J Supercomput (2017) 73:3738–3760
DOI 10.1007/s11227-017-1971-4

HeDPM: load balancing of linear pipeline applications on heterogeneous systems

Andreu Moreno[1] · **Anna Sikora**[2] · **Eduardo César**[2] · **Joan Sorribes**[2] · **Tomàs Margalef**[2]

Published online: 2 February 2017

Abstract This work presents a new algorithm, called Heterogeneous Dynamic Pipeline Mapping, that allows for dynamically improving the performance of pipeline applications running on heterogeneous systems. It is aimed at balancing the application load by determining the best replication (of slow stages) and gathering (of fast stages) combination taking into account processors computation and communication capacities. In addition, the algorithm has been designed with the requirement of keeping complexity low to allow its usage in a dynamic tuning tool. For this reason, it uses an analytical performance model of pipeline applications that addresses hardware heterogeneity and which depends on parameters that can be known in advance or

This work has been partially supported by Ministerio de Economía y Competitividad MINECO-Spain under contract TIN2014-53234-C2-1-R and Generalitat de Catalunya GenCat-DIUiE (GRR) 2014-SGR-576.

✉ Anna Sikora
anna.sikora@uab.cat

Andreu Moreno
amoreno@euss.cat

Eduardo César
eduardo.cesar@uab.cat

Joan Sorribes
joan.sorribes@uab.cat

Tomàs Margalef
tomas.margalef@uab.cat

[1] Escola Universitària Salesiana de Sarrià (EUSS), Passeig Sant Joan Bosco, 74, 08017 Barcelona, Spain

[2] Computer Architecture and Operating Systems Department, Universitat Autònoma de Barcelona, 08193 Bellaterra, Spain

measured at run-time. A wide experimentation is presented, including the comparison with the optimal brute force algorithm, a general comparison with the Binary Search Closest algorithm, and an application example with the Ferret pipeline included in the PARSEC benchmark suite. Results, matching those of the best existing algorithms, show significant performance improvements with lower complexity ($O(N^3)$, where N is the number of pipeline stages).

Keywords Load balancing · Performance · Pipeline · Heterogeneous systems

1 Introduction

This work focuses on developing a comprehensive dynamic performance tuning strategy for linear pipeline applications running on heterogeneous and distributed systems. It represents a further step to our previous contributions [13,25,26] in the development of performance models associated with the application's structure for dynamic performance tuning.

Parallel/distributed programming has gone from being a promising approach for improving the performance of many applications, to be mandatory in order to take full advantage of the current hardware systems and fulfill the requirements of current applications. However, in contrast to sequential programming, several challenges have still to be properly addressed in all phases of the development cycle of parallel/distributed applications. Obviously, one of the best ways to afford them is to develop tools that support the design, coding, analysis and tuning of parallel/distributed applications.

The actual performance of a parallel application usually is far lower than the expectations of its developers. To improve the performance of an application, its behavioral characteristics must be taken into consideration. If the application behaves in a regular way, a static performance analysis and tuning process will be enough. Historically, this process has consisted in a very precise tailoring of the application to a specific architecture, as can be clearly seen in the following sequence of works by Arabnia et al. [2,3,9,10,36]. This approach has laid the groundwork for defining new strategies focused on providing support to a wider set of applications, as for example the strategy for automatically determining the best values for certain application parameters, such as compilation flags or communication-related parameters described in [24,32].

However, if the application changes its behavior from execution to execution, or even within a single execution, dynamic monitoring and tuning strategies should be used. A dynamic monitoring strategy needs to gather enough information about the application at execution time to ensure that performance problems are quickly detected and solved. To obtain this information, the application must be instrumented, and logically, more instrumentation will provide more information. However, inserting instrumentation and applying changes in the application will negatively affect its performance. Consequently, dynamic monitoring and tuning strategies must be efficient and, at the same time, minimize intrusion on the application. We claim that this apparent contradiction can be solved if the application structure (Farmer, Pipeline, SPMD, etc.) is known in advance and high-level performance models can be defined (as in [29] or [30]) to be integrated in an analysis and tuning tool.

One of the well-known parallel programming structures is the *pipeline*. It is used as the most direct way to implement algorithms that consist of performing a sequence of calculations on a sequence of inputs. Each of these calculations are known as stages, which can be concurrently applied to different input sets. Pipeline applications performance has been widely studied, as shown in the related work presented in Sect. 2. Analyzing these studies, it can be concluded that the most important inefficiency associated with this structure is the load unbalance among stages because the throughput of a pipeline is determined by the slowest stage. The application will have bottlenecks if significant differences among the computational effort of the pipeline stages exist. This inefficiency is suitable to be solved dynamically because it does not depend exclusively on the application design, but also on run-time conditions, such as input data with different processing demands and unsteady computational performance.

In our previous work, we developed an algorithm called Dynamic Pipeline Mapping (DPM) [26,27]. It is based on the replication of the slowest stages and the gathering of the fastest ones for solving this problem on clusters of workstations. This model is only focused on the application structure assuming that the hardware is more or less homogeneous. However, the underlying performance model used for making predictions in DPM does not work for really heterogeneous systems.

Heterogeneous systems have become widely spread with grid and cloud technologies, and consequently, new strategies have to be developed to solve the problem of tuning the application performance in these systems. In this paper we define a new algorithm, based on DPM, that considers heterogeneity in hardware processing and communication capacity and can be applied to a wide range of systems, from traditional clusters of workstations to grids or clouds.

This new algorithm for dynamically improving the performance of pipeline applications executed in heterogeneous systems has been called Heterogeneous Dynamic Pipeline Mapping (HeDPM). It is aimed at improving the efficiency in the use of resources with a reasonable complexity ($O(N^3)$, where N is the number of pipeline stages) that allows for tuning the applications performance at run-time. For this reason, we consider that filling-in and draining the pipeline are transient phases, whose inefficiencies cannot be solved dynamically. Similarly to the DPM algorithm, HeDPM improves the application performance by gathering fast consecutive stages in the same processor and by replicating slowest stages in several processors in order to increase their throughput. However, HeDPM takes into consideration the computational capacity of processors and also includes communication cost. These extensions are necessary for the model to be applied to distributed systems, where heterogeneity comes from different processor speed, but also from different communication speeds. Moreover, the proposed algorithm is able to obtain good results in significantly less time than other comparable methods.

The algorithm sorts processors by their computing capacity, and pipeline stages by their computational load and communication requirements. Then, it matches stages to processors following these ordered lists gathering the pipeline's fastest stages and replicating the slowest ones. For this matching process, it is necessary to predict the behavior of gathered and replicated stages, which is accomplished through a set of mathematical expressions defined in Sect. 3. We present the details of the HeDPM

algorithm in Sect. 4, where we show that HeDPM has a polynomial complexity that allows for its usage at run-time in a dynamic tuning tool.

We present a set of experiments in Sect. 5. First, we assess HeDPM against the optimal brute force mapping algorithm. The cost of this optimal algorithm is not affordable because of the underlying complexity of the problem, and consequently, we have executed this algorithm only for small number of stages and processors. Second, for bigger systems we have done a comparison with the Binary Search Closest (BSC) algorithm [8] because this is, to the best of our knowledge, one of the best references in the literature. Finally, we provide a deeper assessment using HeDPM on a real application example, an MPI adaptation of the Ferret pipeline included in the PARSEC benchmark suite [11].

