Mixed Data-Parallel Scheduling for Distributed Continuous Integration

Olivier Beaumont, Nicolas Bonichon, Ludovic Courtès, Xavier Hanin, Eelco Dolstra

Mixed Data-Parallel Scheduling for Distributed Continuous Integration

Olivier Beaumont
Inria,
University of Bordeaux, F-33400 Talence, France
Labri, CNRS UMR 5800
olivier.beaumont@labri.fr

Nicolas Bonichon
University of Bordeaux, F-33400 Talence, France
Inria
Labri, CNRS UMR 5800
bonichon@labri.fr

Ludovic Courtès
Inria, F-33400 Talence, France
ludovic.courtes@inria.fr

Xavier Hanin
4SH
Bordeaux, France
xavier.hanin@4sh.fr

Eelco Dolstra
Delft University of Technology
The Netherlands
e.dolstra@tudelft.nl

***Abstract*—In this paper, we consider the problem of scheduling a special kind of mixed data-parallel applications arising in the context of *continuous integration*. Continuous integration (CI) is a software engineering technique, which consists in rebuilding and testing interdependent software components as soon as developers modify them. The CI tool is able to provide quick feedback to the developers, which allows them to fix the bug soon after it has been introduced. The CI process can be described as a DAG where nodes represent package build tasks, and edges represent dependencies among these packages; build tasks themselves can in turn be run in parallel. Thus, CI can be viewed as a mixed data-parallel application. A crucial point for a successful CI process is its ability to provide quick feedback. Thus, makespan minimization is the main goal. Our contribution is twofold. First we provide and analyze a large dataset corresponding to a build DAG. Second, we compare the performance of several scheduling heuristics on this dataset.**

***Keywords*-DAG Scheduling; mixed parallelism; continuous integration**

I. Introduction

Continuous integration [1], [2] (CI) is a simple idea, that is widely used in software development: as soon as a software component is modified, the CI tools rebuilds it, runs its test suite, and notifies developers of the build status, allowing them to quickly rectify bugs. Widespread CI tools include Jenkins [3], Hudson [4], and CruiseControl [5]. In a collaborative project, it is crucial to detect bugs as soon as possible.

The value of a CI process lies in its ability to provide quick feedback to developers, as they are more likely to understand and fix bugs that when they are uncovered quickly. Therefore, parallelism is a natural solution to improve the efficiency of the process.

In the context of CI, parallelism comes into two flavors. First, the overall process can be seen as a DAG, where nodes correspond to package builds and edges correspond to dependencies among packages. The compilation of a package must complete before the compilation of packages that use it, and if a package is built on a machine, then it has to download the packages it depends on.

Second, most build tasks can in turn run in parallel on a multi-core machine. This is because each package has a build system—*e.g.*, using makefiles or Ant—that describes a DAG of fine-grain build tasks, and build tools can exploit it to launch this build tasks in parallel—*e.g.*, GNU Make's `-j` option. As we will see, the efficiency of parallelization of elementary tasks varies from package to package. We will therefore rely on the *moldable task model* to model parallel tasks (see [6] for a complete survey on parallel task models), where for each task, the processing time of the task is given for each number of cores on which it may run.

Therefore, CI can be thought of as a *mixed data-parallel application* [7], [8], [9]. The scheduling problem for mixed-parallel tasks consists in deciding which compute resources should perform which task and when, so as to optimize a given metric such as the overall execution time—*i.e.*, the *makespan*. For mixed data-parallel applications, the main difficulty is to decide whether to allocate more cores to run tasks faster, or to run more tasks in parallel. This question does not have an obvious answer for general task graphs. Therefore, it is of interest to concentrate on a specific application domain and on a specific dataset.

Our contribution in this paper is twofold

- First, we analyze and make available [10] to the community of large dataset that is a good representative of the task graphs arising in the context of CI and we provide a simulation tool (namely *Hubble* [10] to perform more experimentations. Hubble is based on SimGrid [11]. The dataset consists both in the DAG of tasks and in the moldability results for the tasks themselves. Making such a dataset available to the scheduling community is important, since it enables to compare several algorithms and heuristics in a realistic

context, without relying on random DAGs generators. Moreover, this example shows that the CI DAGs have specific characteristics that strongly influence the algorithms: the overall communication volume is relatively small with respect to computations, the tasks on the critical path represent more than 10of the overall computation volume,... Another important point is that several compilation artifacts are used at many places in the DAG, therefore showing the interest of scheduling algorithms that take into account the fact that tasks share files, such as proposed in [12], [13].

