasks. Our aim in this paper is to consider the more difficult problem of applications sharing input data. In this context, the task allocation decision should be made not only based on the (estimated) processing speed of the resources but also on the location of input data, in order to avoid performing useless communications. Therefore, we concentrate on a simple though widely used kernel arising from Linear Algebra, Matrix Multiplication, where the extra difficulties we aim at studying take place. More specifically, we consider the problem of computing the product of two large matrices A and B. In order to have a good efficiency, we will assume that A and B are partitionned into N^2 blocks whose size is the same for all processing resources. Therefore, computing the matrix multiplication correspond to the computation of N^3 elementary tasks, each corresponding to an elementary matrix product of blocks $C_{i,j,k} = A_{i,k}B_{k,j},\ \forall\ 1 \leq i,j,k \leq N$. In order to compute the result matrix C, all contributions $C_{i,j,k},\ \forall\ 1 \leq k \leq N$ must be summed up. In order to avoid keeping simultaneously too many partial distributed contributions of $C_{i,j}$, we choose to implement the matrix multiplication as a sequence of synchronized Outer Products, as in the Cannon's Algorithm implemented in Scalapack [9]. Note that recently, so-called 2.5D implementations of the matrix product have been proposed [10], where several copies are kept at the same time. Since our goal is to study the impact of the scheduling decisions on the overall volume of communications, we will concentrate on Cannon's-like implementations.

As stated above, we target a heterogeneous computing platform with p computing resources labeled $P_1, P_2, \ldots, P_p$ and we denote by w_l the time necessary to process an elementary block matrix product on P_l. For the sake of simplicity and in order to stick with more traditional scheduling models and notations, we will assume that there exists a master node P_0 that sends out chunks to workers over a network. In practice, data initially resides in the memory and will be sent from there to the different processing resources. Therefore, main memory will act as the master node. In order to concentrate on the difficulties introduced by non-determinism of execution times, we will consider a model for communications where communications can be overlapped with computations, what is a reasonable assumption in the case of dense blocked matrix product. Thus, communication time will not be explicitly taken into account in the makespan, but the overall number of elements sent from the main memory to the processing units and between the processing units themselves will be carefully evaluated, and can be seen either as a measure of possible congestion or as a measure of communication energy.

B. Targeted Platforms

For the sake of realism, we will analyze the behavior of our algorithms on 4 different target platforms.

The two first platforms, which will denoted by HOMOGENEOUS-5-CPUS and HOMOGENEOUS-20-CPUS, are homogeneous platform consisting of respectively 5 and 20 CPUs. Note that due to non-determinism of processing times, these platforms will turn out to be heterogeneous in practice, but it corresponds well to the homogeneous machines that can be found either in multicore HPC systems or in datacenters.

In addition, we consider the HETEROGENEOUS-1-GPU and theHETEROGENEOUS-4-GPUS heterogeneous platforms, that correspond well to machines consisting of a few accelerators such as GPU units and a few multicore nodes. More specifically, HETEROGENEOUS-1-GPU consists in 1 GPU and 4

Name	Distribution	Parameters
UNIFORM-0.80	Uniform in $[a, b]$	$a = 0.8, b = 1.2$
UNIFORM-0.95		$a = 0.95, b = 1.05$
GAUSSIAN-0.1	Truncated Gaussian $\max(\mathcal{N}(\mu, \sigma^2), 0)$	$\mu = 1, \sigma = 0.1$
GAUSSIAN-0.1		$\mu \simeq 0.97, \sigma = 0.5$
GAUSSIAN-0.1		$\mu \simeq 0.48, \sigma = 1$
TWOMODES-2	v_1 with prob. 0.99 v_2 with prob. 0.01	$v_1 = \frac{1}{1.01}, v_2 = \frac{2}{1.01}$
TWOMODES-10		$v_1 = \frac{1}{1.09}, v_2 = \frac{10}{1.09}$

Table I
PROBABILITY DISTRIBUTIONS FOR EXECUTION TIMES. THE PROCESSING TIME OF A TASK PROCESSED BY P_l IS Xw_l, WHERE X IS DRAWN WITH THE CORRESPONDING DISTRIBUTION. ALL DISTRIBUTIONS HAVE AN EXPECTED VALUE OF 1.

CPUs and HETEROGENEOUS-4-GPUS consists in 4 GPUs and 16 CPUs. Timings used for the relative performance of GPUs and CPUs typically correspond to what can be achieved with regular matrix product operations on both devices: a GPU is almost 50 times faster than a regular CPU core.

C. Resource Performance

As already stated, our goal is also to model non-determinism in resource performance. More precisely, the processing time of the $m-$th task on resource P_l will be given by a random function depending on a specific probability distribution. In this paper, we will consider three classes of probability distributions. In order to do a fair comparison of makespan and communication, we fix a value w_{CPU} which is the expected processing time for the CPUs for all the probability distributions studied here. The expected processing time of the GPUs is set to $50w_{CPU}$. Moreover, for all probability distributions described in Table I, parameters are fixed such that the expected duration of a task on a CPU is w_{CPU} (resp. $50w_{CPU}$ on a GPU). Throughout the whole set of simulations, the estimations are obtained with the same algorithm. Each resource processes 5 tasks, whose execution times are chosen at random according to the relevant distribution, as presented on Table I. Then, the estimated processing time will be taken as the mean value of these 5 measurements. This typically corresponds to classical algorithms used in runtime systems.

D. Problem Statement, Bounds and Displayed Values

In order to compute $C_{i,j} \mathrel{+}= A_{i,k} \times B_{k,j}$ a processor has to own in its local memory both $A_{i,k}$, $B_{k,j}$ and $C_{i,j}$ (remember that we concentrate on Matrix Multiplication Algorithms where all contributions $A_{i,k} \times B_{k,j}$ are aggregated into a single copy of $C_{i,j}$, as in the case of Cannon's Algorithm). For processor l we denote as $A_{i,k,l}$, $B_{k,j,l}$ and $C_{i,j,l}$ respectively the set of blocks of A, B and C that have been loaded in the local memory of P_l over the course of the computation. If we denote as V_l the volume of communications for processor P_l, then $V_l = |A_{i,k,l}| + |B_{k,j,l}| + |C_{i,j,l}|$ and $V = \sum_l V_l$ denotes the overall amount of communications. We also denote as W_l the set of tasks (*i.e.* the elementary operations $C_{i,j} \mathrel{+}= A_{i,k} \times B_{k,j}$) assigned to P_l. Finally, let us denote by T_l the processing time for each processor, $T_l = |W_l|/s_l$ and by T the makespan, $T = \max_l T_l$. Our goal is to compute a partition $W_1, \ldots, W_p$ of the N^3 elementary tasks which achieves the optimal makespan ($T = T_{opt}$) and performs as few communications as possible (minimize V).