At the end, conclusions and the summary of the main achievements presented in this study are included in Sect. 6.

2 Related work

Improving the performance of pipeline applications has been an intensive field of research. Subhlok and Vondram [35] stated a mapping algorithm which optimizes latency under some throughput constraints for purely linear pipelines. They proposed a policy for maximizing the throughput of homogeneous pipelines (all processors have the same processing capacity). Hoang and Rabey [19] showed a scheduling algorithm for maximizing the throughput based on the optimization of a flow graph representation of the program. Later this approach was improved by Yang et al. [37] and Guirado et al. [18] considering that the application performs several iterations. All theses works are based on homogeneous pipelines, whereas our approach deals with heterogeneous systems.

Lin et al. [22] proposed a heuristic algorithm, called PaPiLO, to minimize latency under power and throughput constraints in high-performance embedded systems. This algorithm presents several similarities with our proposal because it also uses gathering and replication of stages to meet its objectives. In some cases, it goes beyond our approach because it considers multiple objectives (throughput, latency and power consumption) and more general application structure (DAGs instead of chained pipelines). However, PaPiLO is focused in homogeneous systems, particularly homogeneous embedded systems, while our approach is focused on heterogeneous parallel/distributed systems. As a consequence of the differences on the objective systems, the final goals of both strategies are significantly different and their comparison can be difficult.

Almeida et al. [1] proposed a model that minimizes the execution time of a pipeline application considering its whole execution, including the filling-in and draining phases. The model is used to gather some stages on a processor to improve the performance of the application on a cluster of heterogeneous processors. This is a static approach, while ours can be used at run- time. Besides, we consider that filling-in and draining phases are transient phases that cannot be solved dynamically.

Kijsipongse and Ngamsuriyaroj presented in [21] two methods for optimal pipeline placement on grid systems. The first is a heuristic algorithm for linear pipelines, and the

second uses linear programming for multipath pipelines. In both cases, they considered computation load of stages and the communication costs between stages. Our aim is to present a dynamic tuning algorithm, while the indicated paper showed an optimal algorithm with a high complexity that is not easy to be used at run-time.

Pinar et al. [28] studied the partitioning of one-dimensional non-uniform workload arrays with optimal load balancing for heterogeneous systems. They considered the case where order of processors is specified and the case where processor permutation is allowed. This approach, similarly to HeDPM, uses grouping for improving the resource usage (assigning some stages to the same processor), but it does not use replication (to assign the same stage to the different processors) to balance the load and, in addition, they did not model communications.

Do Nascimento et al. [14] presented a scheduler for Datacutter grid environment. It uses linear programming to decide the numbers of copies of each pipeline stage, called filters, and their placement in the grid node. This approach uses replication like HeDPM to balance the load, but it does not use grouping to improve the use of resources. Moreover, linear programming guaranties the optimal solution, but, at the same time, it implies a high complexity.

Spencer et al. [34] presented a scheduling of multiple data analysis operations, represented as a pipelined chain, for grid-based cluster environments. They executed multiple copies of each pipeline stage, which is the same idea as replication, and allow for placing multiple stages in the same processors (grouping concept). They use a heuristic based on list scheduling that swept the pipeline from beginning to end, but in contrast to HeDPM, their approach does not consider a general heterogeneous system, and it is limited to interconnection of homogeneous clusters.

Murray Cole's group at the School of Informatics of the University of Edinburgh studied the implications of using skeletons in the scheduling algorithms by using models based on process algebras (specifically PEPA in [7,15]). González-Vélez and Cole [17] presented a methodology that uses the information obtained from an application based on a skeleton to improve performance at run-time. It was based on heterogeneous distributed systems and considered task farm and pipeline patterns. The heuristic used was similar to HeDPM but without considering the cost of communications.

Benoit et al. [6] presented an extensive survey of existing algorithms for pipelined workflow scheduling, including our previous work [27] and many references to their work on this topic. From the presented set of algorithms, we have focused on [8], which describes several heuristics to map parallel pipeline applications on heterogeneous platforms. They use the concept of interval-based mapping, a grouping approach similar to HeDPM. Although these algorithms are computer intensive, we have used their BSC algorithm, the one with best results, as a reference to evaluate our results. BSC algorithm runs a binary search on the period at which the pipeline produces results. For a given time, it studies if there is a feasible solution, starting with the first stage and building groups of stages, to fit into the available processors.

Finally, our work is based on general heterogeneous platforms, mainly focused on grid systems, but a lot of work has been recently done for the multi-core and GPU (graphics processing unit) environments. Sanchez et al. [31] presented a scheduler for pipelines that performs fine-grain dynamic load balancing, guarantees bounds on

resources consumed, and has small scheduling overhead. This scheduler is provided for multi-core SMP machines. Goli et al. [16] presented an heterogeneous streaming pipeline implementation using the FastFlow parallel programming framework for multi-core platforms, which allocates the pipeline stages to multi-core CPUs and multi-GPUs. The authors demonstrate an implementation of a scalable GPU numerical linear algebra testbed using FastFlow. This work is under the ParaPhrase project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. Augonnet et al. [4] presented StarPU, a run-time system providing an unified execution model with the goal of furnishing designers with a convenient way to generate parallel tasks over heterogeneous hardware and easily develop and tune powerful scheduling algorithms. Finally, the PEPPHER project [5] addressed the problems of performance portability and efficient use of hybrid systems, which are the fusion of many-cores CPUs and GPUs. It suggests solutions that combine static and dynamic scheduling, run-time tuning, compilation techniques, and monitoring tools. Different parallel languages and frameworks are supported, and pipeline is one of the reported structures.

3 Performance model for pipeline applications

The aim of this work is to develop an algorithm for dynamically improving the performance of a pipeline application executed in an heterogeneous system, such as traditional clusters of workstations, grids and clouds. We only consider linear pipelines that run an orderly sequence of calculations on a sequence of input data sets using a distributed memory programming model. Each of these calculations is known as stage, and each stage receives data from the previous stage, runs some computation, and sends data to the next stage. Table 1 introduces the terminology used in the proposed model.

Table 1 Summary of the terminology used by the HeDPM algorithm

Notation	Description
P	Number of available processors
N	Number of pipeline stages
$s(p)$	Processor speed. Speed of each of P processors in floating point operations per second (FLOPS), $1 \leq p \leq P$
$w(n)$	Stage workload. Workload of each stage in floating point operations, $1 \leq n \leq N$
$d(n)$	Size of output data of each stage in bytes, $1 \leq n \leq N$
$B(p_i, p_j)$	Bandwidth between processors p_i and p_j in bytes per second
$c(p_i, p_j)$	Set-up communication time between processors p_i and p_j in bytes per second
T_{ideal}	Pipeline ideal production time, which is the time between two consecutive outputs of the application when all the stages are perfectly balanced (every stage takes the same time including computation and communication)
R	Number of replicas
$T_{Replica}$	Time spent by a replicated stage to process a task
T_{Group}	Time spent by a set of gathered stages to process a piece of data
$t(n, p)$	Execution time of stage n on processor p

System heterogeneity is modeled by taking into consideration each processor speed ($s(p)$), communication speed and set-up communication time between each pair of processors ($B(p_i, p_j)$, $c(p_i, p_j)$), and we assume that these parameters are constant at run-time and communications are synchronous.