- Second, we propose and compare the performance of several algorithms and heuristics devoted to schedule CI DAGs. For instance, we propose an algorithm to deal with tasks sharing files, although the small amount of communications makes its influence of the overall scheduling time relatively small. We also show that, if we consider that compilation tasks are sequential—*i.e.*, not moldable—it is relatively easy to reach the optimal time (the length of the critical path), even with relatively few processors. On the other hand, we show that taking the moldability of tasks into account enables to decrease significantly the CI makespan. In the context of CI, it is also reasonable to assume that CI tasks are run on a IAAS Cloud. Therefore, our study can also be used in order to dimension how many resources should be reserved to run the CI task at optimal cost.

To model contentions, we rely on the bounded multi-port model, that has already been advocated by Hong et al. [14] for task scheduling on heterogeneous platforms. In this model, node P_i can serve any number of clients P_j simultaneously, provided that its outgoing bandwidth is not exceeded by the sum of the bandwidths allocated to the communications it is involved in. Similarly, P_j can simultaneously receive messages from any set of clients P_i provided that its incoming bandwidth is not exceeded. This corresponds well to modern network infrastructure, where each communication is associated to a TCP connection. This model strongly differs from the traditional one-port model used in the scheduling literature, where connections are made in exclusive mode: the server can communicate with a single client at any time-step. It is worth noting that several QoS mechanisms available in modern operating systems, enable a prescribed sharing of the bandwidth [15], [16], [17]. Therefore, the bounded multiport model encompasses the benefits of both bounded multi-port model and one-port model. It enables several communications to take place simultaneously and practical implementation is achieved using TCP QoS mechanisms.

In this paper, we will concentrate on dynamic (on-line) algorithms. Indeed, we have observed that the time needed to perform a task, especially in the multi-core context, varies from one experiment to another. Therefore, we cannot assume that processing times of moldable compilation tasks are known exactly and we will only rely on scheduling algorithms that make their decision at runtime, depending on the available resources (idle cores), the set of ready tasks (tasks whose all dependencies have been resolved) and the (possibly static) priorities that have been assigned to the tasks.

The rest of the paper is organized as follows. Section II presents the related works. Section III presents the software distribution *Nixpkgs* and the DAG of packages extracted from this distribution. The shape and the moldability of this DAG is also studied. Section IV compares the efficiency of several scheduling heuristics on Nixpkgs DAG. The paper concludes with Section V.

II. Related work

In this paper, we consider the problem of scheduling a mixed data-parallel task graphs under a realistic model for communication contentions. Parallel tasks models have been introduced in order to deal with the complexity of explicitly scheduling communications and have been first advocated in [18]. In the most general setting, a parallel task comes with its completion time on any number of resources (see [19], [20], [21] and references therein for complete surveys). The tasks may be either *rigid* (when the number of processors for executing a dedicated code is fixed), *moldable* (when the number of processors is fixed for the whole execution, but may take several values) or *malleable* [22] (when the number of processors may change during the execution due to preemptions).

Static scheduling algorithms with guaranteed performance for scheduling applications structured as mixed data-parallel task graphs have been developed in the case of a single homogeneous parallel computing platform, such as a cluster in [23], [24] and have been later extended to multi-clusters in [7], [8], [25]. It is worth noting that if communication costs on the edges are neglected and if all tasks have the same moldability profile, then it is possible to find the optimal schedule in polynomial time [26]. Above works do not take contentions of communications in account. The problem DAG scheduling with contentions under a realistic communication model (but sequential tasks only) has been first considered in [27].

III. Nixpkgs Mixed Data-Parallel Build Task Graph

In this section we first present *Nixpkgs* distribution. We then explain how the build process of the whole distribution can be modeled as a DAG. Then, we study the shape of the DAG and the moldability of the tasks of the DAG—*i.e.*, of the packages of the distribution. The complete dataset can be found on the Hubble web page [10].

