WSCOM: Online task scheduling with data transfers

Jean-Noel Quintin, Frédéric Wagner

▶ **To cite this version:**

Jean-Noel Quintin, Frédéric Wagner. WSCOM: Online task scheduling with data transfers. [Research Report] RR-7792, INRIA. 2011, pp.15. hal-00639922

HAL Id: hal-00639922
https://inria.hal.science/hal-00639922v1

Submitted on 10 Nov 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

WSCOM: Online task scheduling with data transfers

Jean-Noël Quintin , Frédéric Wagner

RESEARCH REPORT

N° 7792

November 2011

Project-Team MOAIS

ISSN 0249-6399 ISRN INRIA/RR--7792--FR+ENG

WSCOM: Online task scheduling with data transfers

Jean-Noël Quintin , Frédéric Wagner

Project-Team MOAIS

Research Report n° 7792 — November 2011 — 12 pages

Abstract: In our paper we consider the on-line problem of tasks scheduling with communication. All information on tasks and communication are not available in advance except the DAG of task topology. We take a novel approach by considering the bi-objective problem where we try to minimize both completion time and the number of data transmissions.
We propose a new variation of the work-stealing algorithm: WSCOM. We try here to take advantage of the information of the DAG topology, and improve locality by clustering the tasks together. We propose several variants designed to overlap communication or optimize the graph decomposition.
Performance is evaluated by simulation and we compare our algorithms with off-line list-scheduling algorithms from the literature. These experiments validate the different design choices taken. In particular we show that WSCOM is able to achieve performance closes to off-line algorithms in most cases and is even able to achieve *better* performance in the event of congestion due to less data transfer.

Key-words: load-balancing; online-scheduling; data transfers; work-stealing;

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

WSCOM: Ordonnancement de tâches en-ligne pour des applications traitant des données volumineuses

Résumé : Dans ce papier, nous considérons l'ordonnancement de tâches en-ligne pour des applications traitant des données volumineuses. Pour ces applications, le temps d'exécution et les quantités de données transférées ne sont pas connues. Ainsi, le problème d'ordonnancement qui est déjà NP-Complet avec ces informations, est vraiment difficile. Ainsi pour s'abstraire de ces informations, nous considérons une nouvelle approche du problème, en le modélisant par un problème bi-objectifs. Le premier objectif est l'équilibrage de la charge, et le second la minimisation de nombre de transferts effectués. Pour réaliser le second objectif, l'algorithme a accès à la structure du DAG.

Nous proposons un nouvel algorithme basé sur le vol de travail: WSCOM. Cet algorithme tente d'utiliser la structure du DAG pour réduire le nombre de communication. Dans les différentes variantes proposées, nous nous intéressons aussi au recouvrement des communications au sein de WSCOM.

Pour évaluer les performances de ces algorithmes, nous comparons WSCOM aux algorithmes hors-ligne. Cette comparaison est bien-sûr au désavantage de notre algorithme car les algorithmes hors-ligne ont accès à l'ensemble des informations de l'application. Malgré ce net désavantage lors qu'il y a de la congestion au niveau du réseau, WSCOM obtient de meilleures performances que les algorithmes hors-ligne.

Mots-clés : équilibrage de charge, ordonnancement en-ligne, transferts de données, vol de travail

I. Introduction

In our paper we take interest in the automatic parallelization of the execution of *make* commands. *GNU make* is a widely used program allowing the description of tasks (known as targets) and dependencies among them. While being used mainly as a software development tool to automate compilation, it is not uncommon to see *makefiles* for many different kinds of applications. As an example, *make* is often used as a way to achieve non-regression testing since some tests might depend on successful completion of previous ones. In fact it is even possible to use a Makefile as a way to describe a coarse grained parallel application.

Our team has developed a new tool called *DSMake*. This tool distributes the execution of a makefile by scheduling the tasks on a distributed platform. Its goal is to minimize the global completion time denoted as $C_{\max}$. Achieving this requires an efficient scheduling algorithm. The scheduling problem is difficult for two main reasons. First, as the files sizes might be relatively large we take communication into account. Secondly, we have a non-clairvoyant setting: task sizes and communication are not known in advance, nor the network topology.

We present in the Section II existing algorithms from the literature solving the off-line problem for the case where no network congestion can occur. We present schedule examples, showing that existing algorithms may generate a high amount of communication, and as such are more sensitive to bandwidth changes.

Section III presents *WSCOM*, a variation of the classical work-stealing algorithm [1] where we use a bi-objective approach; we try here to minimize both $C_{\max}$ and the amount of communication. The increased locality of computation achieved by decreasing the number of communication is used as a way to increase performance even without knowledge of network topology.

Section IV presents experimental comparisons using the Simgrid simulator.

We then conclude on the obtained results in Section V.

II. Scheduling with Communication

This section presents existing works for the off-line load-balancing of a data-intensive application on p processors. Classically, this problem is described by the three-field notation: $Q|prec, c, p_i|C_{\max}$. As input, we consider p heterogeneous processors and a DAG representing tasks dependencies, tasks execution times and communication costs. The aim is to minimize the total completion time.

This problem is NP-Hard and a 5/4-inapproximability has been proved for the particular case $P|prec, c = 1, p_j = 1|C_{\max}$ by Hoogeveen *et al* [2].

We can classify heuristics from the literature into severals categories. Some of them group tasks in task clusters. The main idea of such heuristics is to avoid communication by grouping communicating tasks on common resources. Algorithms working with task clustering appears in the Section II-A.

On the opposite, Section II-B shows some heuristics based on list scheduling. The main objective is then to avoid idle times more than to reduce communication.

