WSCOM: Online task scheduling with data transfers

Jean-Noël Quintin
INRIA Moais research team
Laboratory CNRS LIG
Grenoble University
France
jean-noel.quintin@imag.fr

Frédéric Wagner
INRIA Moais research team
Laboratory CNRS LIG
Grenoble University
France
frederic.wagner@imag.fr

***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. This situation is typically encountered when scheduling DAG of tasks corresponding to Makefiles executions.**

We introduce a new variation of the work-stealing algorithm: WSCOM. We take advantage of the knowledge of the DAG topology to cluster communicating tasks together and reduce the total number of communications.

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 and classical work-stealing from the literature. Simulations are executed on both random graphs and a new trace archive of Makefile DAG. These experiments validate the different design choices taken. In particular we show that WSCOM is able to achieve performance close 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. Moreover we show that WSCOM can achieve the same high performances as the classical work-stealing with up to ten times less bandwidth.

***Keywords*-load-balancing; online-scheduling; data transfers; work-stealing;**

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 corresponding scheduling problem is difficult for two main reasons. First, as the files sizes might be relatively large we take communications into account. Secondly, we have a non-clairvoyant setting: task sizes and communication are not known in advance, nor the network topology.

This work has been supported by the minalogic project CILOE

We present Section II related algorithms from the literature. We present off-line scheduling algorithms optimizing communications and the classical work-stealing algorithm [1] optimizing completion time.

Section III presents *WSCOM*, our new online scheduling algorithm. Our approach is to reduce the number of communications performed while keeping the load balanced.

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. Related Algorithms

A. Offline algorithms

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. We can cite the most commonly used heuristics like [4] DCP [5] DL [6] DSC [7]. While these algorithms give good performances for an infinite number of machines, execution

on a limited number of processors requires a folding of the schedule which degrades performances.

Aside from clustering, many useful heuristics are based on list scheduling. The main objective is then to avoid idle times more than to reduce communications. In particular most algorithms execute communications as soon as possible in order to cover communication times by computations. More specifically we can cite HEFT [10], CPOP [10], BIL [11], MinMin [12], MaxMin [12], sufferage [12] and HBMCT [13].

B. 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.

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.

We modify the work-stealing algorithm to take advantage of the additional information on the DAG structure.

Section III-A presents the main idea of *WSCOM*, on the restricted case where DAG are join DAG. The algorithm is then extended to its more general version Section III-B.

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

A. 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 communications, the execution should obviously be sequential and the opposite 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. This change of point of view allows to optimize communications even in the online case since we now consider minimizing the *number* of communications. With this approach we can hope that in reasonable configurations a reduction in the number of communications might show a positive impact on the completion time.

The second important idea is to combine clustering and work-stealing to achieve performance for both our 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 communications.

To achieve the clustering and to provide recursive task creation we add some new virtual tasks to the DAG. These *Fork* tasks require no computations but will simply generate other tasks on the local stack when executed. 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 regroup communicating tasks together.

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 1. The fork-DAG is identical to the task DAG with reversed edge orientation. Moreover, a fork-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.

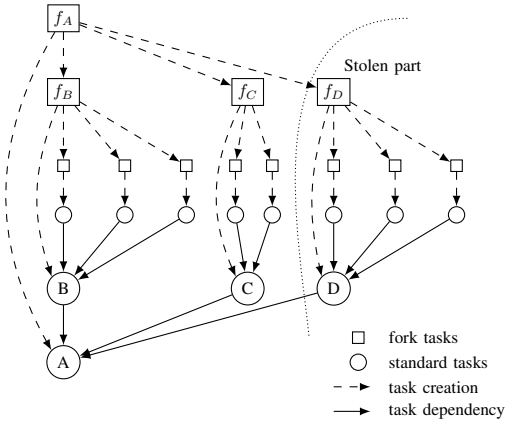

Figure 1. WSCOM DAG using Symmetry

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 **??** (with no information on tasks sizes and communications sizes) to generate clusters. However, obtaining a recursive decomposition is not straightforward.

B. WSCOM on general DAG

Extending WSCOM to DAG leads quickly to the problem displayed Figure 2. 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.