As far as makespan is concerned, remember that for all probability distributions, the expected duration of a task executed on P_l is w_l. Therefore, the expected makespan is given by NT_{exp}, where T_{exp} is the smallest value such that $\sum_k \lfloor \frac{T_{exp}}{w_l} \rfloor \geq N^2$. In what follows, all makespan values will be normalized using this expression of T_{exp}. Note that T_{exp} is the expected value of the optimal makespan only, so that it is possible to have ratios smaller than 1 (if by chance processing speeds are higher than expected and compensate the non-optimality of the algorithm). Nevertheless, it enables to compare fairly the heuristics between them.

As far as communications are concerned, let us consider the $k-th$ phase of the algorithm where all tasks $C_{i,j} \mathrel{+}= A_{i,k} \times B_{k,j}$ are performed. During this phase, processor P_l processes a set of elementary products $W_{k,l}$, $C_{i,j} \mathrel{+}= A_{i,k} \times B_{k,j}$ for some values of i and j. This requires the knowledge of the corresponding blocks of $A_{i,k}$ and $B_{k,j}$ and therefore $W_{l,k} \subseteq A_{i,k,l} \times B_{k,j,l}$ and then $|W_{l,k}| \leq |A_{i,k,l}||B_{k,j,l}|$. Furthermore, the Loomis-Whitney inequality says that the product $A_{i,k,l} \times B_{k,j,l}$ is maximized for $|A_{i,k,l}| = |B_{k,j,l}|$. Therefore, the following lower bound holds true for communication:

$$V_{l,k} \geq 2N \sqrt{s_l / \sum_{l'} s_{l'}}$$

where $V_{l,k}$ denotes the number of elementary blocks of A and B that need to be sent to P_l during phase k. Finally, since P_l has to store the corresponding values of C and since communications are minimized when P_l always stores the same values of C, then

$$\sum_l V_l \geq C_{exp} = N^2 + 2N^2 \sum_l \sqrt{s_l / \sum_{l'} s_{l'}}.$$

In what follows, all communication volume values will be normalized using above expression of C_{exp}. Note again that C_{exp} is the expected value of the a lower bound on the communications, so that it is possible to have ratios smaller than 1.

III. RELATED WORKS

A successful approach to deal with the complexity of modern architecture is centered around the use of runtime systems to manage tasks dynamically, these runtime systems being either generic or specific to the application. Among other successful projects, we may cite KAAPI/XKAAPI [11], PaRSEC [12], SMPSs [13], or StarPU [14]. As a result, higher performance portability is achieved thanks to the hardware abstraction layer introduced by runtime systems [15]. More recently, this approach has been used for more irregular applications. Sparse direct solvers have been redesigned on top of task-based runtime systems [16] leading to a good behavior and an improved portability. From the task scheduling point of view, most of these task-based runtime systems rely

on dynamic strategies for task scheduling. These dynamic scheduling heuristics can be categorized in two families of strategies : *resource centric* and *task centric*. In the first set, a scheduling decision is taken when the queue corresponding to a computing resource is getting empty : a task is then selected either from the set of ready tasks or stolen from other resources. A notable heuristic is the work-stealing strategy [17] where the resource selects a victim to steal from. The stealing criterion may be driven by locality [18] to reduce data movements. These strategies are used in runtime systems like PaRSEC, SMPSs or KAAPI. On the other hand, task centric dynamic strategies take scheduling decisions each time a task is ready to be executed. An example of such strategies is the *Minimum Completion Time* (MCT) [3] where the ready task is assigned to the resource which minimizes the task's finishing time. It is based on performance models for both computations and data. MCT is one of the schedulers provided by the StarPU runtime system. Our goal in this paper in to provide an analysis of such dynamic and hybrid schedulers for simple operations, that do not involve task dependencies but massive data reuse.

Dynamic scheduling strategies have also been studied in a number of other contexts. In particular, in desktop Grids and volunteer computing scenarios, where the unpredictability of resource performance is a crucial issue, several studies [19] have analyzed the benefits and costs of replication when scheduling identical independent tasks, to avoid waiting for some very slow tasks at the end of the schedule. When estimates for task running times are available, Oprescu et al [20] have proposed a more efficient replication strategy based on replicating the task with the largest remaining execution time. Hybrid scheduling techniques have also been proposed and analyzed experimentally for the much more challenging problem of scheduling with precedence constraints [21]. In this paper, we address an intermediate setting, where tasks are independent, but share input data, and we analyze both makespan and communication performance. More recently, a study comparing different schedulers have been carried out in the context of dense linear algebra factorizations on heterogeneous systems [22]. Although, this study is closely related to the work we present in the present paper, it doesn't tackle neither the matrix product, nor the static (resp. hybrid) allocation which is considered in our work.

On the specific issue of analyzing the amount of communication needed to perform linear algebra kernels such as matrix-vector or matrix-matrix multiplication, generic lower bounds have been proposed [23] which provide insights about the lowest possible amount of communications to perform these tasks, when the local memory is limited. Static algorithms achieving these bounds have also been designed. Our work is focused on analyzing dynamic and hybrid approaches, to achieve minimal communications while providing robust implementations.

IV. Algorithm description

A. Static algorithms

The goal of the static algorithm we present here is to balance the computation tasks between the different processors in order to reach the optimal makespan while minimizing the amount of communications. Before the first outer-product, we run a static heuristic that partitions the taks between the processors, thanks to an estimation of their speeds. This partition will be used for later phases. Let us assume that areas allocated to the different processors are rectangles, *i.e.* $W_k = m_{k,line} \times m_{k,row}$. Then, the volume of communications induced by P_k, $C_k = |m_{k,line}| + |m_{k,row}|$ corresponds to the half-perimeter of the rectangle W_k and the processing cost is proportional to the area of W_k and is given by $w_k(|m_{k,line}| \times |m_{k,row}|)$. We obtain the total amount of communications by $C = N \sum_k C_k + N^2$ (the factor N comes from the N phases, and the N^2 term represents the cost of transfering the matrix C at the end of the computation).

Achieving perfect load balance is amenable to partition a square into rectangles of fixed area. This problem has been already studied in [24], [25], [26]. In [24], the ColumnBased algorithm is proposed. This algorithm, given a set of target values S_i summing up to 1, returns a partition of an unit size square into rectangles, each of area S_i, and aims to minimize the total perimeter of the rectangles. ColumnBased has been proven to be a $\frac{7}{4}$-approximation algorithm, and in practice, the approximation ratio is often below 1.1. Nagamochi et al. propose a different algorithm, namely DivideAndConquer, whose approximation ratio is $\frac{5}{4}$ and is as efficient in practice. A little variation on DivideAndConquer allows a approximation ratio of $\frac{2}{\sqrt{3}}$ when the areas of rectangles are not too unbalanced [26].

ColumnBased and DivideAndConquer can easily be turned into two first heuristics for our problem, by setting the areas to the relative speed of the processors. More specifically, StaticColumnBased (respectively StaticDivideAndConquer) uses the output of ColumnBased$(s_1/\sum s_k, \ldots, s_p/\sum s_k)$ (respectively DivideAndConquer$(s_1/\sum s_k, \ldots, s_p/\sum s_k)$) to partition the matrix of size N into p rectangles. In order to transform the partition of the unit size square into an integer block-based partition of the $N \times N$ square, we simply round values so as to build a partition into rectangles with integer lengths.