A. Task Clustering

While scheduling tasks with communication it can be expected that too large communication can prevent to obtain an acceptable completion time. To avoid sending such data, some heuristics group tasks in task clusters where all tasks from a given cluster are executed on the same machine. Such algorithms start by computing a clustering and in a second phase assign all clusters to the different computing resources.

The main idea used to compute clusters is that while computing an optimal schedule for $P|prec, c = 1, p_j = 1|C_{\max}$ is NP-Hard, the problem becomes easy for an unbounded number of processors. It is therefore possible to assess the communication effect on the optimal schedule for unbounded resources.

For example, Sakkar [3] proposed a task-clustering algorithm called EZ (Edge-Zeroing). This algorithm sorts the edges in decreasing order of communication costs. For each edge in this order, the parallel completion time is computed with a cost equal to zero for the chosen edge. If the parallel completion time is lower than before, then nodes or task clusters which are linked by this edge, are put in the same task cluster.

The schedule is finally completed by assigning the clusters to the available computing resources.

Other algorithms from the literature following similar principle provide bounds on the schedule length and reduce the algorithm cost: MD [4] DCP [5] DL [6] DSC [7].

Of all these heuristics, DSC is known for giving optimal schedules on Fork and Join DAG (with infinite number of processors). Moreover, this heuristic proposes a mapping of clusters on P processors in a software called PYRROS [8].

We present here two examples illustrating two different problems affecting performance of clustering algorithms.

Figure 1 highlights with an example, that large clusters can prevent to obtain an optimal schedules even on a fork DAG. In this example, if we try to schedule directly task clusters, the obtained schedule has a length equal to $4 + 3 * \epsilon$, while the optimal schedule has a length equal to $3 + 3 * \epsilon$. In addition the optimal solution for this example requires Cluster 1 to be split and scheduled on two different machines.

Figure 2 introduces another example suffering from the opposite problem. This example highlights that the schedule proposed by PYRROS does not take into account the communication. The PYRROS schedule transfers 5 data while the optimal solution transfers 2 data. The problem in this case appears because the clusters obtained are too

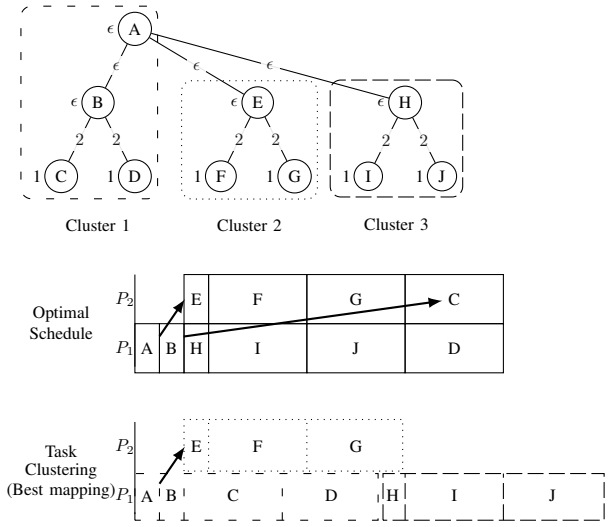


Figure 1. Task clustering vs optimal schedule

small. Therefore scheduling the clusters on the resources can require large communication times.

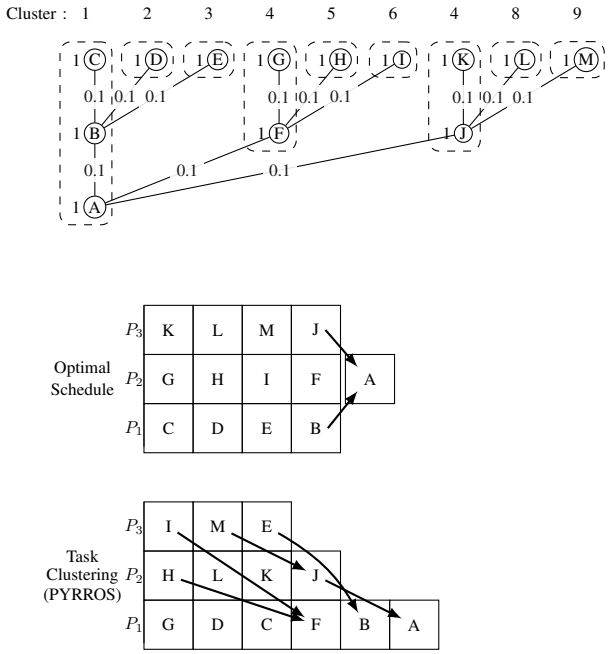


Figure 2. Task clustering (PYRROS) vs optimal schedule

These two simple examples show that it might be difficult to map clusters on resources because the clustering is not computed according to available resources. Moreover these heuristics are limited to the homogeneous problem.

Our examples also illustrates that it might be of interest to provide a recursive clustering of the tasks. Such a clustering would grant more freedom during the mapping phase by allowing joining or breaking clusters on demand. We take advantage of this idea in the algorithms of the Section III.

B. List Scheduling

List-scheduling algorithms rely on a list of ready tasks, eventually sorted by classical criteria like task top level, task bottom level, task completion time or communication cost. The principle is the following: each time a machine becomes idle, it starts executing the first task in the list. All list algorithms provide the good property to achieve an approximation ratio of 2 [9] for the $P|prec, p_i|C_{\max}$ problem (where communication are not taken into account).

Many variants of list-scheduling algorithms exist in the literature. In our paper, we focus more particularly on the most ones commonly used: HEFT [10], CPOP [10], BIL [11], MinMin [12], MaxMin [12], sufferage [12] and HBMCT [13].

While these heuristics usually result in good performance, they might generate an important amount of communication. Figure 3 presents the amount of data transfer of several list schedules (HEFT, MinMin and MaxMin), which give the same schedule for the considered example. Although these schedules have an optimal execution time in this case, the amount of data transfer is significantly higher than the optimal solution. This effect can therefore have an important impact on real world use because low network bandwidths will directly affect performance.