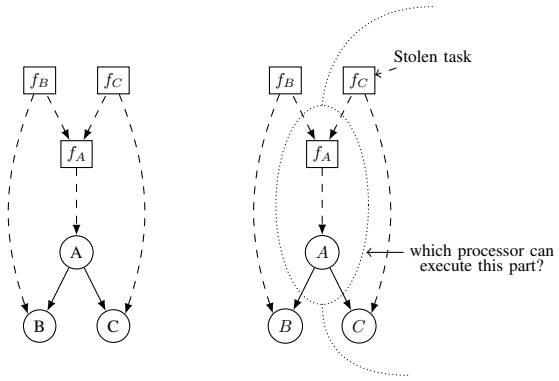

Figure 2. Problem with Outgoing Degree > 1

To solve this problem we need to remove fork edges between tasks such that each 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.

We provide two different algorithms solving this problem.

The first algorithm 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.

An other option is to 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 rest of this paper we refer to WSCOM DHT as WSCOM.

Algorithm 1 WSCOM

```
Require: G(V,E)// Application DAG
  initially the current task c is undefined
  we denote by m(v) the mirrored vertex of any vertex v
  let s be the unique sink of G
  if proc_id = 0 then
    c = m(s)
  end if
  while s is not executed do
    while c is undefined do
      if a task is ready on the stack then
        pop last ready task from the stack into c
      else
        try to steal a task from a random machine and put
        in c
      end if
    end while
    // Check if the task is virtual or not
    if c is a fork task then
      if c has not been forked already then
        mark c as forked
        for all predecessor p of m(c) do
          push m(p) on the stack
        end for
        push m(c) on the stack
      end if
    else
      execute c
      update dependencies
    end if
  end while
```

C. Data transfer

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

Let A and B be two tasks such that B depends on the output of A to start. We assume the required data cannot be sent until A completes. Moreover, sending this data also requires knowing the machine which will execute B. Since our algorithm allows un-executed tasks to be moved between 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 operations since the generate extra transfers of data. Such costs can eventually involve tasks on the critical path and impact the completion time. However since a task can only be stolen once (because it is executed right away), 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.

In this paper, as we try to minimize the number of communications, we will choose the second approach. We refer to this algorithm as WSCOM$_{PF}$(pre-fetching).

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 listed 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. The use of simulations allows us to compare executions on a large set of different platforms and thus to test our algorithms under many different bandwiths.

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 analyse execution times and communications volumes.

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.

On the other side, we have extracted a set of around 500 Makefiles from the widely known MacPort [**?**] repository. A set of large applications has been compiled on a platform. Resulting compilation times have been monitored, as were the communication sizes and the resulting graphs are stored in an online archive available on our website at .

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 platforms 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 here 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 a number of processors comprised between 1 and 50.

Link capacities are defined with a latency equal to 0.1 millisecond and a bandwidth equal to 1 Gbit per second.

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).

Trust intervals are not displayed on our curves as the variations on the obtained results are minimal.

In all experiments the *list_min* curve represents the best results obtained among all list-scheduling algorithms: HEFT [10], CPOP [10], BIL [11], MinMin [12], MaxMin [12], sufferage [12] and HBMCT [13].

1) WSCOM on random DAG: We start by presenting a comparison between *list_min* and *WSCOM* using the distributed hash table and allowing or not pre-fetching.

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.

Note 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 on-line and only knows the DAG topology. The comparison of these algorithms is still meaningful as it allows us to assess the performance of WSCOM.

For small communication volumes, all algorithms exhibit very close performances and we do not present these results in more detail.

Large communication times: As the communication volumes increase, congestion on shared links starts appearing. For experiments on large 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 3 presents a comparison of list_min, WSCOM and WSCOM$_{PF}$ for a clique topology. It can be seen that the

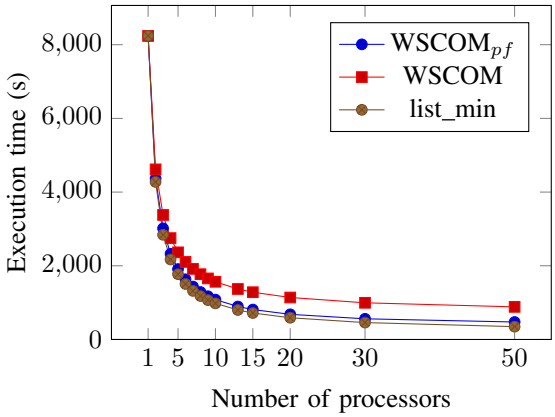

Figure 3. Comparison between list_min, WSCOM and WSCOM PF, large communications, clique topology

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 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 4 introduces the performance of list_min WSCOM and WSCOM$_{PF}$ on a cluster topology.