A pipeline application is perfectly balanced when all stages take the same time using the available resources. In this case, the time between two consecutive outputs (*production time*) is minimal (T_{ideal}) and the throughput is maximized. Logically, any difference between the amount of computation done by different stages or differences in communication links or computing capacities between processors may cause the application unbalance. Consequently, the algorithm must be able to find a mapping of the N pipe stages to P available processors that leads to a production time T close to T_{ideal}.

It is worth noticing that a pipeline production time is equal to the execution time of its slowest stage. Therefore, if the algorithm is able to estimate T_{ideal} it will be able to match stages to processors in such a way that each stage execution time will be close to T_{ideal}.

We have based this mapping process on two techniques: replication of slower stages and gathering of faster stages. Stage replication consists in executing a copy of the replicated stage in an unused processor with the objective of processing multiple data elements in parallel, while stage gathering consists in executing two or more consecutive stages in the same processor.

The replication approach used in our proposal differs from the replication and duplication concepts introduced in PaPiLO because we only replicate single stages and our replicas always process different data elements. These differences are due to the particular goals and application structure considered by each algorithm.

To determine the best replication-gathering combination, HeDPM must be able to predict the behavior of the application using any mapping. This is fulfilled by defining expressions that model a replicated stage and a set of gathered stages. This model has been defined as simply as possible to decrease its associated computational complexity because it should be used at run-time. This section explains this performance model.

3.1 Model for stage replication

It can be said that a stage is slow when its execution time is bigger than the one of its neighboring stages. In this case, this difference can be reduced if the code of the stage is replicated in other processors in order to process several input data sets in parallel, but how many processors should be assigned to replicate it? The answer to this question depends on whether the data elements have to be processed in order or not.

In this work, we assume the worst case when data order must be preserved. Then, we need a model for predicting how many processors should be used for replicating the stage considering that the order of data elements must be preserved. In this case, the slowest processor assigned to the replicated stage will determine the replica overall execution time. This processor will be involved in processing a new task every R data

sets, where R is the number of replicas. Therefore, the time spent by a replicated stage to process a task can be expressed as:

$$t_{Replica}(n,\, p = \{i, \ldots, i+R\}) = \frac{1}{R} \max_{p=i}^{i+R}(t(n,\, p)) \tag{1}$$

where

$t(n,\, p)$: execution time of stage n on processor p.

3.2 Model for stage grouping

A stage is fast when its execution time is smaller than the one of its neighboring stages. When the number of resources (processors) is limited, a set of contiguous fast stages can be gathered in the same processor in order to free resources that can be used to replicate slow stages. The amount of time that a set of gathered stages takes to process a piece of data must be modeled in order to be able to predict how many fast stages can be gathered in a particular processor. We have modeled a set of grouped stages as the addition of the execution times of each of the stages that belong to the group in a specific processor. Consequently, for k stages we define:

$$t_{Group}(n = \{i, \ldots, i+k\},\, p) = \sum_{n=i}^{i+k} t(n,\, p) \tag{2}$$

Except for the first and the last stages in the group, communications within a group are internal communications. These internal communications are significantly faster than the inter-process ones, and for this reason they are not considered in Eq. 2.

4 HeDPM algorithm

In this section we show the details of the algorithm to balance the load of pipeline applications running on heterogeneous systems. The aim of the algorithm is to efficiently find a pattern of stage gathering and replication that leads to a balanced, or as balanced as possible, resource utilization.

As we have seen in Sect. 2, there are several proposals of load balancing optimization strategies, but most of them are infeasible for dynamic tuning environments because of their high complexity. We have devised an heuristic algorithm (HeDPM) that may not lead to an optimal execution time, but it leads with low complexity to an execution time close to the optimal one.

Algorithm 4.1: HeDPM($ProductionTimeObjective$)

```
comment: First step: stages and processors sorting
StagesAndProcessorsSorting()

while stages
  comment: Second step: ideal production time calculation
  if ProductionTimeObjective == 0
    then T_ideal ← IdealProductionTime()
    else T_ideal ← ProductionTimeObjective

  comment: Third step: mapping
  T ← ExecutionTime(1stStage, 1stProcessor)
  if T ∈ [0.95 * T_ideal, 1.05 * T_ideal]
    then AssignStageToProcessor()

  else if T > 1.05 * T_ideal
    then
      while T > 1.05 * T_ideal
        do
          Replica(AddProcessor)
          T ← ExecutionTime(Replica)
      AssignStageToProcessors(Replica)

  else if T < 0.95 * T_ideal
    then
      while T < 0.95 * T_ideal and ContStages()
        do
          Group(AddStage)
          T ← ExecutionTime(Group)
      AssignStageToProcessor(Group)
return (T)
```

The algorithm starts building a sorted list of stages and processors taking into account the processing needs of stages and the processing capacity for processors, which can be obtained by inserting the appropriate instrumentation. Then, the ideal production time is calculated and used as a reference. Finally, the algorithm runs an iterative process at run-time where stages and processors are matched, beginning with the most demanding stages and with the most powerful processors. It frees processors by gathering faster stages and uses these processors to replicate slower stages with the objective to be as close as possible to the ideal production time. The result is a mapping of stages to processors that tends to the optimal load balancing.

As it can be seen in Algorithm 4.1, HeDPM has three steps:

1. *First step: stages and processors sorting*

   The algorithm starts by creating two sorted lists, one with the stages and another one with processors. The stage list is sorted by the stage production time in the average processor in descending order, i.e., slowest stage first. The use of average processor is motivated because calculation of production time needs to know the

placement of stages to processors, but this is actually the algorithm output and it is not known at this step. Each stage is characterized by its workload ($w(n)$) and output data size ($d(n)$). The execution time of each stage n (from 1 to N) is computed using the arithmetic mean values of processor speed ($\bar{s}$), communication bandwidth ($\bar{B}$), and set-up time ($\bar{c}$) as shown in Eq. 3.

$$t(n, \bar{p}) = \left[\bar{c} + \frac{d(n-1)}{\bar{B}}\right] + \left[\frac{w(n)}{\bar{s}}\right] + \left[\bar{c} + \frac{d(n)}{\bar{B}}\right], \forall n \in [1, N]. \quad (3)$$

In this expression, $\bar{c} + \frac{d(n-1)}{\bar{B}}$ is the communication time from the processor with previous stage, $\frac{w(n)}{\bar{s}}$ is the processing time of the stage, and $\bar{c} + \frac{d(n)}{\bar{B}}$ is the communication time to the processor with next stage.