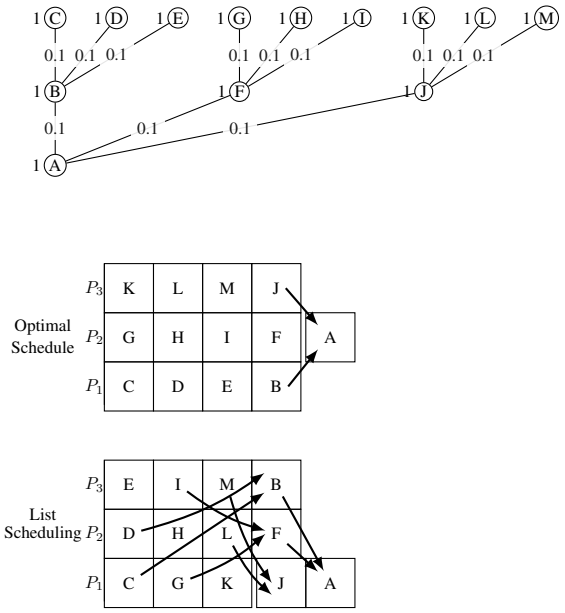


Figure 3. List schedules (HEFT, MinMin, MaxMin) compared to an optimal schedule

III. WSCOM: Work-stealing with communication on general DAG

We now consider an on-line version of the problem $Q|prec, c, p_i|C_{\max}$. This problem corresponds to the real problem of tasks scheduling for DSMake.

We make the following assumptions:

- Tasks processing times are unknown
- Network topology is unknown
- Data sizes are unknown
- Application DAG is known in advance

Most of these assumptions are pretty common: it is often difficult to know processing times in advance and this is particularly true for DSMake as the application is provided by the user. Communication times are very difficult to predict as no information on the network is available and moreover the network might be shared by several users.

Notice that in our model we know in advance the DAG of tasks. This property stems from our use of DSMake. In practice, the whole DAG is described by users in the Makefile before execution start. We intend to take advantage of this knowledge to achieve efficient schedules.

While it might seem difficult to obtain performance with so many unknowns we can rely on work-stealing algorithm as an on-line distributed list-scheduling algorithm achieving good schedules even with unknown processing times.

Section III-A introduces the classical work-stealing algorithm. This algorithm is then modified to take advantage of the additional information on the DAG structure. Section III-B introduces *WSCOM*, a new variation of the work-stealing algorithm intended to reduce the communication effect on join DAG. This algorithm is then extended to general DAG Section III-C.

Finally Section III-D describes in more details several possibilities to achieve the communication.

A. Work-Stealing

Blumofe and Leiserson [1] have introduced an on-line dynamic scheduling algorithm providing good execution times while being fully decentralized. Each time a processor becomes idle it sends a steal request to another one. Each processor keeps a stack of tasks to execute and eventually provides some to others. Different versions of the work-stealing algorithm exist, by refining the choices of the stolen processor, the stolen task and the local execution order.

In [14] Arora *et al* bound the number of steal requests by $O(pD)$ and the execution time by $W/p + O(D)$ where p is the number of processors, D the critical path and W the total work. For this proof, the stolen processor is chosen randomly with a uniform probability. At each steal, only the oldest task is stolen; the local task execution order follows the sequential order.

The aim of each steal is to balance the load between both processors. Stealing half of the work on the target processor has been shown to be efficient in [15] [16]. In practice only the oldest task is stolen because this task generally represents a significant amount of work on the target processor. This property derives from the fact that tasks are created recursively. Several libraries implement the work-stealing algorithm like Cilk [18], Kaapi [19], Satin [20], TBB [21], X10 [22].

Moreover, all these libraries are not directly suited for our problem as the DAG is discovered at runtime since tasks are created recursively.

For example, the Cilk language provides the keywords spawn and sync. The programmer has to describe how the work is recursively divided into smaller and smaller tasks. And There are no way to describe some dependencies among severals tasks created in different parts of the program. Also with these keywords, the programmer is restricted to fork-join DAG.

On the opposite, for DSMake, no tasks are created recursively. All tasks are known in advance together with all dependencies.

B. WSCOM on join DAG

While our WSCOM algorithm is working on general DAG, we initially present the main idea of the algorithm on the special case where the input graph is a join DAG, i.e. the outgoing degree of vertices is bounded by one and there is only one leaf. On such graphs, the complexity of WSCOM is reduced and the algorithm easier to understand.

We basically rely on two different ideas.

First, it seems difficult to manage communications while it is impossible to know in advance their sizes. In the event of very large communication, the execution should obviously be sequential and the other extreme case will require dispatching tasks on the largest number of machines. We avoid this difficulty by switching to a bi-objective problem. Our primary objective is to minimize the execution time without communications and our secondary objective is to minimize the total amount of communication. Things should now be much easier since we know the DAG in advance and we are interested in the *number* of communications and not their *time*. Of course, this change of objectives might impact the real completion time. We intuitively hope that a reduction in the amount of communication will imply a reduction in communication times and this assumption will be validated experimentally in Section IV.

The second important idea is to combine clustering (as presented in Section II-A) and work-stealing algorithm to achieve performance for our both objectives. The work-stealing schedule will provide a guarantee on the completion time without communications while the clustering part of the algorithm will impact the overall amount of communication.

To achieve the clustering and to provide recursive task creation we add some new virtual tasks to the DAG. *Fork* tasks require no computations but generate during theirs

execution other depending tasks on the local stack. Initially only one fork task is available and this task will recursively create all real tasks to execute. What is more, the recursive splitting is allowing us to cluster the tasks.