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

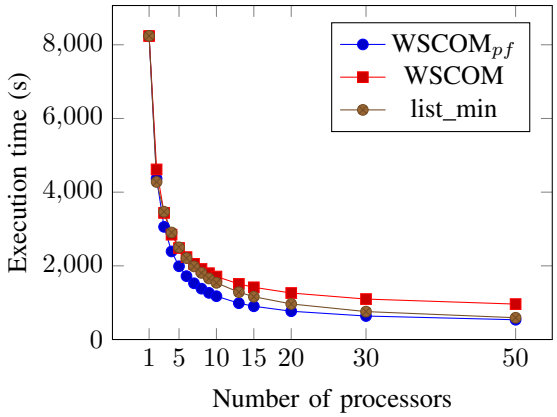

Figure 4. Comparison of list_min, WSCOM and WSCOM PF, large communications, cluster topology

One very interesting point of this experiment is that WSCOM$_{PF}$ is now achieving lower executions times than list_min. To explain such a result, we are interested in a more detailed analysis.

Figure 5 represents the number of data transfers for the different algorithms.

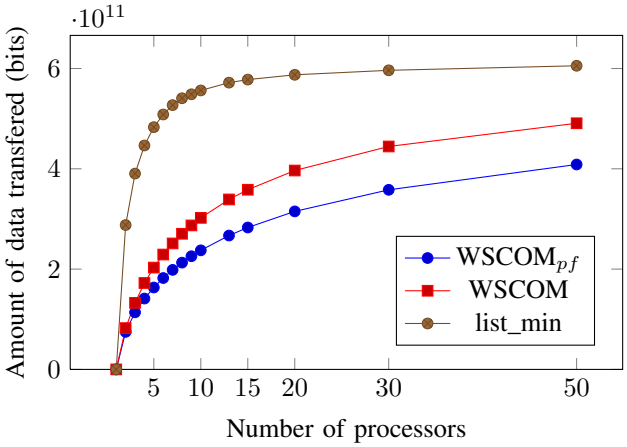

Figure 5. Amount of data transfered, large communication, cluster topology

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.

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 4 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.

2) WSCOM on the Trace Archive: We have executed similar experiments with graphs from the Trace Archive and we now present the most significant results obtained.

For the Trace experiments, different graphs may generate different kind of behaviours. All curves are therefore presented as clouds of points where each point represents the average of the results obtained on one graph.

First, we start by presenting the performances obtained on a 1Gbit clique network. With such settings, Figure **??** shows that for most experiments, performances of WSCOM and list_min are similar. These results are consistent with results on random graphs with few data communicated.

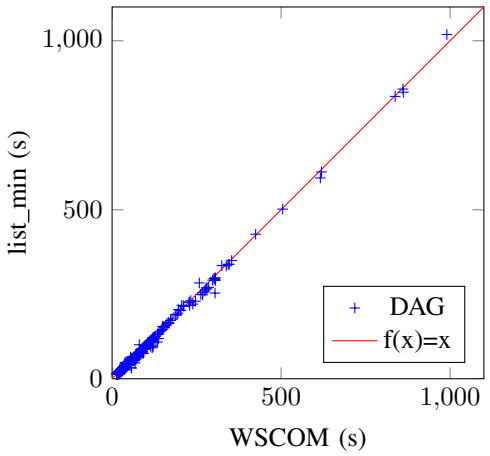

Figure 6. Compare list_min and WSCOM on DAG extracted from Makefile (clique with 5 processors).

The next experiment evaluates the difference between WSCOM and WSCOM$_{PF}$ and the same 1Gbit/s clique configuration. We can see on Figure 7 that this time, the behaviour observed differs widely from the behaviour on random graphs. WSCOM$_{PF}$ presents no performance increase over WSCOM and even exhibit performances degradations on some graphs.