In the same way, the processor list is sorted by the execution time of the average stage in ascending order, i.e., fastest processor first. The use of average stage is also motivated because calculation of production time needs to know the placement of stages to processors, which is not known at this step. Each processor is characterized by its speed ($s(p)$), communication bandwidth ($B(p, p+1)$) and communication set-up time ($c(p, p+1)$). For each processor p (from 1 to P), the execution time of the average stage is computed using the arithmetic mean values of stage workload ($\bar{w}$) and output data size ($\bar{d}$) as shown in Eq. 4.

$$t(\bar{n}, p) = \left[c(p-1, p) + \frac{\bar{d}}{B(p-1, p)}\right] + \left[\frac{\bar{w}}{s(p)}\right] + \left[c(p, p+1) + \frac{\bar{d}}{B(p, p+1)}\right], \forall p \in [1, P]. \quad (4)$$

where $c(p-1, p) + \frac{\bar{d}}{B(p-1,p)}$ is the communication time from the processor with previous stage, $\frac{\bar{w}}{s(p)}$ is the processing time, and $c(p, p+1) + \frac{\bar{d}}{B(p,p+1)}$ is the communication time to the processor with next stage.

2. *Second step: ideal production time calculation*

The criteria for seeking the best mapping is to be as close as possible to the ideal production time (T_{ideal}). First, we calculate the mean available processor speed per stage (s_a), where we take into account the average processor speed ($\bar{s}$) and the effect of having a number of processors greater or smaller than the number of stages (Eq. 5).

$$s_a = \frac{P}{N}\bar{s} \quad (5)$$

Then, the ideal production time for a generic stage is calculated by adding the averages of communication time with the previous stage, the processing time, and communication time with the next stage, using the average values of stage workload ($\bar{w}$), size of output data of each stage ($\bar{d}$), bandwidth ($\bar{B}$), and set-up

time ($\bar{c}$) as shown in Eq. 6.

$$T_{ideal} = \bar{c} + \frac{\bar{d}}{\bar{B}} + \frac{\bar{w}}{s_a} + \bar{c} + \frac{\bar{d}}{\bar{B}}. \quad (6)$$

3. *Third step: mapping*

The algorithm assigns the first processor from the processor list to the first stage from the stage list, matching in this way the most demanding stage with the most powerful processor. Then, it calculates the execution time (T) and compares it with the ideal production time (T_{ideal}) obtained in the second step:

- If $T \in [0.95 * T_{ideal}, 1.05 * T_{ideal}]$ the match between the stage and the processor is adequate. We present here an interval of radius 5% because we have empirically observed good results to keep the algorithm in a steady state. In general, this radius might need to be increased if a steady state is not reached.
- If $T > 1.05 * T_{ideal}$ the stage being analyzed demands more processing performance than the one offered by the processor under consideration. Consequently, the stage should be replicated using more processors from the processors list. The execution time of the stage is updated for each new processor added using Eq. 1. The replication process finishes when $T \leq 1.05 * T_{ideal}$.
- If $T < 0.95 * T_{ideal}$ the processor being analyzed will not be efficiently used by the stage under consideration. Consequently, more stages could be gathered in the same processor. However, to avoid external communications within a group, the algorithm only selects continuous stages. The execution time for the group of stages is updated for each new stage added to the group using Eq. 2. The gathering process finishes when $T \geq 0.95 * T_{ideal}$ or when no more continuous stages are available.

At the end of this step, the algorithm has matched one processor with one or more stages, creating a group of stages, or one stage with one or more processors, making a replica. Finally, the algorithm removes the matched stages and processors from the stages and processors lists and jumps to the second step updating the ideal production time for the remaining stages and processors. This iteration process finishes when there are no more stages to match.

The second step must be repeated for the remaining stages and processors because the assignment process usually does not fit exactly due to unavoidable rounding operations in the number of stages and processors. Therefore, sometimes more, sometimes less resources than necessary are used in each matching.

In general the resulting mapping is different whether the number of processors (P) is smaller, equal or greater than the number of stages (N). When the number of processors is smaller the mapping has a lot of groups, and when the number of processors is greater the mapping has a lot of replicas. When the number of stages and processors are similar the number of groups and replicas can be also similar, it depends on the system heterogeneity.

The main weakness of this algorithm is that the ideal production time T_{ideal} computed using Eq. 6 can be unrealistic if the heterogeneity of the hardware and/or computational load of stages is too high. This can lead to two different scenarios:

- Equation 6 gives a value smaller than the realistic ideal value. In this case, the matching of stages to processors will be too conservative, i.e., assigned resources will be underloaded. Consequently, the last iteration will have too much load to assign. This results in an unbalanced pipeline where the last processor increases the global production time.
- Equation 6 gives a value higher than the realistic ideal value. In this case, the matching of stages to processors will be too optimistic, i.e., assigned resources will be overloaded. Consequently, the last iteration will have too little load to assign. This results in an unbalanced pipeline where the last processor does not help the rest of the pipeline to reduce the global production time.

Algorithm 4.2 is used to solve this problem. After a first execution of Algorithm 4.1, the resulting mapping is analyzed to determine which of these two scenarios happens. Then, Algorithm 4.1 is executed several times fixing a decreasing or increasing (depending on the scenario) ideal production time objective. We have tested this algorithm in several situations, and results have been good by sweeping the production time objective in the interval $[TObj0, 1.5 * TObj0]$ in twenty equally spaced steps when the last stage is the most loaded one, and sweeping the production time objective in the interval $[0.5 * TObj0, TObj0]$ in twenty equally spaced steps when the last stage is the least loaded one.

Taking into account the number of data accesses done by the proposed algorithm, the time complexity is $O(N^3)$.

Taking into account the number of data accesses done by the proposed algorithm, Algorithm 4.1 has a time complexity of $O(N^2)$ because it has a loop within a loop. Considering that Algorithm 4.2 adds another outer loop, the global time complexity is $O(N^3)$.

Algorithm 4.2: HeDPMScan()

$TObj0 \leftarrow HeDPM(0)$
$TObj \leftarrow TObj0$
if $LastIsWorst()$

then
- **while** $LastIsMostLoaded()$ **and** $TObj < 1.5 * TObj0$
  - **then**
    - $TObj \leftarrow TObj + 0.5 * TObj0/20$
    - $HeDPM(TObj)$

else

then
- **while** $LastIsLeastLoaded()$ **and** $TObj > 0.5 * TObj0$
  - **then**
    - $TObj \leftarrow TObj - 0.5 * TObj0/20$
    - $HeDPM(TObj)$

4.1 Case study

In order to have a better understanding of how the algorithm works, we propose to analyze a particular application example. We have eight stages ($s_1, s_2, \ldots, s_8$) and eight processors ($p_1, p_2, \ldots, p_8$) and the following values (the subscript corresponds to the processor/stage id):

1. Processor speed: $s(p) = (27.4_1, 1.0_2, 20.0_3, 23.3_4, 1.0_5, 19.7_6, 15.5_7, 14.1_8)$
2. Stage workload: $w(n) = (6.0_1, 1.3_2, 2.5_3, 4.0_4, 20.3_5, 28.1_6, 19.1_7, 12.8_8)$
3. Stage output data size: $d(n) = (10.0, 10.0, 10.0, \ldots, 10.0, 10.0, 10.0)$
4. Communication bandwidth: $B(p_i, p_j) = 10.0$, except all communications with the first processor $B(p_1, p_i) = B(p_i, p_1) = 5.0$
5. Communication set-up time: $c(p_i, p_j) = 0.1$, except all communications with the first processor $c(p_1, p_i) = c(p_i, p_1) = 0.2$