To optimize communication we take advantage of our knowledge of the DAG topology. To the initial join DAG, we add a fork-DAG built by symmetry as illustrated Figure 4. The fork-DAG is identical to the task DAG with inversing the edge orientation. Moreover, an edge between the fork task and its symmetrical node is added. If we take for example an execution on 2 processors we end up with the following situation: Initially only one task f_A exists and is located on p_1 (first processor). p_1 executes it and adds to its stack f_B, f_C, f_D, A. p_2 steals a task from p_1 and ends up with f_D while p_1 executes f_B and generates the underneath tasks. At this point p_1 executes the sub-graph between f_B and B while p_2 executes the sub-graph between f_D and D. We can clearly see that using the symmetry allows us to improve the locality of computations.

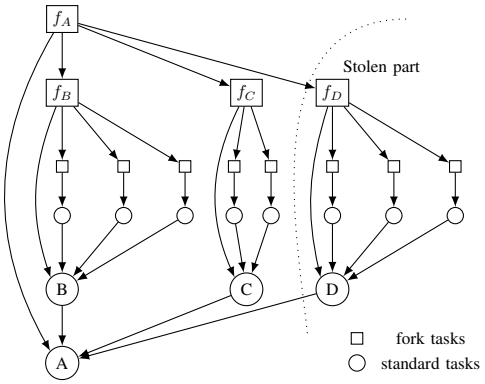


Figure 4. WSCOM DAG using Symmetry

While this algorithm enables us to build a recursive clustering of the tasks, some others options are possible.

For example, a very basic way to cluster recursively all tasks is to build a perfect binary tree of fork tasks on top of all real tasks. This scheme depends on the order of DAG sources. However,since this basic scheme does not take into account dependencies, it might generate an important amount of data transfer.

Another possibility is to use a task clustering algorithm from Section II-A (with no information on tasks sizes and communications sizes) to generate clusters. However, obtaining a recursive decomposition is not straightforward.

C. WSCOM on general DAG

Extending WSCOM to DAG leads quickly to the problem displayed Figure 5. Since A has outgoing edges to B and C, by symmetry, f_A has incoming edges from f_B and f_C. Thus if f_B is executed on processor p_1 and f_C on processor p_2, both processors should contain the task f_A.

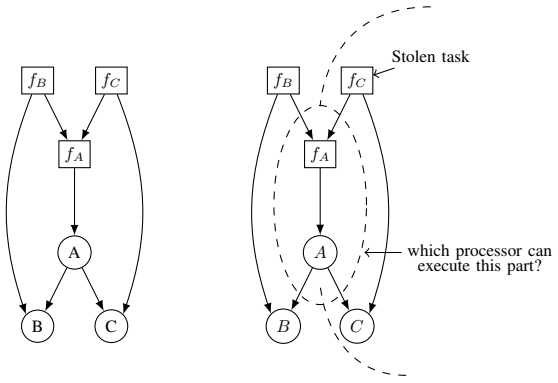


Figure 5. Problem with Outgoing Degree > 1

To solve this problem we need to remove fork edges between tasks such that each task except the initial task can only be forked once. The choices of the edges to keep can however impact performance since they might split the graph in very different ways: inducing more or less communications or generating unbalanced tasks clusters.

Algorithm 1 WSCOM

Require: G(V,E)// Application DAG
Require: S_{tack}// Tasks stack
  $T_{\text{cur}} = VIDE$
  S_{ink}=sink(G) // unique sink in G
  while !is_executed(S_{ink}) **do**
    if $proc\_id = 0$ **then**
      $T_{cur} = fork(S)$; // create a virtual task associated to S.
    end if
    while $T_{\text{cur}} = VIDE$ **do**
      // Check if the task is virtual or not
      if is_fork(T_{cur}) **then**
        // Ask if the task is already forked.
        if ask_fork(T_{cur}) **then**
          for all $I \in$ predecessors(S) **do**
            push(S_{tack},fork(I)); // add the virtual task in the stack.
          end for
          push(S_{tack}, T_{cur})
        end if
      else
        process(T_{cur});
      end if
      $T_{\text{cur}} = pop\_ready(S_{\text{tack}})$ // pop a ready task return VIDE if there is no ready task
    end while
    push($S_{\text{tack}}, steal\_task()$);
    $T_{\text{cur}} = pop\_ready(S_{\text{tack}})$
  end while

We provide two different algorithms solving this problem.

Our first algorithm, WSCOM ST works by building the symmetric graph of the initial task graph and solving the fork requests concurrency problem by arbitrary choosing a spanning tree on the fork requests DAG.

On the opposite we can try to postpone the decision at runtime to take advantage of our partial knowledge of the processing times of the tasks. WSCOM DHT works by keeping *all* edges and allowing a fork to proceed if it is the first time the corresponding task is forked.

To minimize the overhead of this operation while keeping the algorithm decentralized, we advocate the use of a distributed hash table storing for each task a boolean variable indicating whether a previous fork already succeeded. As a side note these tables can also provide an alternate way to update dependencies statuses.

It is difficult at first sight to evaluate which of these two algorithms will lead to better performance. WSCOM DHT provides the advantage to delay choices until more information is available and should therefore induce a better load-balancing but on the other side the DHT requests will incur additional costs. Both algorithms are therefore evaluated independently in our experimental work.

In the following we refer to WSCOM DHT as WSCOM.

D. Data transfer

The last remaining part of the WSCOM algorithm deals with data transfers between two dependent tasks.

Let A and B two tasks such that B depends on the output of A to start. In addition the data cannot be sent until A completes. However, sending the data also requires knowing the machine which will execute B. Since work-stealing algorithm allow un-executed tasks to move among processors, the exact information about the location of B cannot be known before the start of the execution of B.