We believe that this effect comes from a shape difference in graphs. WSCOM$_{PF}$ generates on trace graphs a load imbalance as explained section III-C This shows that actual graphs generators are not completely capturing the characteristics of real Makefile applications.

Variable Bandwidths: We conclude this section with experiments on variable bandwidths. We try here to evaluate how much WSCOM is able to widen the application range of

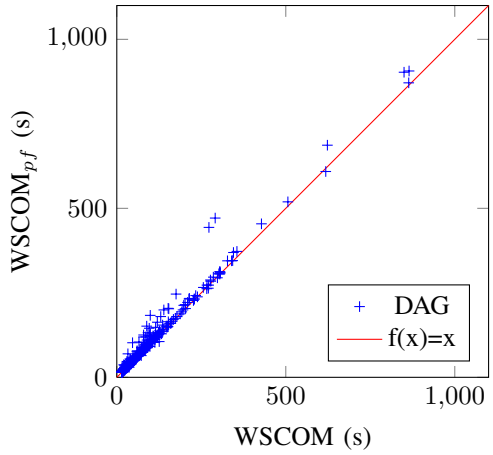

Figure 7. Compare WSCOM$_{pf}$ and WSCOM on DAG extracted from Makefile (cluster with 5 processors).

traditional work-stealing. More precisely, we study a subset of the trace graphs exhibiting large communications (above 100MB) and speed-ups greater than five. For each graph we run experiments with different bandwidths in order to determine the minimum bandwidth necessary to reach a speed-up of 4 on 5 computers. We hope to show here what kind of platforms can be considered to achieve acceptable performances.

We plot Figure 8 for both WS and WSCOM the minimal amount of bandwidth necessary to reach a speedup of 4 on a cluster platform. Each point displayed is proportional in size to the number of graphs reaching the two corresponding bandwidths. For some graphs, presented on the top, WS never achieves a speed-up of 4.

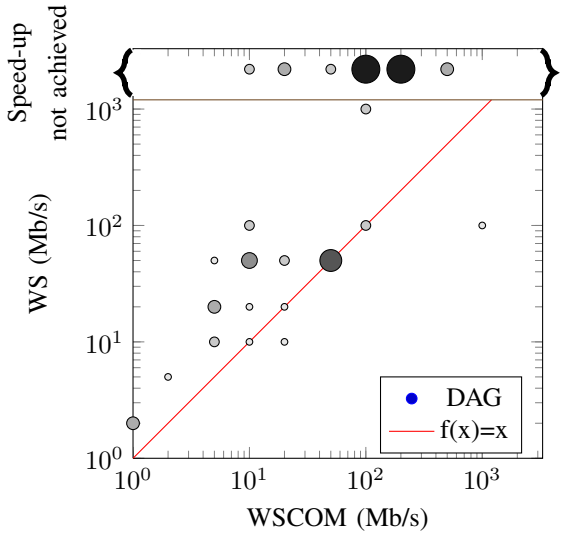

Figure 8. Minial bandwidth to achieve a speed-up above 4 on five processors (WS, WSCOM).

We clearly see that WSCOM is able to reduce the bandwidth requirements of the applications with for many graphs a reduction of the needed bandwidth by a factor 10. This experiment shows more than the others the usefulness of WSCOM. We are able here to extend the field of application of distributed makefile computations to a wider set of platforms. This property really corresponds to the initial objectives and developmental approach chosen.

3) 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. In the case where the communications become more intensive, we exhibit differences of behaviour for WSCOM$_{PF}$ between randomly generated graphs and graphs coming from the trace archive.

We also show that WSCOM is able to achieve high performances even in the case of reduced bandwidths. As such, we are able to consider executions on a wider range of platforms.

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.

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 on random graphs.

Future works are of many different kinds.

We can hope to achieve real-world executions as we are now finalizing the implementations of the different WSCOM algorithms within DSMake.

Other improvements might be to consider re-execution of communications instead of relying on WSCOM$_{PF}$ to enable pre-fetching.

It should also be possible to generate more realistic random graphs by developing new random generation algorithms matching more closely the characteristics of the trace archive graphs.

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

ACKNOWLEDGMENT

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.