A. Nixpkgs, Nix, and Hydra

The DAG of tasks used in our experiments is a part of *Nixpkgs*, a software distribution used as the basis of the NixOS GNU/Linux distribution [28]. Nixpkgs is both a description of the DAG of packages of the distribution—*i.e.*, the dependencies among them—and the set of *build scripts* necessary to build these packages. In turn, these packages can be built using the *Nix* package manager [29].

Build results are stored in the *Nix store*, a special directory in the file system. There is a direct, deterministic mapping between a build task (build script and set of build inputs) and the name of its result in the Nix store. Thus, the Nix store can be seen as a *build cache*: build results can be reused whenever they are needed, without having to perform the build again and the directory where they are stored can be easily determined.

The Hydra continuous integration system is built on top of Nix[30]. Hydra is notably deployed on a heterogeneous cluster at TU Delft[1], currently consisting of two 48-core machines and three 8-core machines running GNU/Linux, as well as two dual-core machines and three VMs running other operating systems. A Hydra instance at Inria runs on a smaller, but similarly heterogeneous cluster[2].

The Hydra instance at TU Delft continuously builds several large free software projects, in particular Nixpkgs. By continuously building the packages defined in Nixpkgs, it allows developers of the distribution to quickly spot *package integration issues*. For instance, changing part of the standard environment (known as *stdenv*) such as the C compiler or C library triggers a rebuild of all its successors in the DAG; thus, it can be quickly determined if a change in *stdenv* lead to a build failure.

Nix and Hydra currently rely on naive greedy scheduling strategies. Within Nix, libstore topologically sorts the build DAG and then executes tasks in this order. When a build task is ready, it is submitted to the *build hook*, which, by default, submits it to one of the participating machines or declines it. In the latter case, the task is scheduled locally. In the former case, the build hook copies all necessary build inputs to the target machine, so that the build can actually take place.

Optionally, Nix can exploit parallelism *within* a build task: if a package uses a makefile-based build system, then Nix can be told to run GNU Make with its `-j`N option, which instructs it to launch up to N jobs in parallel, as permitted by the sub-DAG described in each makefile.

B. Extraction of the DAG of Mixed Data-Parallel Tasks

Here we explain how we have computed the DAG of mixed data-parallel tasks that models the Nixpkgs Distribution.

For each package (that is, each node of the DAG), the *size* is obtained by actually building the package and measuring the on-disk size of the build output.

Likewise, the *sequential time* is the observed build time, and the parallel build times are obtained when trying to exploit parallelism within the task using GNU Make with the `-j`N, for several values of $N > 1$. Given that package build tasks are deterministic, these build times are reproducible.

A subset of Nixpkgs packages (approximately 60% of them) was built on a 2.8 GHz multi-core `x86_64` machine with 128 GiB of RAM, with `-j`N `-l`$2N$, for $N = 1, 2, 4, 8, 16$. Values above 16 were not tried because, as we will see below, experiments showed that few packages would be able to take advantage of more parallelism.

Nixpkgs contains depencies information on build packages:

- The *direct dependencies*, that is the least of packages needed to build the packages.
- The residual *run-time dependencies* of each node. Typically, the set of direct dependencies needed at run-time by a build is a subset of the dependencies required to build it; for instance, a package's build process may require a compiler and a parser generator, but these are no longer needed at run-time when using the package.

Thus, for any node of the DAG, its actual dependencies are the union of its direct dependencies and the transitive closure of their run-time dependencies. The *size* of a dependency is given by the size of build output of the source of the dependency.

C. DAG Characteristics

The Nixpkgs DAG (revision 31312) of build tasks has the following characteristics:

- it consists of 7,361 nodes, 38% of which are sources such as source tarballs, build scripts, and patches;
- 198,637 edges—27 times the number of nodes;
- a sequential build of the whole DAG on a 2.8 GHz machine takes 100 hours;
- a complete build produces 36 GiB of data (this excludes source tarballs, patches, etc.);
- the total volume of dependencies is 2,302 GiB;
- 96% of the nodes have a build result greater than 8 KiB;
- speedup information is available for 86% of the build tasks; these build tasks correspond to 52% of the total amount of computation; some of the packages failed to build with `-j`N for $N > 1$, for instance because of errors in the build system, preventing packages that depend on it to build, hence this percentage;
- the critical path consists of 33 packages and its duration is 6 hours and 47 minutes, when build tasks are themselves performed sequentially.