This limitation is not present for the off-line algorithms since the entire mapping of the tasks is known in advance. Communication of the data of A can in such cases start as soon as A is completed.

It is therefore interesting to bypass these restrictions to start sending as soon as possible. There are mainly two ways to achieve this.

The first solution is to send as soon as possible and in the event of a task migration to re-send the corresponding data. These task migrations add some wasted time for the stolen processor, which has to wait the data transfer as before. These wasted times could involve the critical path. Moreover since a task can only be stolen once, the number of extra communications is limited.

A second approach is to restrict steal requests to fork tasks. Since these tasks require no transfer they do not generate communication overheads. The disadvantage of this method is that since the steal mechanisms are restricted, the overall load balancing might be degraded.

We study the second approach as additional sending of data might impact the length of the critical path and increase congestion. We refer to this algorithm as WSCOM PF (prefetching).

IV. Experimental Analysis

In this section, we validate experimentally the WSCOM algorithm presented in Section III. To obtain meaningful results, we provide comparisons between WSCOM, the different variants proposed and the scheduling algorithms presented in Section II.

Since we intend to simulate communications we rely on the *Simgrid* [23] simulator to achieve simulations where network congestion, bandwidths and latencies can affect the results.

Section IV-A presents details on the chosen configurations and simulation parameters. Simulation results are analyzed in Section IV-B.

A. Experimental Setup

Simulations work in the following way: We generate input graphs randomly or from traces and simulate their execution with different scheduling algorithms using Simgrid on several network topologies (with homogeneous machines). We then compare execution times and communications sizes.

The goals are here to compare the different algorithms and to estimate the bandwidth and networking effect on performance.

1) Input Graphs: Input generation is an important step to obtain meaningful simulation results. As the DAG represents the application, restraining the input DAG to specific graphs might create a bias between the different scheduling algorithms in use.

In our experiments we use two different kinds of graphs. We use on one-hand random graphs, generated by different methods and on the other hand graphs generated from real execution traces.

GGEN [24] is a graph-generation software aiming to incorporate all standard random graphs generation techniques. By using different generators from the literature we hope to achieve fair comparisons of the algorithms.

We choose to use two generation algorithms: TGFF [25] and layer-by-layer [26].

On each DAG, the expected number of nodes is five hundred and tasks processing times are uniformly chosen at random between 7 and 25 seconds. The communication sizes are also uniformly generated. In some experiments they are between 0 and 1 Kilobyte while in other experiments targeting higher communication costs, the sizes are generated between 0 and 1 Gigabytes.

Finally we also consider a graph from a real Makefile.

Kaapi [19] is a middle-ware for parallel computing developed within our team. The Kaapi compilation is relatively

heavy and fully justifies a distributed execution. By instrumenting a sequential execution we obtain the annotated DAG corresponding to the Kaapi compilation. Communication sizes are here lower than 50 Megabytes with 350 tasks, a sequential time of 1475 seconds and a critical path of 50 seconds.

2) Simulated Platforms: The Simgrid simulator allows us to provide a xml description of the platform architecture enabling tests on a wide range of platforms.

We consider two different platform which are chosen relatively simple on purpose as a way to acknowledge and understand the behavior of the different algorithms under controlled conditions.

Our first topology (clique) is a complete graph, which is the topology considered in the list-scheduling algorithms: no congestions occur because no links are shared.

Since the clique topology does not reflect actual networks, we also consider a second topology (cluster) where all computers are connected by one switch. As such a congestion could be obtained if several senders are sending to the same receiver (or vice versa). In our experiments we consider platforms with the processor number comprised between 2 and 50.

Figure 6 illustrates both topologies for five computers.

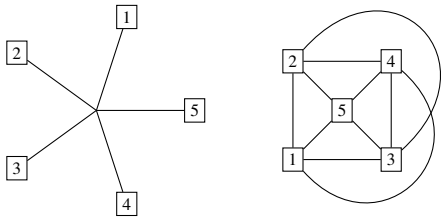


Figure 6. Cluster and Clique platforms

Link capacities are defined with a latency equal to 0.1 millisecond and a bandwidth equal to 1 Gbit per second. Under such parameters the longest possible data transfer (without congestion) is of 10 seconds for a transfer size equal to 1 GB.

Node capacities are homogeneous and set to 3.2 GHz.

B. Experimental Results

We now present results obtained from our set of experiments. For each experiment, we consider a set of input graphs (randomly generated or from traces) and execute different scheduling algorithms with different computing resources. When using TGFF we consider the average results over 400 random graphs and 100 graphs for layer-by-layer (which requires less parameters).

Each curve displays on the x-axis the number of processors available on the platform and on the y-axis the resulting execution time or number of transfers. Trust intervals are not displayed on these curves as the variations on the obtained results are minimal.

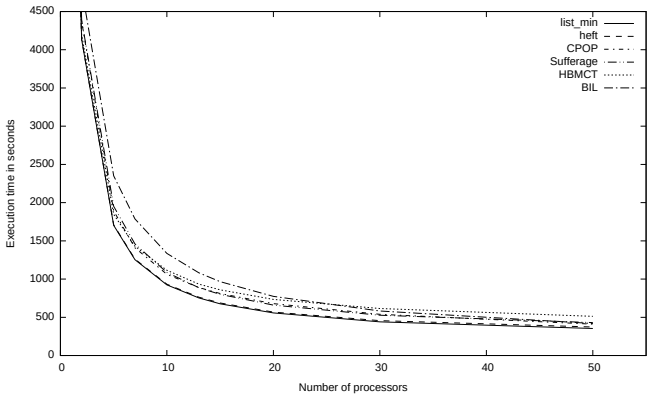


Figure 7. Comparison between list-scheduling algorithms

In all experiments the *list_min* curve represents the best results obtained among all list-scheduling algorithms.