First, we sort processors and stages depending on their contribution to the production time in Eqs. 3 and 4:

1. $s_{sorted}(p) = (23.3_4, 20.0_3, 19.7_6, 15.5_7, 14.1_8, 27.4_1, 1.0_2, 1.0_5)$
2. $w_{sorted}(n) = (28.1_6, 20.3_5, 19.1_7, 12.8_8, 6.0_1, 4.0_4, 2.5_3, 1.3_2)$

It is noticeable that processor p_1 has the highest speed, but its low communication bandwidth shifted it to the sixth position. Table 2 shows the result of HeDPM after six iterations of executing the second and the third steps of Algorithm 4.1. For each iteration, we present the list of indexes of sorted processors and stages, the optimal production time T_{ideal}, the processor or processors and stage or stages matched (in bold type), and the final production time of the resulting match T_{match}. We can also see that for each iteration we only consider the remaining processors and stages.

For example, in the first iteration of the algorithm T_{ideal} is 3.2478. The algorithm tries a matching of processor p_4 with stage s_6. The processing time of this match (T_{match}) is 3.5459, which would have been the realistic production time. The similarity between T_{match} and T_{ideal} confirms that the matching of processor p_4 with stage s_6 is adequate. In the fifth iteration T_{ideal} is 3.0283 and the best matching (T_{match} is 2.9650) is obtained with the group of stages s_1, s_2 and s_3 mapped to processor p_8. In the sixth iteration, the last one, the stage s_4 is replicated on processors p_1, p_2 and p_5.

It can be seen that the smallest T_{match} is obtained in the last iteration (6). This means that processors p_1, p_2, and p_5 are underloaded, which produces a load unbalance.

Table 2 HeDPM application example (bold type indicates processor to stage matching)

Iter	Processor list	Stage list	T_{ideal}	Matching	T_{match}
1	(**4**, 3, 6, 7, 8, 1, 2, 5)	(**6**, 5, 7, 8, 1, 4, 3, 2)	3.2478	$p_4 s_6$	3.5459
2	(**3**, 6, 7, 8, 1, 2, 5)	(**5**, 7, 8, 1, 4, 3, 2)	3.1811	$p_3 s_5$	3.2849
3	(**6**, 7, 8, 1, 2, 5)	(**7**, 8, 1, 4, 3, 2)	3.1396	$p_6 s_7$	3.2395
4	(**7**, 8, 1, 2, 5)	(**8**, 1, 4, 3, 2)	3.0726	$p_7 s_8$	3.0258
5	(**8**, 1, 2, 5)	(**1**, 4, **3**, **2**)	3.0283	$p_8 s_1 s_2 s_3$	2.9650
6	(**1**, **2**, **5**)	(**4**)	–	$p_1 p_2 p_5 s_4$	2.1944

Therefore, Algorithm 4.2 is used to improve the computation balance. It executes Algorithm 4.1 lowering the $ProductionTimeObjective$ until the smallest T_{match} is not obtained in the last iteration. In this case, when this condition is met the total production time increases to 5.08 and, consequently, the algorithm concludes that the initial mapping is the best it can obtain.

4.2 Dynamic tuning

As we have commented before, the proposed algorithm for balancing pipeline applications can be used in a dynamic tuning environment for improving the execution time of such applications at run-time. In this case, we can use MATE [12], a tool that performs online monitoring, analysis, and tuning of parallel applications. First, at run-time MATE instruments the application to gather information about its behavior. During the analysis phase MATE receives events, searches for bottlenecks and specifies solutions for solving the performance problems encountered. Finally, the application is dynamically modified by applying the given solutions.

MATE is composed of the following modules which cooperate to control and improve the application performance:

- The Application Controller (AC) is a daemon that controls the execution and the dynamic instrumentation of each individual application process.
- The Analyzer is a centralized process that carries out the application performance analysis and decides on monitoring and tuning. It automatically detects existing performance problems on the fly and requests appropriate changes to improve the application performance.
- The Dynamic Monitoring Library (DMLib) is a shared library that is dynamically loaded by the AC in the application tasks to facilitate collecting data and delivering it to the Analyzer.

Performance models constitute the knowledge used by MATE to conduct the performance analysis process. Each performance model is encapsulated in MATE in a piece of software called a tunlet. Each tunlet implements the logic to overcome a particular performance problem by encapsulating information concerning: (a) measurement points, where to insert instrumentation in the target application to gather performance information, (b) performance functions, which are a set of expressions that model the application behavior, and (c) tuning points, which are the points of the applications that can be changed by a tuning action to improve its performance.

Taking into account the knowledge required by MATE for tuning and the proposed performance model for pipeline applications, we can define all necessary information for developing the corresponding tunlet (Table 3).

The detailed design and implementation of this tunlet is out of the scope of this paper.

Table 3 Definition of the knowledge required by MATE

Measurement points	$w(n)$: Execution time of the computation function of each stage (instrumenting the entry and exit points of the function)
	$d(n)$: Size of output message of each stage (instrumenting the corresponding MPI_Send call)
Performance functions	Algorithm 4.1
	Algorithm 4.2
Tuning points	Mapping of stages

5 Experimental assessment

After the presentation of the HeDPM algorithm, we assess its behavior. First we present a comparison with the optimal solution based on a brute force approach. Second we compare our algorithm with BSC algorithm [8] to have a comparison with other relevant works, and we also compare it with our previous algorithm (DPM [26]) to show the impact of the contributions presented in this work. Finally, we provide an application example with an MPI implementation of the Ferret pipeline included in the PARSEC benchmark suite [11]. This way, we go beyond synthetic applications and illustrate the quality of HeDPM results on a real application.

5.1 Algorithm for searching the optimal mapping

The optimal solution to the problem of mapping the N stages of a pipeline in P heterogeneous processors can be found if the production times of all mappings are known. However, the cost of this solution is unaffordable because the complexity of this problem is NP-Hard. Nevertheless, we will use the optimal mapping algorithm defined in this section with the aim of assessing the results given by HeDPM for a small number of processors and stages because it is only in this situation when the number of calculations can be afforded. The number of stages and processors selected for this test is 4.

We have developed a brute force searching algorithm. The input parameters are the number of processors (P), the number of stages (N), processor speeds (s), stage workloads (w), size of output data (d), bandwidth (B), and communication set-up time (c). The algorithm outputs the optimal solution by generating all possible mappings, evaluating the production time of every mapping, and returning the one with the lowest production time.