B. Rounding errors for small matrices

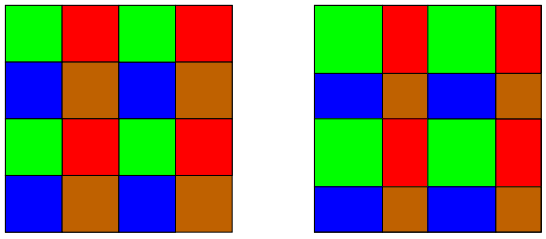
The approximation ratios mentioned above are particularly appealing for our problem. However, due to rounding errors, even in the case of known and constant over time processing speeds, StaticDivideAndConquer turns out to perform relatively poorly with respect to makespan minimization. Indeed, Table II provides the makespan and communication ratios in the case $N = 50$ for the different platforms (and constant processing times). The communication ratio is indeed good (much better, as expected, than the worst case bound

$\frac{5}{4}$), however the ratio for the makespan, that would be 1 if rounding was not used, is much worse than expected. In particular, on heterogeneous platforms, the makespan ratio can be as high as 1.38 for the platform HETEROGENEOUS-4-GPUS. Moreover, we can observe that the situation gets worse when new processors are added.

	StaticColumnBased		StaticDivideAndConquer	
	Makespan	Communication	Makespan	Communication
Homo-5	1.02	1.02	1.02	1.02
Homo-20	1.04	1.01	1.08	1.03
Hetero-1	1.02	1.04	1.12	1.06
Hetero-4	1.04	1.04	1.38	1.00

Table II
RATIO BETWEEN THE THE RESULT OF STATICCOLUMNBASED OR STATICDIVIDEANDCONQUER AND THE LOWER BOUNDS IN THE CASE OF CONSTANT AND WELL-ESTIMATED SPEEDS

The rationale behind these results is that COLUMNBASED and DIVIDEANDCONQUER are designed for the continuous case. On the other hand, the block size needs to represent a good trade-off between large granularity to fully exploit accelerators like GPUs and fine granularity to have good behavior on regular cores. Thus block sizes of order 1000 are required and the number of blocks is therefore relatively small: for our test platform, realistic problem sizes correspond to square matrices of several tens of thousands of order. Thus we consider rather small number of blocks ($N = 50$ which corresponds to a matrix size of 50000x50000), so that the effect of rounding errors is important. As an illustration, let us consider a platform with 16 identical processors (with speed 1) and a matrix of size 10. With this input, COLUMNBASED and DIVIDEANDCONQUER returns the same optimal partition, 16 squares each with a half perimeter of 2.5, see Figure 1(a). After rounding, in the integer partition (depicted in Figure 1(b)), some processors are assigned $3 \times 3 = 9$ tasks while other processors are only assigned $2 \times 2 = 4$ tasks. In this case, the rational optimal makespan would be $100/16 = 6.25$, but the makespan of the partition returned by STATICCOLUMNBASED and STATICDIVIDEANDCONQUER is 9.



(a) Partition from COLUMNBASED or DIVIDEANDCONQUER

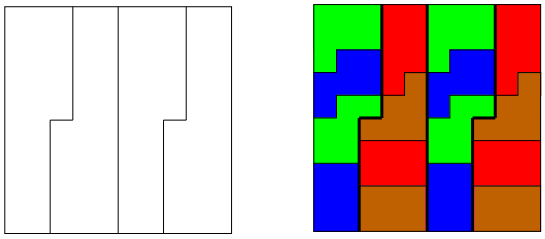
(b) Partition from STATICCOLUMNBASED or STATICDIVIDEANDCONQUER

Figure 1. Comparaison between the results of COLUMNBASED and STATICCOLUMNBASED for 16 identical processors and a matrix of size 10.

In order to deal with rounding errors, we have implemented two new heuristics STATICCOLUMNBASEDACCURATE and STATICDIVIDEANDCONQUERACCURATE that allow the assignment of non-rectangular areas, while remaining close to the partition provided by COLUMNBASED or DIVIDEANDCONQUER. This is achieved with the following procedure. By design, the output of COLUMNBASED is divided in a certain number n of columns, $c_1, \ldots, c_n$, and each processor is assigned to a certain column. The idea is to redefine the frontier between these columns so that each column has the correct area (what is not the case in Figure 1(b), where the first column has 30 tasks instead of 25). To do so, we go through the matrix column by column until the target number of tasks for this column is reached (see Figure 2(a)), and we later proceed similarly with rows (see Figure 2(b)) so that each cell contains exactly $\lceil \frac{s_k N^2}{\sum_{k'} s_{k'}} \rceil$ or $\lfloor \frac{s_k N^2}{\sum_{k'} s_{k'}} \rfloor$ tasks. STATICDIVIDEANDCONQUERACCURATE is designed along the same ideas and is illustrated on Figure 3.

	StaticColumnBased-Accurate		StaticDivideAndConquer-Accurate	
	Makespan	Communication	Makespan	Communication
Homo-5	1	1.03	1	1.03
Homo-20	1	1.04	1	1.05
Hetero-1	1.02	1.07	1.02	1.08
Hetero-4	1.04	1.12	1.04	1.05

Table III
RATIO BETWEEN THE THE RESULT OF STATICCOLUMNBASEDACCURATE AND STATICDIVIDEANDCONQUER AND THE LOWER BOUNDS IN THE CASE OF CONSTANT AND WELL-ESTIMATED SPEEDS



(a) Previous split for COLUMNBASED

(b) Partition from STATICCOLUMNBASEDACCURATE

Figure 2. Illustration of STATICCOLUMNBASEDACCURATE

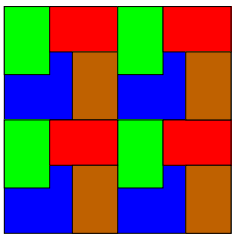


Figure 3. Illustration of STATICDIVIDEANDCONQUERACCURATE

Table III depicts the results achieved by STATICCOLUMNBASEDACCURATE and STATICDIVIDEANDCONQUER under the same conditions as in Figure 1. We can observe that STATICDIVIDEANDCONQUERACCURATE is more efficient in terms of makespan than STATICDIVIDEANDCONQUER and achieves similar results with respect to communications.

C. Dynamic Algorithms

In contrast to the static algorithms presented above, the common practice in task-based runtime systems is to rely on dynamic algorithms, which are naturally able to adapt their decisions to the actual behavior and to the current state of the resources. However, these strategies are often myopic, based solely on a short-term view.

As described in Section III, we can identify two main classes of greedy dynamic algorithms to allocate tasks on resources. In the *task-centric* view [14], the algorithm considers all tasks in some order, and for a given task greedily selects the most appropriate resource for this task. In the *resource-centric* approach [12], the algorithm considers resources when they are about to become idle, and greedily selects an appropriate task for this resource. In the following, we will consider two dynamic algorithms: MCT, a task-centric strategy based on the MCT (Minimum Completion Time) algorithm, and MINCOST, a resource-centric strategy which given a resource selects a random task among those which incur a minimal amount of communication, given the data already received by the resource.