1) List-scheduling analysis: We start by testing the performance of the different list-scheduling algorithms presented in Section II-B. This initial experiment has for objective to show the behavior of list-scheduling algorithms.

For this experiment, we use both TGFF and layer-by-layer graph generation algorithms. We also consider different data transfer sizes. Overall the results obtained are very similar among all parameter sets. Figure 7 shows results obtained with the largest data size tested (up to 1GB) and the TGFF algorithm. The network topology in use is here *clique*.

It can be seen that the overall behaviors of the algorithms are identical by increasing the processor number, speeding up the execution until reaching a limit due to critical paths.

We see that globally, most algorithms have the same performance with excluding *CPOP* and *MaxMin* which show larger execution times.

In the remainder of the paper, we consider only the *list_min* algorithm.

2) WSCOM on random DAG: We present now a comparison between *list_min* and *WSCOM* using the distributed hash table and allowing or not pre-fetching.

Note again that the list-scheduling algorithms are working off-line and as such know in advance all processing times and transfers sizes. On the opposite WSCOM is working online and only knows the DAG topology. The comparison of these algorithms is still meaningful as it allows us to assess the performance of WSCOM.

We start with experiments with low communications where the amount of data is lower than 50 Megabytes and continue with larger communication sizes.

Small communication times ($< 1s$): We initially consider small communication times as list-scheduling algorithms do not take congestion into account. Indeed with a

low volume of communications we can expect congestion to be minimal. For this experiment both cluster and clique topologies render similar results.

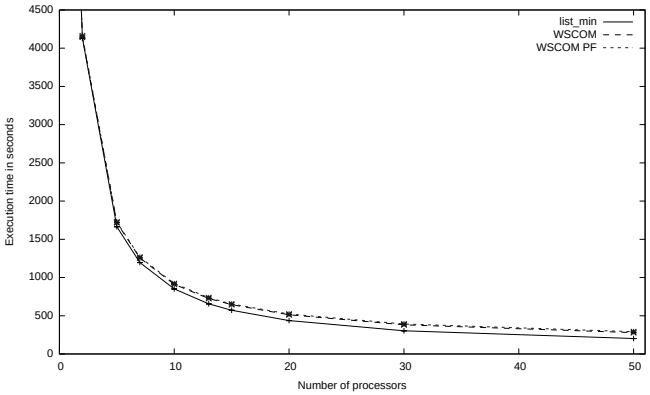


Figure 8. Comparison of list_min, WSCOM and WSCOM PF, small communications

Figure 8 displays execution times as the number of processors increases. We can clearly see that all algorithms achieve similar performance. This is in fact the expected behavior of WSCOM since with low communication times work-stealing algorithm should behave like a decentralized list-scheduling algorithm.

The fact that WSCOM PF is behaving equally well is more interesting. WSCOM PF is (as detailed in Section III-D) limiting steal requests to fork tasks and as such potentially reducing the parallelism of the application. Moreover, this limitation does not in fact impact results negatively.

Large communication times: As the communication volumes increase, congestion on shared link starts appearing. For experiments on larger data sizes, we therefore consider both cluster and clique topologies to assess the shared link effect on performance. We recall that on the clique topology, no link is shared and therefore list-scheduling algorithms are as efficient as they predict. On cluster topologies however, congestion can affect communications and decrease the performance of these algorithms.

Figure 9 presents a comparison of list_min, WSCOM and WSCOM PF for a clique topology with a maximal communication time equal to 10 seconds. It can be seen that the performance of WSCOM is now worse than performance of list_min. This comes from the fact that list-scheduling algorithms can overlap communications with computations while WSCOM is waiting for all communications before the start of each task.

Thus, WSCOM PF which sends the data in advance achieves a better execution time, close to the list_min execution time.

We should also emphasize that while the on-line execution of WSCOM PF does not take advantage on information

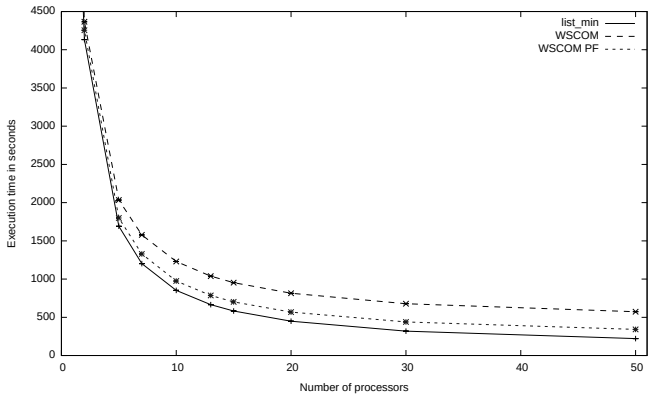


Figure 9. Comparison between list_min, WSCOM and WSCOM PF, large communications, clique topology

on transfer sizes it still achieves close execution times to list_min. This behavior validates the recursive clustering of WSCOM as a way to achieve efficient communications.

Figure 10 introduces the performance of list_min WSCOM and WSCOM PF on a cluster topology.

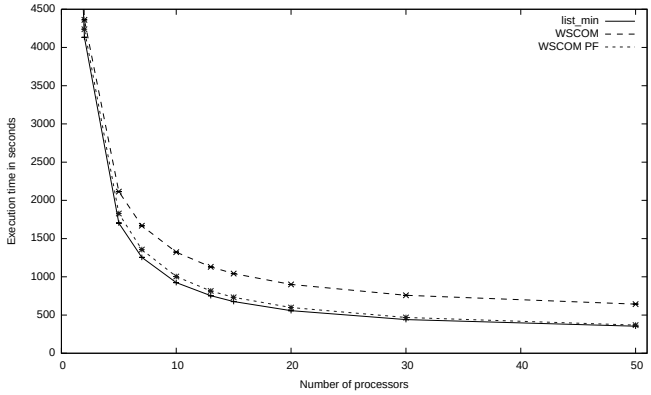


Figure 10. Comparison of list_min, WSCOM and WSCOM PF, large communications, cluster topology

As the cluster topology induces congestions the performance of the list-scheduling algorithms decrease.