Figure 1 shows an histogram of the differences of HeDPM results over the optimal values for 100 pipeline samples for a number of processors and stages of 4. For every pipeline sample the processor speeds and the stage loads are generated randomly with statistic parameters $\mu = 10$ and $\sigma = 5$. Communication load is also randomly obtained but with a weight ten times lower. It can be seen that the majority of values are in the 20 and 40% centered bins, and the average distance is 40.8%. Taking into account that the HeDPM algorithm has a much more lower complexity, and considering a trade-off

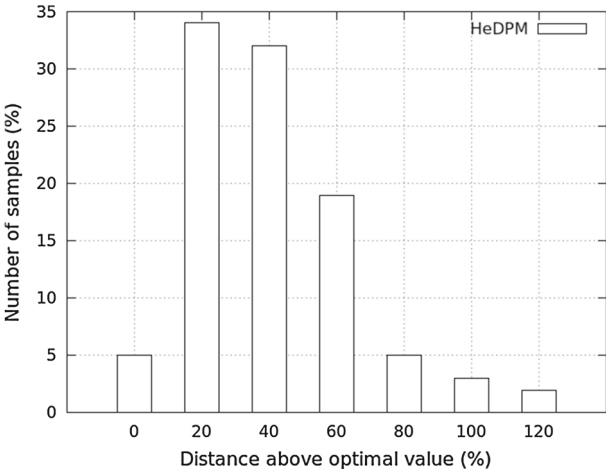

Fig. 1 Difference between HeDPM and the optimal mapping

between complexity and optimization, the HeDPM is a good option to be implemented in a run-time optimization platform.

5.2 Comparison with BSC and DPM

Comparing with the optimal solution has the disadvantage that it is only possible for a few processors and stages. To have a wider analysis, we present a comparison with BSC algorithm using the test described in [8]. The characteristics of this test are: 90 simulated cases parametrized by the number of processors (P) from 10 to 100; the number of stages (N) is 30 for all cases. Processors speed and stage workload are generated randomly, and homogeneous and heterogeneous communications have been considered generating two scenarios (homogeneous, heterogenous). For each of these cases we have taken 100 pipeline samples to run both HeDPM and BSC algorithms.

We have tested all the heuristics presented in [8], and we have selected the BSC algorithm because it has been the most effective one. Essentially, this algorithm performs a binary search on the pipeline production time. For a given production time, it studies whether there is a feasible solution, starting with the first stage and building groups to fit on processors, it checks whether it is possible to map all stages in the available processors with their associated production time below or equal to the given production time. This approach is similar to our HeDPM in that it also improves the application's performance by gathering fast consecutive stages in the same processor, but it is different in that we have added the replication of slowest stages.

Figure 2 shows the results of the homogeneous communication scenario. Three regions are clearly identified: from 10 to 30 processors BSC is the best; from 30 to 60 processors both algorithms have similar results; and from 60 processors HeDPM is the best. This different behavior is caused by the effect of replication incorporated

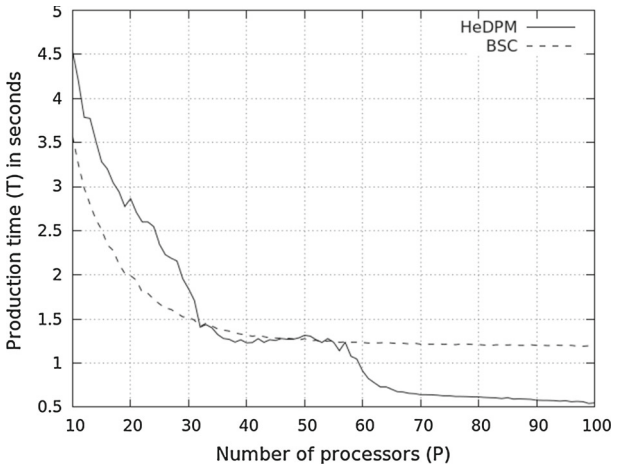

Fig. 2 Production time of BSC and HeDPM in homogeneous communications

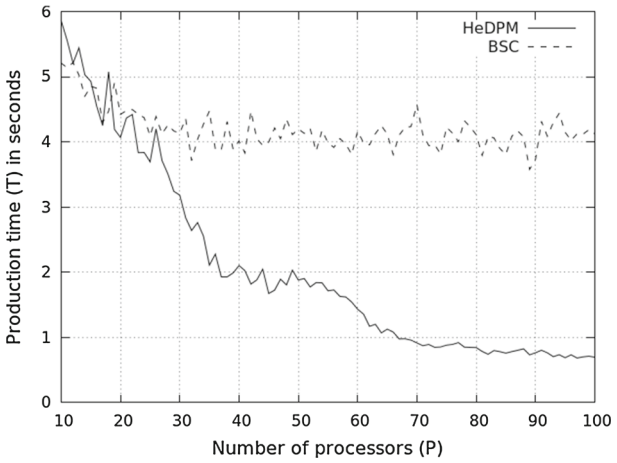

Fig. 3 Production time of BSC and HeDPM in heterogeneous communications

to HeDPM, whose positive effect increases when the number of processors surpasses the number of stages.

Figure 3 shows the results of the heterogeneous communication scenario. In this case the capacity of HeDPM to manage heterogeneous communication explicitly leads to consistently better results, and again the difference is greater when the number of processes increases.

Overall, HeDPM obtains better results than BSC, except for the case of a low ratio of processors to stages in conjunction with homogeneous communication, which would be rather uncommon for current systems. In addition, the complexity of HeDPM ($O(N^3)$) is lower than the one of BSC ($O(N^5)$), which makes it more convenient for dynamic tuning.

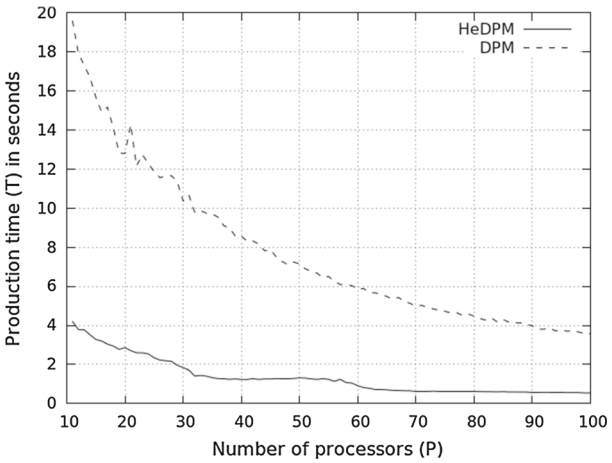

Fig. 4 Production time of DPM and HeDPM

Finally, we show a comparison of HeDPM with DPM [26,27], our previous algorithm for homogeneous systems. Results are shown in Fig. 4 and they have been obtained in the same conditions as results of Fig. 3. It can be seen that HeDPM is producing significantly better results than DPM, demonstrating the importance of taking into consideration communication and computation capacity heterogeneity.

5.3 Application example: Ferret

Finally, we show the algorithm behavior for a concrete application. We have used the content-based similarity image search Ferret application included in the PARSEC benchmark suite [11]. Originally, it was designed for a shared memory machine, but we have modified the code to use MPI for a distributed memory machine and to configure replicas and groups of stages. Ferret is parallelized using the pipeline structure with six stages. The first and the last stage, *Image load (Load)* and *Result output (Out)*, are serial stages. The first stage reads a set of images to be analyzed against a database and the last one outputs the list of names of similar images. The middle four stages are:

- *Image segmentation (Seg)* The image is decomposed into separate areas, which display different objects. These areas are called segments.
- *Feature extraction (Ext)* A feature vector is extracted from every segment. This feature vector is a multi-dimensional mathematical description of the segment contents that encodes fundamental properties such as color, shape and area.
- *Indexing (Ind)* This stage and the next one query the image database to obtain a candidate set of images. The database is organized as a set of hash tables that are indexed with multi-probe locality-sensitive hashing (LSH) [23]. It uses hash functions that map similar feature vectors to the same hash bucket with high probability. The indexing stages gives a probing sequence that is considered to have a high probability for finding a candidate image in a bucket.

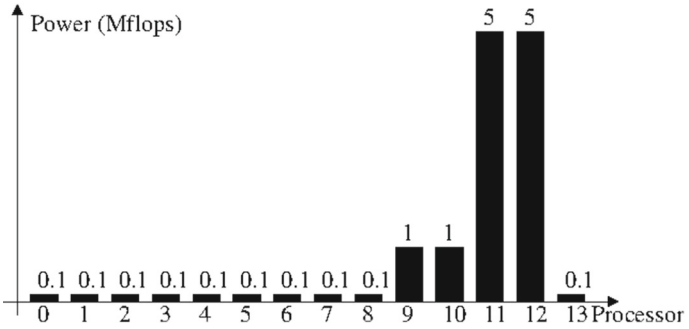

Fig. 5 Processor performance of the test system

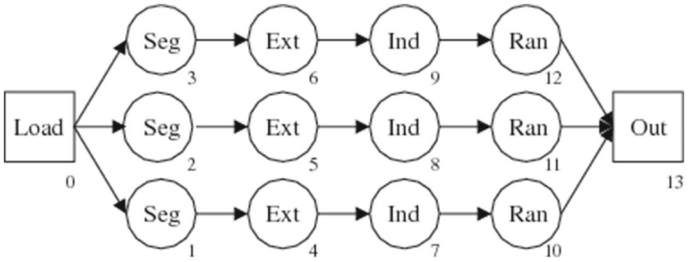

Fig. 6 Ferret pipeline application in an orderly assignation of processors to stages

– *Ranking (Ran)* A detailed similarity estimate is computed and images are ordered according to their calculated rank.

We have selected SimGrid [33] as a platform to run Ferret application. SimGrid is a scientific simulator to study the behavior of large-scale distributed systems, it is an open source project and it is widely used by the research community. It was developed in the USS-SimGrid and SONGS scientific projects funded by the ANR French research funding agency, and it has the support of Inria [20].

We have carried out a test that consists of searching some images of a video sample in an image database. The image database has 3500 images from the native input data of Ferret. Then, we have transformed a video sample with 7 shots into a sequence of images, in concrete 700 images. The heterogeneous test system has 14 processors with the processor performance indicated in Fig. 5. We have run the Ferret application in two scenarios. The first one with an orderly assignation of stages to processors. Figure 6 shows this mapping, the numbers identify which processor runs each pipeline stage. The second scenario has been created extracting the HeDPM input parameters and executing the algorithm to obtain a new mapping. Figure 7 shows this new mapping. The total execution time of both scenarios are presented in Table 4. It can be concluded that the improvement obtained with HeDPM (i.e., speedup of 9.2x) in that application is very remarkable.

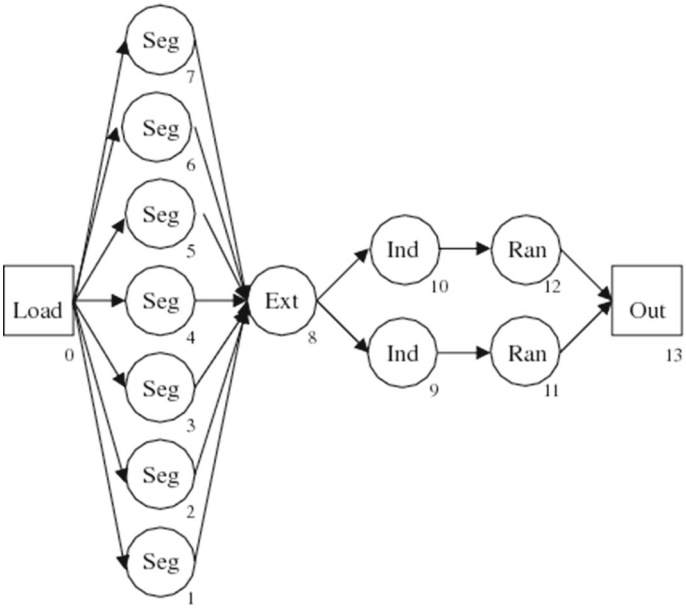

Fig. 7 Ferret pipeline application with the assignation of processors to stages from HeDPM

Table 4 Simulaton results of Ferret application in SimGrid platform

Scenario	Total execution time (s)
Orderly assignation	12.243
HeDPM assignation	1.330

6 Conclusions

We have proposed a new approach that can be applied to dynamically improve the performance of heterogeneous pipeline applications. The resulting algorithm, which has been called Heterogeneous Dynamic Pipeline Mapping (HeDPM), improves the application's throughput by gathering the pipeline's fastest stages and replicating the slowest ones. The algorithm is based on simple analytical models of the pipeline stages behavior. The complexity of this new approach is far lower than the optimal one, and also lower than the Binary Search Closest (BSC) algorithm [8]. We have presented a wide set of results: a comparison with an optimal algorithm for small values of stages and processors, a general comparison with the Binary Search Closest (BSC) algorithm [8], and an application example with the Ferret pipeline included in the PARSEC benchmark suite [11] that allows us to observe the HeDPM behavior in a real application and to analyze its performance.

We conclude that the HeDPM-proposed mappings usually lead to significant performance improvements that justify the effort of dynamically implementing the necessary changes. For example, in the case of Ferret application, HeDPM mapping is leading to a 9.2x improvement over the original mapping.

The new approach outperforms DPM [26] by a factor of at least 5x, demonstrating the importance of explicitly taking heterogeneity into consideration. Compared to BSC (to the best of our knowledge one of the best comparable algorithms), HeDPM shows significantly better results when the number of available processors is greater than the number of stages and communication is highly heterogeneous. However, in the case of scarce resources (number of processors $<$ number of stages), BSC behaves better than HeDPM. In addition, HeDPM complexity, also lower than the one of other algorithms, makes it suitable for being used in dynamic tuning tools.

Future work will include an implementation of a MATE tunlet and an extension of the scope of the pipeline model to consider pipelines that are not linear.

Acknowledgements The comparison with the Binary Search Closest (BSC) algorithm has been possible thanks to the available code on the web [8]. The application example with the Ferret pipeline has been possible thanks to the PARSEC benchmark suite [11].

References

1. Almeida F, González D, Moreno LM, Rodríguez C (2002) An analytical model for pipeline algorithms on heterogeneous clusters. In: Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp 441–449. Springer, London, UK
2. Arabnia HR (1990) A parallel algorithm for the arbitrary rotation of digitized images using process-and-data-decomposition approach. J Parallel Distrib Comput 10(2):188–192
3. Arabnia HR, Oliver MA (1986) Fast operations on raster images with simd machine architectures. Comput Graph Forum 5(3):179–188
4. Augonnet C, Thibault S, Namyst R, Wacrenier PA (2009) StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Springer, Berlin
5. Benkner S, Pllana S, Traff JL, Tsigas P, Dolinsky U, Augonnet C, Bachmayer B, Kessler C, Moloney D, Osipov V (2011) PEPPHER: efficient and productive usage of hybrid computing systems. IEEE Micro 31:28–41
6. Benoit A, Çatalyürek ÜV, Robert Y, Saule E (2013) A survey of pipelined workflow scheduling: models and algorithms. ACM Comput Surv CSUR 45(4):50
7. Benoit A, Cole M, Gilmore S, Hillston J (2004) Evaluating the performance of skeleton-based high level parallel programs. In: The International Conference on Computational Science (ICCS 2004), Part III, LNCS, pp 299–306. Springer
8. Benoit A, Robert Y (2008) Mapping pipeline skeletons onto heterogeneous platforms. J Parallel Distrib Comput 68(6):790–808
9. Bhandarkar SM, Arabnia HR (1995) The hough transform on a reconfigurable multi-ring network. J Parallel Distrib Comput 24(1):107–114
10. Bhandarkar SM, Arabnia HR (1995) The refine multiprocessor? Theoretical properties and algorithms. Parallel Comput 21(11):1783–1805
11. Bienia C (2011) Benchmarking modern multiprocessors. Ph.D. thesis, Princeton University
12. Caymes-Scutari P, Morajko A, Margalef T, Luque E (2010) Scalable dynamic monitoring, analysis and tuning environment for parallel applications. J Parallel Distrib Comput 70(4):330–337
13. Cesar E, Moreno A, Sorribes J, Luque E (2006) Modeling master/worker applications for automatic performance tuning. Parallel Comput 32(7):568–589

14. do Nascimento LT, Ferreira RA, Guedes D (2005) Scheduling data flow applications using linear programming. In: Proceedings of the 2005 International Conference on Parallel Processing, ICPP '05, pp 638–645. IEEE Computer Society, Washington, DC, USA
15. Gilmore S, Hillston J, Kloul L, Ribaudo M (2003) PEPA nets: a structured performance modelling formalism. Perform Eval 54(2):79–104
16. Goli M, Garba MT, Gonzalez-Velez H (2012) Streaming dynamic coarse-grained CPU/GPU workloads with heterogeneous pipelines in FastFlow. In: Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication and 2012 IEEE 9th International Conference on Embedded Software and Systems, HPCC '12, pp 445–452. IEEE Computer Society, Washington, DC, USA
17. González-Vélez H, Cole M (2010) Adaptive structured parallelism for distributed heterogeneous architectures: a methodological approach with pipelines and farms. Concurr Comput Pract Exper 22(15):2073–2094
18. Guirado F, Ripoll A, Roig C, Luque E (2005) Exploitation of parallelism for applications with an input data stream: optimal resource-throughput tradeoffs. Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on pp 170–178
19. Hoang PD, Rabaey J (1993) Scheduling of DSP programs onto multiprocessors for maximum throughput. IEEE Trans Signal Process 41(6):2225–2235
20. Inria: Inria: Inventors for the digital world. http://www.inria.fr/en/
21. Kijsipongse E, Ngamsuriyaroj S (2010) Placing pipeline stages on a grid: single path and multipath pipeline execution. Future Gener Comput Syst 26:50–62
22. Lin CS, Lin CS, Lin YS, Hsiung PA, Shih C (2013) Multi-objective exploitation of pipeline parallelism using clustering, replication and duplication in embedded multi-core systems. J Syst Archit 59(10):1083–1094
23. Lv Q, Josephson W, Wang Z, Charikar M, Li K (2007) Multi-probe LSH: efficient indexing for high-dimensional similarity search. In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp 950–961. ACM
24. Miceli R, Civario G, Sikora A, César E, Gerndt M, Haitof H, Navarrete CB, Benkner S, Sandrieser M, Morin L, Bodin F (2012) Autotune: a plugin-driven approach to the automatic tuning of parallel applications. In: Applied Parallel and Scientific Computing—11th International Conference, PARA 2012, Helsinki, Finland, June 10–13, 2012, Revised Selected Papers, pp 328–342
25. Morajko A, Cesar E, Caymes-Scutari P, Margalef T, Sorribes J, Luque E (2005) Automatic tuning of master/worker applications. In: Euro-Par 2005 Parallel Processing, Lecture Notes in Computer Science, 3648, pp 95–103
26. Moreno A, Cesar E, Guevara A, Sorribes J, Margalef T (2012) Load balancing in homogeneous pipeline based applications. Parallel Comput 38(3):125–139
27. Moreno A, Cesar E, Guevara A, Sorribes J, Margalef T, Luque E (2008) Dynamic Pipeline Mapping (DPM). In: LNCS, vol 5168, pp 295–304
28. Pinar A, Kartal Tabak E, Aykanat C (2008) One-dimensional partitioning for heterogeneous systems: theory and practice. J Parallel Distrib Comput 68(11):1473–1486
29. Rosas C, Sikora A, Jorba J, Moreno A, Espinosa A, César E (2014) Dynamic tuning of the workload partition factor and the resource utilization in data-intensive applications. Future Gener Comput Syst 37:162–177
30. Salawdeh I, Morajko A, César E, Margalef T, Luque E (2009) Automatic performance tuning of parallel mathematical libraries. In: Parallel Computing: From Multicores and GPU's to Petascale, Proceedings of the Conference ParCo 2009, 1–4 September 2009, Lyon, France, pp 407–414
31. Sanchez D, Lo D, Yoo RM, Sugerman J, Kozyrakis C (2011) Dynamic fine-grain scheduling of pipeline parallelism. In: Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques, PACT '11, pp 22–32. IEEE Computer Society, Washington, DC, USA
32. Sikora A, César E, Ureña IAC, Gerndt M (2016) Autotuning of MPI applications using PTF. In: Proceedings of the ACM Workshop on Software Engineering Methods for Parallel and High Performance Applications, Kyoto, Japan, May 31–June 04, 2016, pp 31–38
33. SimGrid: Versatile simulation of distributed systems. http://simgrid.gforge.inria.fr
34. Spencer M, Ferreira R, Beynon M, Kurc T, Catalyurek U, Sussman A, Saltz J (2002) Executing multiple pipelined data analysis operations in the grid. In: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, Supercomputing '02, pp 1–18. IEEE Computer Society Press, Los Alamitos, CA, USA

35. Subhlok J, Vondran G (2000) Optimal use of mixed task and data parallelism for pipelined computations. J Parallel Distrib Comput 60(3):297–319
36. Wani MA, Arabnia HR (2003) Parallel edge-region-based segmentation algorithm targeted at reconfigurable multiring network. J Supercomput 25(1):43–62
37. Yang M-T, Kasturi R, Sivasubramaniam A (2003) A pipeline-based approach for scheduling video processing algorithms on NOW. Trans Parallel Distrib Syst 14(2):119–130