Resource-centric algorithms can also be used as ingredients for hybrid algorithms, where a static allocation is performed at the beginning of the computation, and a greedy strategy is used whenever a resource has finished computing its statically allocated tasks. This is somewhat close to "work-stealing" strategies (see Section III). In this paper, we will consider several versions of this hybrid approach, depending on which static allocation is used at the beginning: COLUMNBASED gives rise to HYBRIDCOLUMNBASED and the accurate version HYBRIDCOLUMNBASEDACCURATE, and by using DIVIDEANDCONQUER we obtain HYBRIDDIVIDEANDCONQUER and HYBRIDDIVIDEANDCONQUERACCURATE.

Another natural feature for resource-centric algorithms is *replication*: at the end of the computation, when a resource becomes idle but no tasks are available, it can duplicate the execution of a task already started on another resource, in the hope that this will allow to finish the task earlier. Such strategies are quite rare in HPC scenarios, but have been extensively used in the Grid Computing community [19]. The consensus is that the most efficient approach is WQR, which replicates all running tasks equally up to a certain limit on the number of replications. In our experiments, only the fast resources are allowed to duplicate tasks, in order to make sure that most replications actually do improve the finish time of the task.

V. EXPERIMENTAL EVALUATION

In this section we present the results of our simulations of the different algorithms presented above. We present here the figures we judge the most relevant, but most exhaustive ones can be found on the Section VIII.In a first subsection we describe the case where the settings are completely static, i.e the speeds do not change during the execution. We present the dynamic setting in the second subsection. We recall that the two purely dynamic strategies are MCT (task-centric strategy) and MINCOST (resource-centric strategy) and 4 variants of static (resp. hybrd) algorithms. Finally, for each heuristic and platform, we perform 50 runs. In order to facilitate the comparison between the different strategies, the results have been normalized by the expected communication cost (resp. makespan) provided in Section II-C.

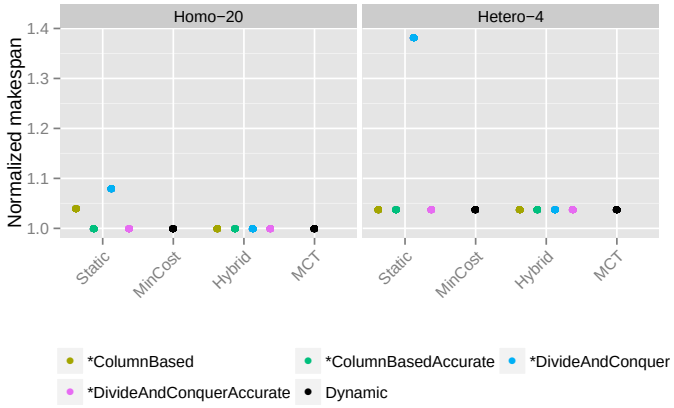


Figure 4. Makespan for different algorithms for HOMOGENEOUS-20-CPUS and HETEROGENEOUS-4-GPUS for static setting.

A. Static setting

When the performance of the resources is constant over time and well estimated, static heuristics are based on fully accurate previsions. In practice, one can notice on Figure 4 (or Figure 8) that, except STATICCOLUMNBASED and STATICDIVIDEANDCONQUER because of rounding errors, all algorithms are close to the optimal. The small difference for the heterogeneous platforms comes from the fact that some last tasks may be given to a cpu instead of gpu (which was busy at that time working on another task).

As for the communication costs, we can see in Figure 5 (or Figure 9) that static heuristics are better, as expected, and the "accurate" versions do not create a significant extra cost. At the same time, hybrid strategies perform well and the ratio with the optimal is always under 1.5. This good result can be explained by the fact that very few job stealing operations take place (except for HYBRIDCOLUMNBASED on HETEROGENEOUS-4-GPUS where the task balancing is quite odd, GPUs have a relatively unbalanced repartition of tasks, which does not degrade the makespan, but implies many jobs stealing). For purely dynamic strategies, the communication cost is larger and thus they have been excluded from Figure 5 (and Figure 9) to have a better view on the performance of hybrid strategies but hey can be seen on Figure 10 in the Appendix. For MINCOST, the ratio with the optimal value is close to 2 for heterogeneous platform and 2.5 for homogeneous ones. For MCT, it is larger than 9 for HOMOGENEOUS-20-CPUS.

B. Dynamic setting

In practice, performance are rarely constant over time and perfectly known, what explains the practical success of

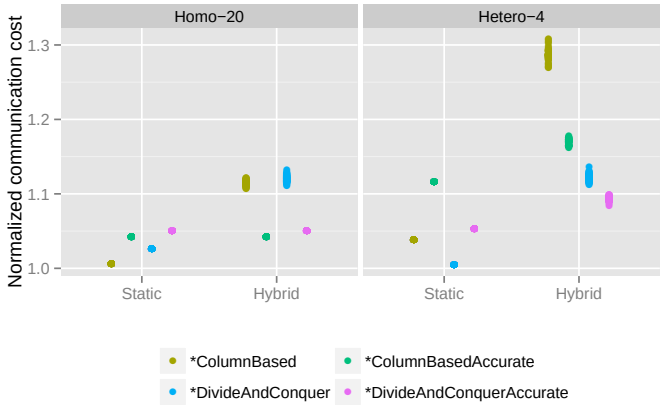


Figure 5. Communication cost of MINCOST and hybrid strategies for HOMOGENEOUS-20-CPUS and HETEROGENEOUS-4-GPUS for static setting

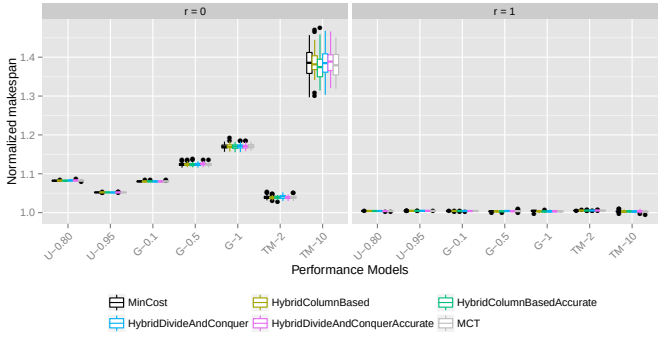


Figure 6. Makespan for different level of replication and different algorithms

dynamic strategies.

Let us first consider makespan, since indeed our goal is to minimize the execution time. In this setting, static heuristics fail to achieve this objective, and the results are even worse when the variance of speeds increases, like for GAUSSIAN-0.5, GAUSSIAN-1 or TWOMODES-10, where the ratio on the makespan can be larger than 2, see Figure 11 in the Appendix. This can be explained by the estimations of the speeds and by the fact that one processor can have a really bad execution sometimes (what we want to simulate with TWOMODES-10). In the case of smaller variance, in particular UNIFORM-0.95, the results are better and close to the expected time for the "accurate" versions, but the ratios increase with the number of processors, the risk of a bad estimation increasing in this case. For the heuristics with a dynamic part (see Figure 6 and Figure 12), replication is compulsory to have a good makespan, in particular when the variance is large. However, limiting to at most one replication by task is enough. The reason of such a bad execution time (1.6 in the worst case for HETEROGENEOUS-4-GPUS) when there is no replication is that CPUs are really slower than the GPU, so that assigning a task to a CPU instead of a GPU makes a large difference on the computation time. The replication mechanism fixes this problem.

For the communication cost results (see Figure 7 and Figure 7), static algorithms are excluded since they achieve the same performance than in the static case, but their makespan is very large. We also excluded MCT in order to keep it readable (but they can be seen in Figure 14), since the communication cost of MCT is really higher than those of the other algorithms (it is larger than 8 for HETEROGENEOUS-4-GPUS or HOMOGENEOUS-20-CPUS). For the other algorithms, MINCOST and the hybrid ones, we only consider the case where we allow one replication by task, as explained previously. First, we can notice that even if MINCOST has the worst ratio of these five algorithms, it stays below 3. We can also notice that MINCOST exhibits a better robustness against the variance of the platform.

Hybrid strategies suffer more of an increase of the variance since their static assignment becomes less effective, in particular because of the poor reliability of the speeds estimation. However the ratio is less than 1.5 (even less than 1.25 for HETEROGENEOUS-1-GPU) for the small variance (GAUSSIAN-0.1, TWOMODES-2, UNIFORM-0.80 and UNIFORM-0.95) and less than 2 in most cases in presence of large variance, what is always better than MINCOST. One can notice that HYBRIDDIVIDEANDCONQUERACCURATE, that achieve a better balance, is more effective for low variance since there is less job stealing. In the case of high variance, the dispersion of the results makes it difficult to have a clear hierarchy, even if HYBRIDDIVIDEANDCONQUER is a little better because it has a cheaper initial static repartition.

Discussion: From these simulations we can retain some facts. First, resource based strategies are more efficient when we want to minimize communications, and they can be time-optimal. Second, purely static partitioning are not reliable enough to be used in practice, but adding a dynamic part to them is both a cost-efficient and time-optimal strategy.

VI. CONCLUSION

In this paper, we consider the problem of allocating and scheduling a Matrix Multiplication onto a set of heterogeneous resources whose performance are not known in advance and may vary over time. On the one hand, since tasks share input data, it is crucial to use clever allocation schemes that typically cannot be found with the myopic view of a purely dynamic runtime strategy. On the other hand, it is often believed that in such unpredictable settings, only dynamic runtime strategies have a chance to obtain good results in practice. We have thus analyzed the behavior of purely static, purely dynamic and hybrid strategies on a number of representative platforms and distributions of processing speeds. We have shown that both static strategies and dynamic strategies without replication fail to obtain a reasonable makespan in some scenarios, but that replicating once is enough. On the other hand, dynamic strategies yield a consistently low makespan but at the expense of a very high communication cost. At last, hybrid strategies are able to get the best of both worlds, even in situations with very large variance. This supports strongly the addition

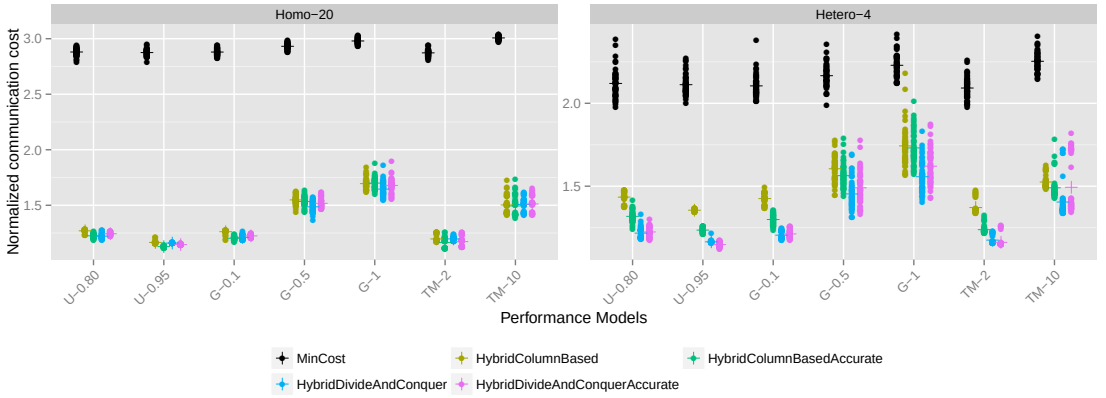


Figure 7. Communication cost of MINCOST and hybrid strategies for HOMOGENEOUS-20-CPUS and HETEROGENEOUS-4-GPUS for dynamic setting.

of more static knowledge in task-based runtime systems. Furthermore, this also motivates the design and analysis of good hybrid strategies, as well as the theoretical development of static algorithms to be used as input of hybrid strategies.

VII. ACKNOWLEDGMENTS

This work is partially supported by the Agence Nationale de la Recherche, under grant ANR-13-MONU-0007 (project Solhar, http://solhar.gforge.inria.fr/).

REFERENCES

[1] M. R. Garey and D. S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[2] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi, *Complexity and Approximation*. Springer Verlag, 1999.

[3] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[4] D. Hochbaum, *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company, 1997.

[5] L. Epstein and R. van Stee, "Online bin packing with resource augmentation." *Discrete Optimization*, vol. 4, no. 3-4, pp. 322–333, 2007.

[6] O. Beaumont, H. Larchevêque, and L. Marchal, "Non linear divisible loads: There is no free lunch," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 863–873.

[7] O. Beaumont and L. Marchal, "Analysis of dynamic scheduling strategies for matrix multiplication on heterogeneous platforms," in *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. ACM, 2014.

[8] T. White, *Hadoop: The definitive guide*. Yahoo Press, 2010.

[9] H.-J. Lee, J. P. Robertson, and J. A. Fortes, "Generalized cannon's algorithm for parallel matrix multiplication," in *Proceedings of the 11th international conference on Supercomputing*. ACM, 1997, pp. 44–51.

[10] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms," in *Euro-Par 2011 Parallel Processing*. Springer, 2011, pp. 90–109.

[11] T. Gautier, X. Besseron, and L. Pigeon, "Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors," in *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, ser. PASCO '07. New York, NY, USA: ACM, 2007.

[12] G. Bosilca, A. Bouteiller, A. Danalis, T. Hérault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed dag engine for high performance computing," *Parallel Computing*, vol. 38, no. 1-2, pp. 37–51, 2012.

[13] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí, "Parallelizing dense and banded linear algebra libraries using SMPSs," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 18, pp. 2438–2456, 2009.

[14] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[15] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, "A hybridization methodology for high-performance linear algebra software for GPUs," *in GPU Computing Gems, Jade Edition*, vol. 2, pp. 473–484, 2011.

[16] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Multifrontal qr factorization for multicore architectures over runtime systems," in *Euro-Par 2013*, 2013, pp. 521–532.

[17] J. V. F. Lima, T. Gautier, N. Maillard, and V. Danjean, "Exploiting concurrent gpu operations for efficient work stealing on multi-gpus," in *Proceedings of SBAC-PAD '12*, 2012, pp. 75–82.

[18] J. Bueno, J. Planas, A. Duran, R. Badia, X. Martorell, E. Ayguade, and J. Labarta, "Productive programming of gpu clusters with ompss," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, May 2012, pp. 557–568.

[19] W. Cirne, F. Brasileiro, D. Paranhos, L. F. W. Góes, and W. Voorsluys, "On the efficacy, efficiency and emergent behavior of task replication in large distributed systems," *Parallel Computing*, vol. 33, no. 3, pp. 213–234, 2007.

[20] A.-M. Oprescu, T. Kielmann, and H. Leahu, "Stochastic tail-phase optimization for bag-of-tasks execution in clouds," in *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, ser. UCC '12. IEEE Computer Society, 2012.

[21] C. Boeres, A. Lima, and V. Rebello, "Hybrid task scheduling: integrating static and dynamic heuristics," in *15th Symposium on Computer Architecture and High Performance Computing, 2003. Proceedings*, Nov. 2003, pp. 199–206.

[22] J. V. F. Lima, T. Gautier, V. Danjean, B. Raffin, and N. Maillard, "Design and analysis of scheduling strategies for multi-cpu and multi-gpu architectures," *Parallel Computing*, vol. 44, pp. 37–52, 2015.

[23] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Minimizing communication in linear algebra," *SIAM Journal on Matrix Analysis and Applications*, vol. 32, no. 3, pp. 866–901, Jul. 2011, arXiv: 0905.2485. [Online]. Available: http://arxiv.org/abs/0905.2485

[24] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Partitioning a square into rectangles: Np-completeness and approximation algorithms," *Algorithmica*, vol. 34, no. 3, pp. 217–239, 2002.

[25] H. Nagamochi and Y. Abe, "An approximation algorithm for dissecting a rectangle into rectangles with specified areas," *Discrete Applied Mathematics*, vol. 155, no. 4, pp. 523 – 537, 2007.

[26] A. Fügenschuh, K. Junosza-Szaniawski, and Z. Lonc, "Exact and approximation algorithms for a soft rectangle packing problem," *Optimization*, vol. 63, no. 11, pp. 1637–1663, 2014.

VIII. Appendix

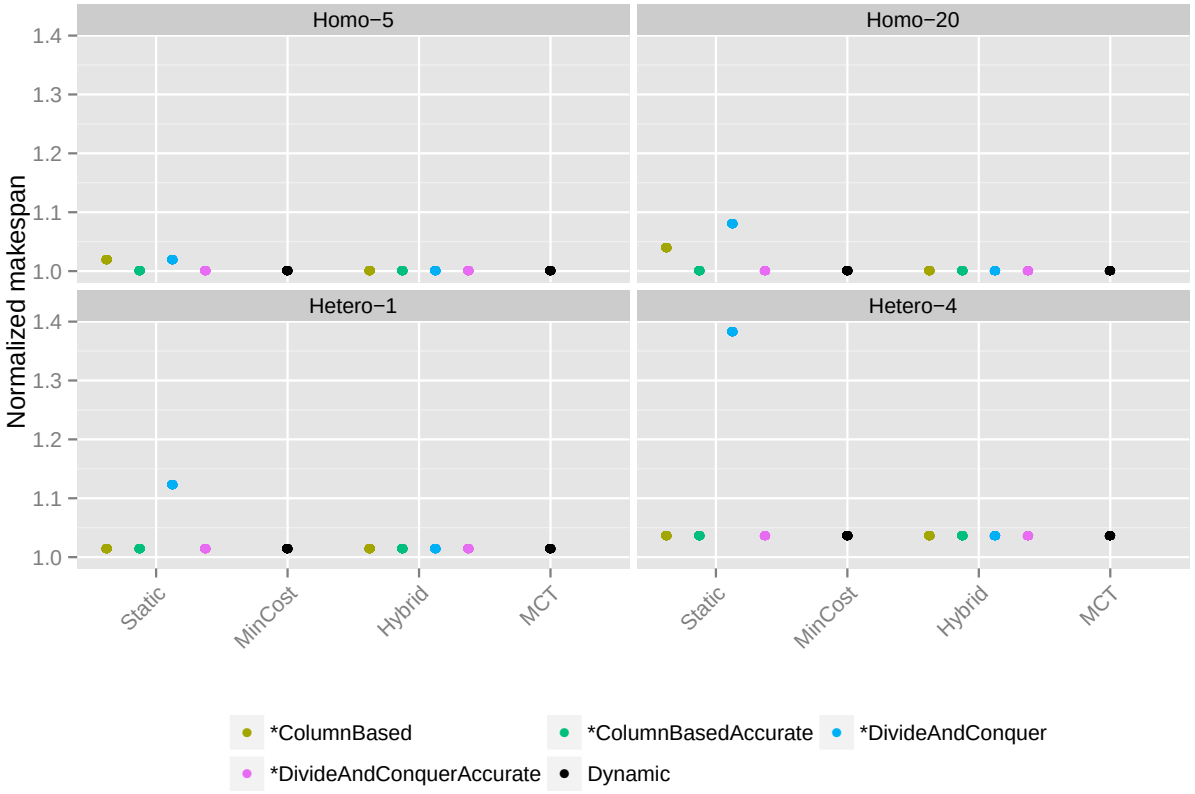


Figure 8. Makespan for different algorithms and different platforms and static setting.

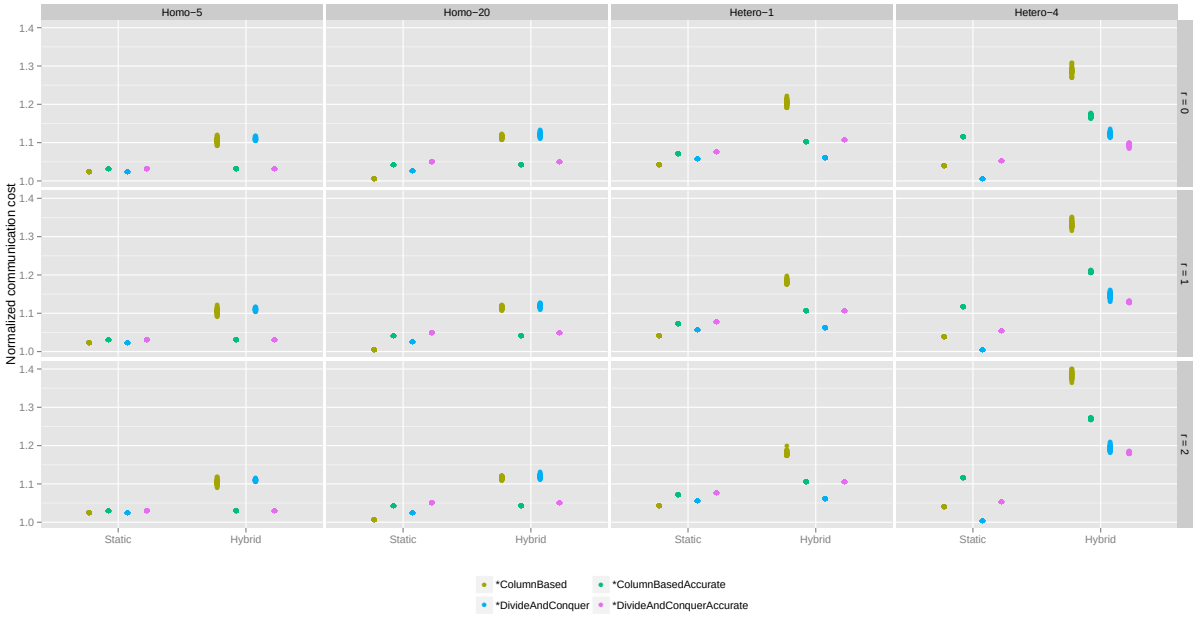


Figure 9. Communication cost of MINCOST and hybrid strategies for static setting

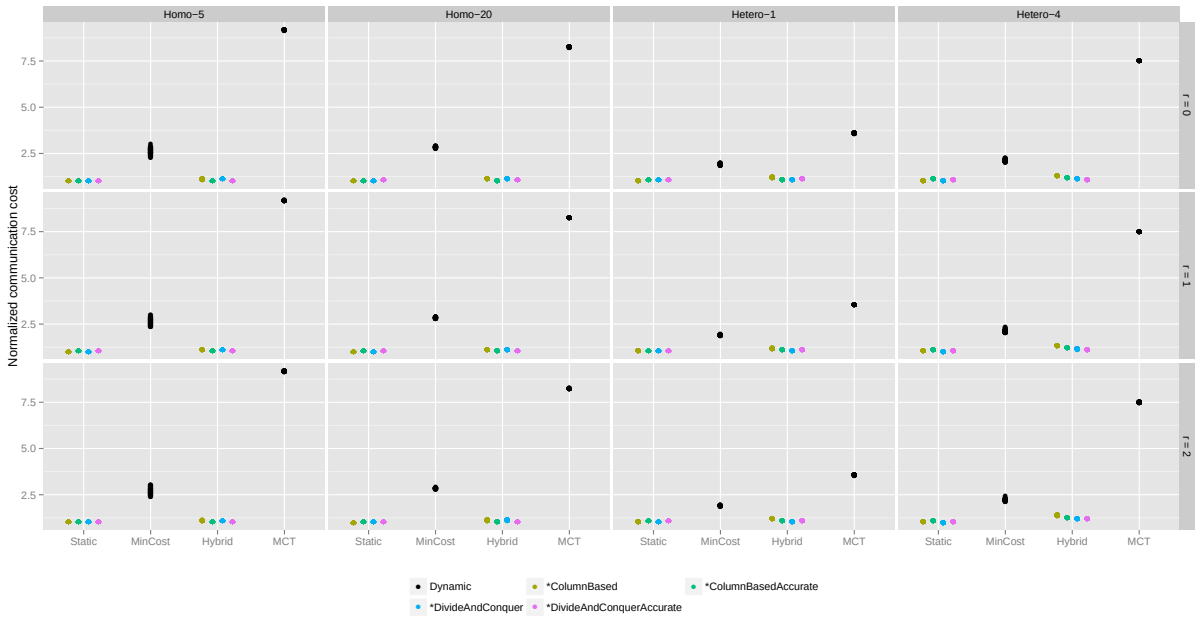


Figure 10. Communication cost of MINCOST and hybrid strategies for static setting

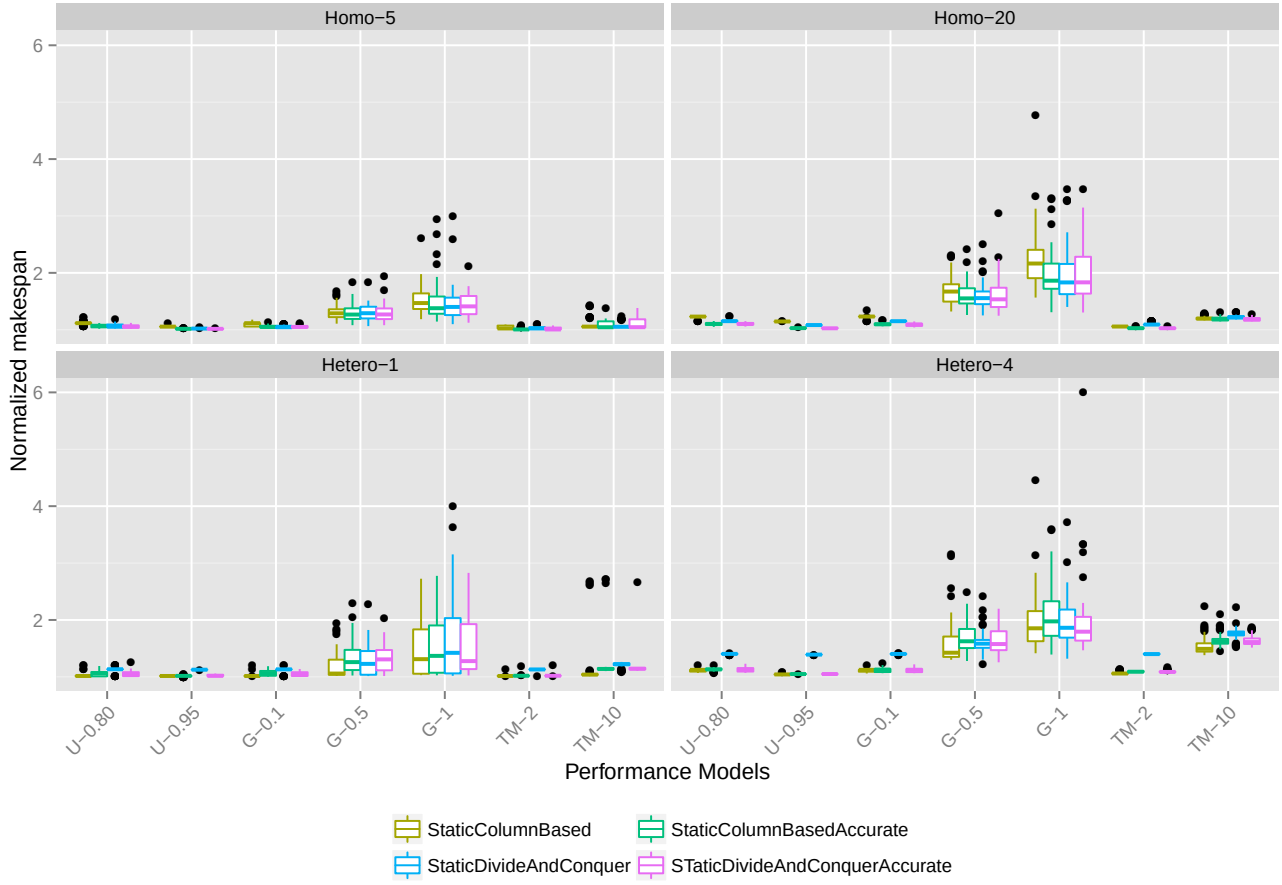
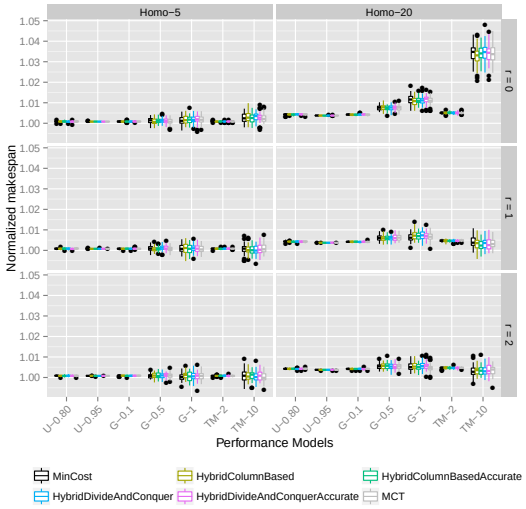
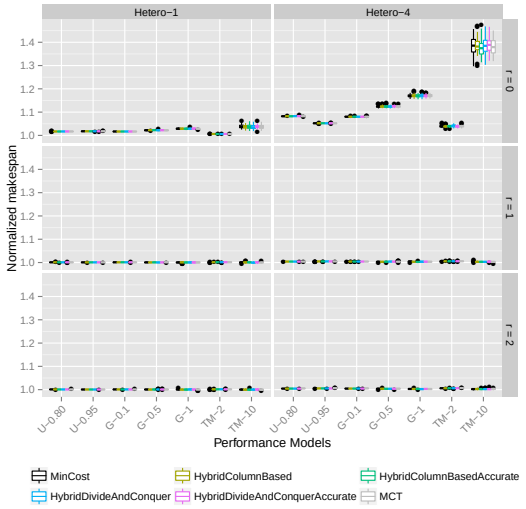


Figure 11. Makespan for static algorithms



(a) Makespan as a function of the level of replication for different algorithms on homogeneous platforms



(b) Makespan as a function of the level of replication for different algorithms on heterogeneous platforms

Figure 12. Makespan as a function of the level of replication for different algorithms

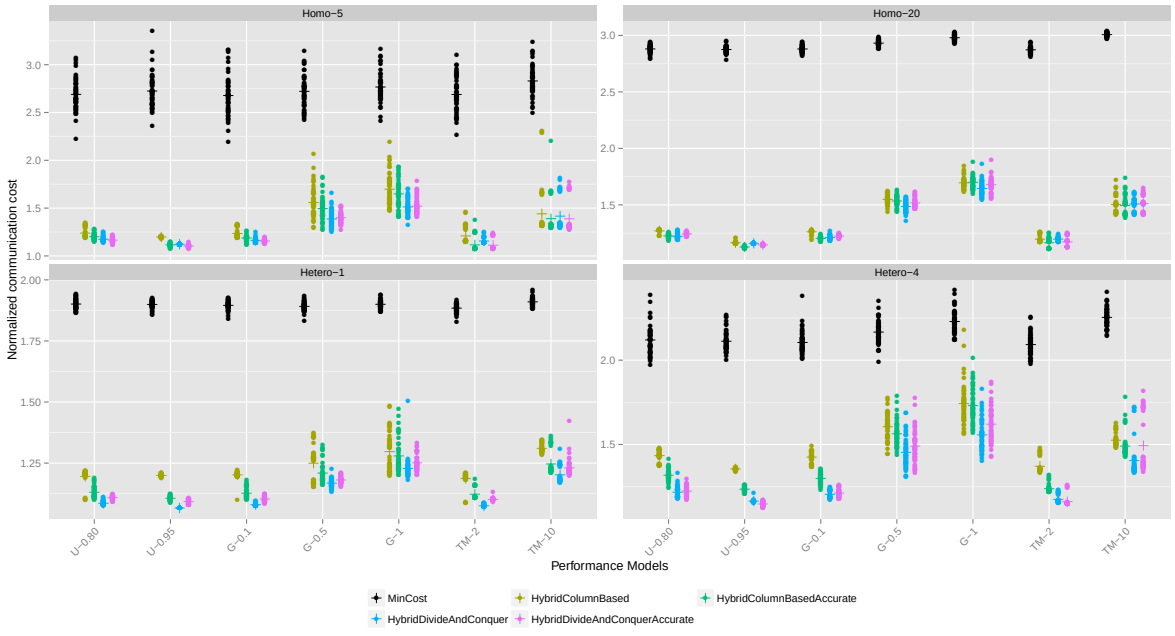


Figure 13. Communication cost of MINCOST and hybrid strategies for dynamic setting.

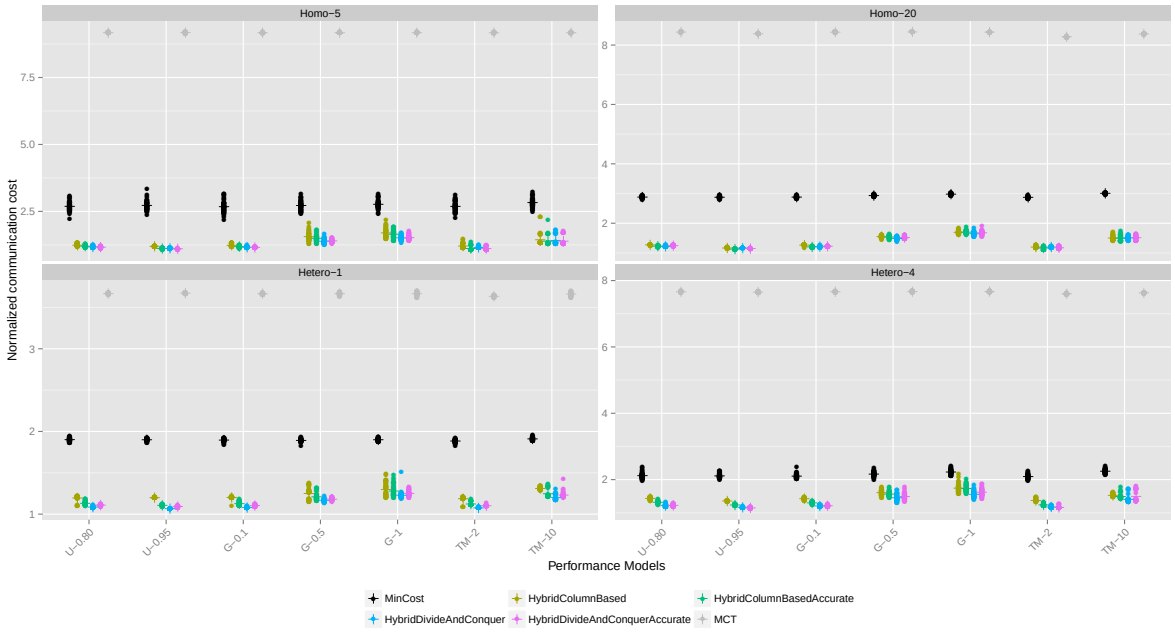


Figure 14. Communication cost of MINCOST, MCT and hybrid strategies for dynamic setting.