One very interesting point of this experiment is that WSCOM PF is now achieving lower executions times than list_min. We consider this result as a very strong point in favor of WSCOM and as such are interested in a more detailed analysis.

Figure 11 represents the number of data transfers for the different algorithms.

This figure shows that WSCOM and WSCOM PF are indeed executing a lower amount of communication than list_min. We recall that WSCOM was designed as a bi-objective algorithm with a first goal to minimize the execution time and a second goal to decrease the overall among of communication.

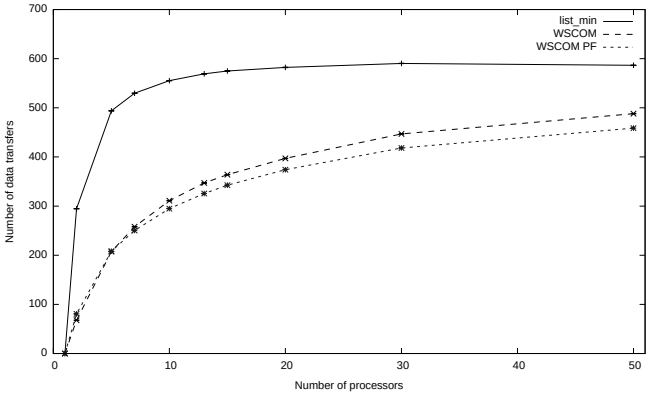


Figure 11. Number of transfers, large communication, cluster topology

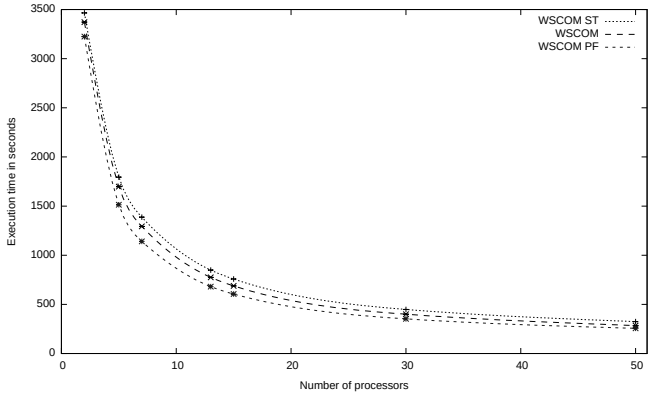


Figure 12. Comparison between WSCOM ST and WSCOM, large communications, layer-by-layer generation

Fewer communications result in a reduced congestion and at the same time means that congestion can affect less communications. The execution time is therefore less likely to depend on the network state for WSCOM and WSCOM PF algorithms.

In particular, the difference in the amount of communication between WSCOM PF and list_min is the greatest for a processor number comprised between 10 and 20. This impacts the execution times as the differences between list_min and WSCOM PF on Figure 10 are also more important for these numbers of processors.

Of course, as the number of processors grows, the amount of transfers required to balance the load grows as well and the differences between the algorithms reduce.

3) WSCOM ST: We continue by evaluating a variant of the WSCOM algorithm: *WSCOM ST* introduced in Section III-C. We recall quickly that WSCOM ST differs from WSCOM (DHT) in that the fork edges are chosen off-line.

WSCOM ST is in most experiments giving results identical to results from WSCOM. However, the algorithms differ slightly in a specific experimental setting.

We consider here a layer-by-layer graph generation on a clique platform with large communications. We emphasize that the algorithms perform equally well for TGFF generation (and thus, the graph generator *can* impact performance).

Figure 12 displays results comparison of WSCOM and WSCOM ST on the clique topology with large communications. Here WSCOM achieves shorter execution times than WSCOM ST.

Thus, this experiment justifies the choices to postpone fork decisions at runtime.

4) WSCOM on the Kaapi Makefile: Finally we validate our results on a real application DAG (as presented in Section IV-A1. Figure 13 introduces the comparison of WSCOM, WSCOM PF and the list_min schedule for the Kaapi DAG on a cluster platform.

Results present no difference to the results obtained on random graphs.

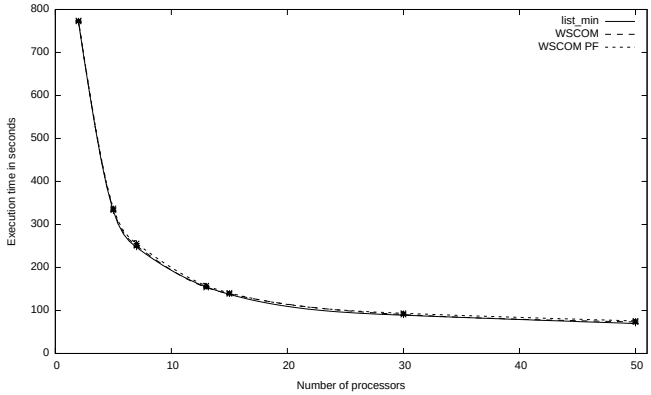


Figure 13. Comparison of list_min, WSCOM and WSCOM PF on Kaapi DAG

5) Conclusion on experiments: In these experiments WSCOM and WSCOM PF are compared to the list_min schedule which selects on each DAG the list-scheduling schedule with the shortest schedule.