[1]Hydra instance at TU Delft: `http://hydra.nixos.org`

[2]Hydra instance at Inria: `http://hydra.bordeaux.inria.fr`

We expect these characteristics to be typical of GNU/Linux distributions.

These characteristics strongly influence the relative importance of computations and communications. Indeed, performing all the transfers corresponding to all the dependencies sequentially (which is a very pessimistic assumption) only takes 8 hours (on a GiB Ethernet LAN) whereas the total computation volume is about 100 hours. Our intuition is that this ratio of computation to data transfer volume is typical of continuous integration DAGs.

Another important piece of information is the "shape" of the DAG. The Nixpkgs DAG starts with a mostly-sequential sequence of build tasks, which corresponds to the tasks that contribute to bootstrapping the standard build environment (known as *stdenv*), that contains the C library and the C compiler, along with build tools such as GNU Make and utilities like GNU Coreutils and Perl. The *stdenv* build is followed by a fork, which corresponds to the fact that *stdenv* is a prerequisite for all the remaining build tasks. The *stdenv* build phase is obviously on the critical path. A similar fork/join pattern occurs close at the other end of the DAG, with the Qt library and KDE support libraries from which many applications depend.

D. Moldability

After analyzing the topology of the DAG, we now turn to the moldability of the nodes.

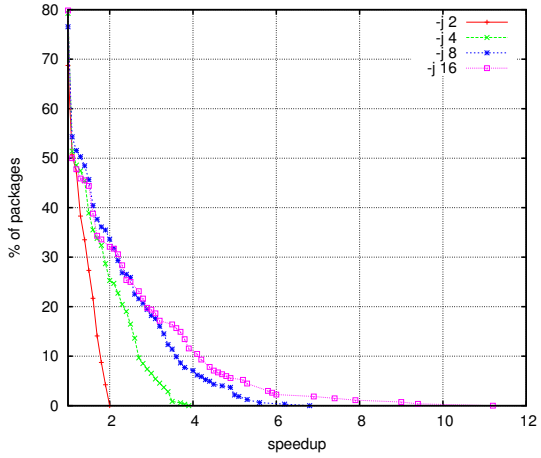

Figure 1. Complementary cumulative distribution of package speedups for different values of the option $-\texttt{j}N$, and for package builds with a sequential duration longer than 60 seconds.

We can see from Figure 1 that 50% of packages are purely sequential. Few package build processes are slower with $-\texttt{j}N$ option. The other packages have quite different speedup values.

We now assume that each package follows the Amdahl's law: the time $T(n)$ to build a package on n cores is equal to $T(1).((1-\alpha)+\alpha/n)$, where α is the fraction of the task that can be run perfectly in parallel. We performed a linear regression to estimate the α. Figure 2 shows the cumulative distribution of the parameter α for the tasks for which we have measures for at least 3 different values of n and whose sequential duration is larger than 300s. We can see that 45% of tasks are purely sequential and 40% of tasks have a perfectly parallel fraction greater than 80%. The mean value of α is 0.4.

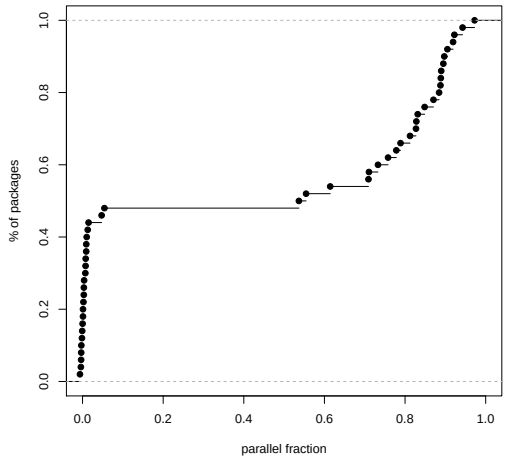

Figure 2. Cumulate distribution function of the parallelization fraction of tasks.

The vast majority of packages in Nixpkgs are built using makefiles, so they should potentially benefit from parallel builds with `make -j`. Yet, our measurements show that many package builds do not scale well. There are mainly two explanations:

- In general, package build tasks start with a purely sequential phase, often corresponding to a `./configure` run. When the `make` phase itself has a duration comparable to the `configure` phase, the build task as a whole does not scale well.
- Many packages rely on recursive makefiles, which limits make's ability to parallelize the build task to individual directories [31].

Tackling the two above problems is out of the scope of this paper. We collected speedup information for 40% of the build tasks. For those build tasks lacking speedup information, we use the following estimation: build tasks shorter than 300 sec. are considered purely sequential, and others are assumed to follow Amdalh's law with $\alpha = 0.4$. From this, we are able to predict what can be expected if we allow to use at most k cores per task. Table I summarizes this information by displaying for each possible number of cores per node ($N = 1, 2, \ldots, 16$) the length of the critical

Table I
LENGTH OF THE CRITICAL PATH AND TOTAL COMPUTATION VOLUME AS A FUNCTION (SUM OF TASK DURATIONS TIMES THE NUMBER OF CORES PER TASK) OF THE NUMBER OF CORES USED FOR EACH BUILD TASK (N CORES CORRESPONDS TO $-\jmath N$).

cores per task	critical path (sec.)	comp. volume (sec.)
1	24,427	359,978
2	18,378	552,004
4	16,003	945,770
8	14,728	1,752,833
16	14,115	3,399,700

path and the overall computation time if an infinite number of N-cores nodes were available.

Using only one core per task, it is not possible to build a schedule with a makespan lower than 6 hours and 47 minutes. As the ratio between the total volume of computation and the critical path length is 14.7, at least 15 cores are necessary to achieve performance close to this bound.

If we allow to run each task on at most 2 cores, 5 hours and 6 minutes is a lower bound on the makespan. Moreover if we use exactly two cores per task, at least 30 cores are necessary to achieve performance close to this bound.

If we allow to use at most 16 cores per task, the critical path length drops to 4 hours and 6 minutes.

IV. EXPERIMENTATIONS

In this section we compare the performance of several heuristics based on HEFT algorithm [32] on multi-cores machines connected with a Ethernet network.

A. Scheduling Heuristics

In all the algorithms we consider, tasks are ordered according to their upward rank—*i.e.*, the length of the longest path from the task to any other task, the length of a path being the sum of the computation cost of the tasks of the path. Tasks are scheduled in the non increasing order of their upward rank.

We consider one static algorithm and three dynamic algorithms. The static algorithm assigns a priori tasks to cores. Dynamic algorithms run the simulation and, as soon as there are idle cores and ready tasks—*i.e.*, tasks such that all the packages they depend on are done—it schedules the task with the highest priority (according to the upward rank order) to one or several cores. Note that the dynamic algorithms we consider can schedule a task on one or several idle cores, even if this task could finish earlier if scheduled on another machine that is currently busy.

The considered algorithms are the following ones:

- **HEFT (static)**. This heuristic assigns the task to the core that minimizes the finish time of the task (knowing that the previous tasks are already scheduled).
- **HEFT (dynamic)**. While there are ready tasks and idle cores do: schedule the highest priority (considering the upward-rank) ready task on the core that minimizes its finish time.

The next two heuristics use the moldability of tasks:

- **MHEFT (dynamic)**. While there are ready tasks and idle cores do: schedule the first ready task on the smallest set of idle cores on the same machine that minimizes its finish time.
- **2-CORES (dynamic)**. While there are ready tasks and idle cores do: schedule the first ready task on a couple of idle cores of a machine that minimize its finish time. This algorithm works only on platforms with machines with an even number of cores.

Many other heuristics have been proposed in the literature. Nevertheless, we believe that these four heuristics well represent what can be achieved in the context of Continuous Integration, given the characteristics of the DAG and the parallel profile of the tasks.

When a package artifact generated on a machine M_1 is used by another package on another machine M_2, this artifact must be transfered from M_1 to M_2. Here, we consider that if an artifact that has been transfered on a machine, it will remain available for the other packages (there is no need to transfer again the same file). This *cache system* does not change the performance of static scheduling algorithm when no contention occurs. But when we consider dynamic scheduling algorithm or take contentions into account, this has an impact on the makespan of produced schedules. On the considered dataset (and by extension in the context of CI), the cache system is quite convenient since each generated file is used on average (and therefore potentially downloaded) 24 times.

Finally, let us note that even when the considered platforms are homogeneous, the choice of the machine on which the task will be scheduled is not driven only by the number of idle cores. It is also driven by the repartition of artifacts needed by a given tasks. Hence if several machines have one idle core, the task will be scheduled on the machine that minimizes the task completion time, taking transfers into account.

B. Target systems

Experimentations were performed using the SimGrid 3.5 simulator [11]. The platforms we have considered are multicore machines connected by a GiB Ethernet network (with an effective bandwidth of 81MiB/s). The power of each cores is 91.5 GFlop/s. These settings correspond to typical clusters used for CI, such as those described in Section III-A.

C. Results

The first question we try to answer is: what is the smallest number of processors such that the makespan is roughly equal to the critical path length? Since communications are negligible compared with computation, such lower bound may almost be reached. Since the critical path length is 14.7 times smaller than the overall volume of computations, at least 15 cores are needed. As shown in Figure 3, only 20 to cores are in fact needed by HEFT (dynamic) algorithm to reach this bound whereas HEFT (static) doesn't reach this bound with 24 cores. Concretely, with only 2 of the 3 opto-cores used in Delft, the same level of reactivity of the CI server can be achieved using a quite straightforward heuristic.

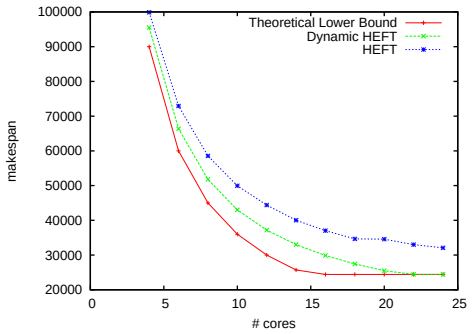

Figure 3. Execution of the DAG with one core per task.(on dual cores)

To obtain better reactivity, the moldability of tasks needs to be taken into account. If we allow tasks to run on at most 2 cores, then the theoretical bound on the makespan drops to 5 hours and 6 minutes. As shown in Figure 4, this result is achieved using 40 cores only. MHEFT seems even better since almost the same performance is achieved with only 28 cores. The main drawback of MHEFT is that it only makes use idle cores. Hence if a parallel task becomes ready when there is only one idle core, this task will be schedule on this single core.

Performances could be improved using static mixed parallel scheduling heuristics (see for instance [7]). However using static heuristics can be efficient only when it is possible to predict accurately the duration of tasks. This last assumption is not realistic when several applications run on the cluster on which the CI server is running.

V. Summary and Conclusion

In this paper, we consider the problem of scheduling DAGs obtained in the context of Continuous Integration. Our contribution is the analysis of the characteristics of a large DAG corresponding to the packages of the software

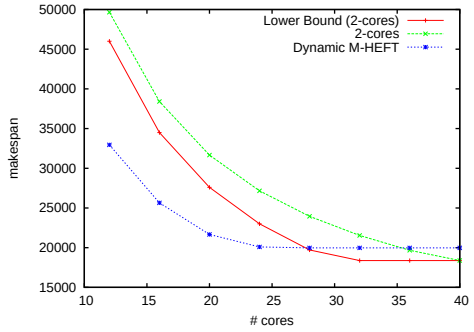

Figure 4. Execution of the DAG with several cores per task. (On quad cores)

distribution *Nixpkgs*. Continuous Integration is a very attractive application for both practical and theoretical studies on DAG Scheduling. Indeed, its practical importance is clear and the natural optimization function, so as to detect bugs as soon as possible, is makespan minimization. The process can be described as a mixed data-parallel application, where the volume of communications can in practice be neglected so that efficient approximation algorithms can be derived. In this paper, we have shown that basic scheduling heuristics can achieve quasi-optimal makespan at the price of using slightly more cores, but there is still room for improvement, both to design better algorithms and to theoretically justify why basic heuristics perform well.

Acknowledgment

The authors are very grateful to Arnaud Legrand and Samuel Thibault for helfpul discussions. The authors would also like to thank Lionel Eyraud and Frédéric Suter for their help with SimDAG, and the anonymous reviewers for their insightful comments.

References

[1] M. Fowler and M. Foemmel, "Continuous integration," *Thought-Works) http://www.thoughtworks.com/Continuous Integration.pdf*, 2006.

[2] P. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Addison-Wesley Professional, 2007.

[3] "Jenkins," 2011, http://jenkins-ci.org.

[4] "Hudson," 2011, http://hudson-ci.org.

[5] "CruiseControl," 2011, http://cruisecontrol.sourceforge.net.

[6] P. Dutot, G. Mounié, D. Trystram *et al.*, "Scheduling parallel tasks: Approximation algorithms," *Handbook of Scheduling: Algorithms, Models, and Performance Analysis, JY-T. Leung (Eds.)*, 2004.

[7] F. Suter, F. Desprez, and H. Casanova, "From heterogeneous task scheduling to heterogeneous mixed parallel scheduling," in *Euro-Par 2004 Parallel Processing*, ser. Lecture Notes in Computer Science, M. Danelutto, M. Vanneschi, and D. Laforenza, Eds. Springer Berlin / Heidelberg, 2004, vol. 3149, pp. 230–237.

[8] T. N'Takpé, F. Suter, and H. Casanova, "A comparison of scheduling approaches for mixed-parallel applications on heterogeneous platforms," in *Parallel and Distributed Computing, 2007. ISPDC '07. Sixth International Symposium on*, july 2007, p. 35.

[9] K. Aida and H. Casanova, "Scheduling mixed-parallel applications with advance reservations," *Cluster Computing*, vol. 12, no. 2, pp. 205–220, 2009.

[10] "Hubble, a simulator for the Nix/Hydra build tools," 2011, http://hubble.gforge.inria.fr.

[11] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: a Generic Framework for Large-Scale Distributed Experiments," in *10th International Conference on Computer Modeling and Simulation (UKSIM 2008)*. Washington, DC, USA: IEEE Computer Society, april 2008, pp. 126–131. [Online]. Available: http://hal.inria.fr/inria-00260697/en/

[12] A. Giersch, Y. Robert, and F. Vivien, "Scheduling tasks sharing files on heterogeneous master–slave platforms," *Journal of Systems Architecture*, vol. 52, no. 2, pp. 88–104, 2006.

[13] B. Ucar, C. Aykanat, K. Kaya, and M. Ikinci, "Task assignment in heterogeneous computing systems," *Journal of parallel and Distributed Computing*, vol. 66, no. 1, pp. 32–46, 2006.

[14] B. Hong and V. Prasanna, "Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput," *IEEE IPDPS*, 2004.

[15] A. B. Downey, "TCP self-clocking and bandwidth sharing," *Computer Networks*, vol. 51, no. 13, pp. 3844–3863, 2007.

[16] D. Abendroth, H. van den Berg, and M. Mandjes, "A versatile model for TCP bandwidth sharing in networks with heterogeneous users," *AEU - International Journal of Electronics and Communications*, vol. 60, no. 4, pp. 267–278, 2006.

[17] Y. Zhu, A. Velayutham, O. Oladeji, and R. Sivakumar, "Enhancing TCP for networks with guaranteed bandwidth services," *Computer Networks*, vol. 51, no. 10, pp. 2788–2804, 2007.

[18] S. Chakrabarti, J. Demmel, and K. Yelick, "Modeling the benefits of mixed data and task parallelism," in *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*. ACM, 1995, pp. 74–83.

[19] D. G. Feitelson, "Scheduling parallel jobs on clusters," *High Performance Cluster Computing*, vol. 1, pp. 519–533, 1999.

[20] M. Drozdowski, "Scheduling Parallel Tasks—Algorithms and Complexity," in *Handbook of Scheduling*. Boca Raton, FL, USA: CRC Press, 2004, ch. 25.

[21] P. Dutot, G. Mounié, and D. Trystram, "Scheduling Parallel Tasks—Approximation Algorithms," in *Handbook of Scheduling*. Boca Raton, FL, USA: CRC Press, 2004, ch. 26.

[22] J. Blazewicz, M. Kovalyov, M. Machowiak, D. Trystram, and J. Weglarz, "Preemptable malleable task scheduling problem," *IEEE Transactions on Computers*, pp. 486–490, 2006.

[23] J. Turek, J. Wolf, and P. Yu, "Approximate algorithms scheduling parallelizable tasks," in *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, 1992, pp. 323–332.

[24] R. Lepère, D. Trystram, and G. Woeginger, "Approximation algorithms for scheduling malleable tasks under precedence constraints," *International Journal of Foundations of Computer Science*, vol. 13, no. 4, p. 613, 2002.

[25] P.-F. Dutot, T. N'Takpé, F. Suter, and H. Casanova, "Scheduling Parallel Task Graphs on (Almost) Homogeneous Multicluster Platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 7, pp. 940–952, 2008.

[26] G. N. Srinivasa Prasanna and B. R. Musicus, "Generalised multiprocessor scheduling using optimal control," in *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '91. New York, NY, USA: ACM, 1991, pp. 216–228. [Online]. Available: http://doi.acm.org/10.1145/113379.113399

[27] O. Sinnen and L. Sousa, "Communication contention in task scheduling," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 16, no. 6, pp. 503–515, june 2005.

[28] E. Dolstra, A. Löh, and N. Pierron, "NixOS: A purely functional Linux distribution," *Journal of Functional Programming*, no. 5-6, pp. 577–615, november 2010. [Online]. Available: http://nixos.org/nixos/docs.html

[29] E. Dolstra, M. de Jonge, and E. Visser, "Nix: A safe and policy-free system for software deployment," in *Proceedings of the 18th Large Installation System Administration Conference (LISA '04)*. Atlanta, Georgia, USA: USENIX, november 2004, pp. 79–92. [Online]. Available: http://nixos.org/nix/docs.html

[30] "Hydra," 2011, http://nixos.org/hydra.

[31] P. Miller, "Recursive make considered harmful," *Australian UNIX and Open Systems User Group Newsletter*, vol. 19, no. 1, pp. 14–25, 1997.

[32] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Task scheduling algorithms for heterogeneous processors," in *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth*, 1999, pp. 3–14.

BIBLIOGRAPHIES

Olivier Beaumont received his PhD degree from the University of Rennes in 1999. Between 1999 and 2006, he was assistant professor at Ecole Normale Superieure de Lyon and then at ENSEIRB in Bordeaux. In 2004, he defended his "habilitation a diriger les recherches" and was appointed

as Senior Scientist at INRIA in 2007. His research interests focus on the design of parallel and distributed algorithms, overlay networks on large scale heterogeneous platforms and combinatorial optimization.

Nicolas Bonichon received his PhD degree from the University of Bordeaux in 2002. He has been holding a position as assistant professor at University of Bordeaux since 2004. His research interests include distributed algorithms, compact data structure, graph drawing and enumerative combinatorics.

Ludovic Courtès received his PhD (entitled "Cooperative Data Backup for Mobile Devices") from the University of Toulouse in 2007. From then, he works as a research engineer at INRIA Bordeaux Sud-Ouest. His research interests focus on the design of efficient parallel algorithms and runtime systems.

Xavier Hanin received his MsC degree from the ENSERB in 1999. After several years as a freelancer during which he worked for companies as diverse as PSA, EMC, Stanford University, SAS, Halliburton or LinkedIn, he joined 4SH, a french consulting company in 2008. He is the creator of Apache Ivy, a member of Apache Ant Project Management Comittee and the creator of Xooctory, an open source Continuous Integration server.

Eelco Dolstra is a postdoc at the Software Engineering Research Group in the Department of Software Technology, Delft University of Technology. Previously, he was a PhD student and postdoc in the Center for Software Technology at Utrecht University.