On applications with few communications WSCOM and WSCOM PF achieve a schedule as efficient as the list_min schedule without information neither on the amount of data transfer nor on processing times. For data intensive application, results depend on the network topology and the congestion on links.

Experiments validate our choices to develop WSCOM PF since the use of pre-fetching allows us to achieve the same performance as list_min when no congestion takes place and even outperforms list scheduling in the event of congestions.

We believe that our experiments validate all design choices on the proposed WSCOM algorithms. Experiments show that a reduction in the amount of communication can indeed improve performance. Comparisons between WSCOM ST and WSCOM DHT show that it presents interest to postpone choices at runtime. Finally we have been able to show that information on DAG topology enables us to build

a recursive clustering of tasks which can yield to acceptable performance.

V. Conclusion

In our paper we study the scheduling of DAG of tasks with communication. We introduce an on-line scheduling algorithm *WSCOM* together with several variants. WSCOM is taking advantage of the knowledge of the graph to compute one recursive clustering of the tasks. This clustering enables our algorithms to reduce the amount of communication and thus to achieve performance even in the event of congestion.

We conducted a set of experiments evaluating the proposed algorithms and comparing them to off-line list-scheduling heuristics from the literature. With a low amount of communication, our algorithms and list-scheduling algorithms show similar performance. Moreover, in the event of network congestion WSCOM with pre-fetching is able to achieve better results than the off-line algorithms.

In future work, we will focus on real-world execution as we are now finalizing the implementations of the different WSCOM algorithms within DSMake.

Moreover we hope to provide a more theoretical analysis of performance on different classes of graphs.

Acknowledgment

The authors thank F. Suter for his help on the list-scheduling algorithm development.

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, pp. 720–748, September 1999.

[2] H. J. Hoogeveen, j. K. Lenstra, and B. Veltman, "Three, four, five, six, or the complexity of scheduling with communication delays," *Operations Research Letters*, vol. 16, no. 3, pp. 129 – 137, 1994.

[3] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Cambridge, MA, USA: MIT Press, 1989.

[4] M.-Y. Wu and D. D. Gajski, "Hypertool: A programming aid for message-passing systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 1, pp. 330–343, 1990.

[5] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, pp. 506–521, 1996.

[6] G. C. Sih and E. A. Lee, "Dynamic-level scheduling for heterogeneous processor networks," in *SPDP*, 1990, pp. 42–49.

[7] T. Yang and A. Gerasoulis, "Dsc: Scheduling parallel tasks on an unbounded number of processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 951–967, 1994.

[8] Y. Tao and G. Apostolos, "Pyrros: static task scheduling and code generation for message passing multiprocessors," in *Proceedings of the 6th international conference on Supercomputing*, ser. ICS '92. New York, USA: ACM, 1992, pp. 428–437.

[9] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal of Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[10] H. Topcuouglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, pp. 260–274, March 2002.

[11] H. Oh and S. Ha, "A static scheduling heuristic for heterogeneous processors," in *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, ser. Euro-Par '96. London, UK: Springer-Verlag, 1996, pp. 573–577.

[12] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems," in *HCW '99*. Washington, DC, USA: IEEE Computer Society, 1999.

[13] S. Rizos and Z. Henan, "A hybrid heuristic for dag scheduling on heterogeneous systems," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, april 2004.

[14] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," *Theory Comput. Syst.*, vol. 34, no. 2, pp. 115–144, 2001.

[15] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009.

[16] N. Gast and B. Gaujal, "A mean field model of work stealing in large-scale systems," in *ACM sigmetrics*, New-York, 2010.

[17] A. Shivali, B. Rajkishore, B. Dan, S. Vivek, S. R. K., and Y. Katherine, "Deadlock-free scheduling of x10 computations with bounded resources," in *SPAA*, 2007.

[18] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded lan guage," in *ACM SIGPLAN*, june 1998, pp. 212–223.

[19] T. Gautier, X. Besseron, and L. Pigeon, "Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors," in *PASCO*, 2007.

[20] R. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal, "Satin: Simple and efficient Java-based grid programming," in *AGridM Workshop*, 2003.

[21] A. Robison, M. Voss, and A. Kukanov, "Optimization via reflection on work stealing in tbb," in *IPDPS*, 2008, pp. 1–8.

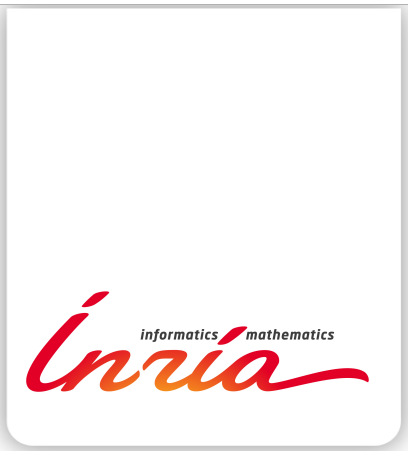
[22] J. K. Lee and J. Palsberg, "Featherweight x10: A core calculus for async-finish parallelism," in *PPoPP*, 2010.

[23] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: a Generic Framework for Large-Scale Distributed Experiments," in *10th IEEE International Conference on Computer Modeling and Simulation*, Mar. 2008.

[24] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," in *Proceedings of 3rd International ICST Conference on Simulation Tools and Techniques*. Malaga Espagne: ICST, mar 2010.

[25] R. P. Dick, D. L. Rhodes, and W. Wolf, "Tgff: task graphs for free," in *Proceedings of the 6th international workshop on Hardware/software codesign*, ser. CODES/CASHE '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 97–101.

[26] T. Tobita and H. Kasahara, "A standard task graph set for fair evaluation of multiprocessor scheduling algorithms," *Journal of Scheduling*, vol. 5, no. 5, pp. 379–394, 2002.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr