Mapping Tightly-Coupled Applications on Volatile Resources

Henri Casanova[1], Fanny Dufossé[2], Yves Robert[2,3] and Frédéric Vivien[2]
1. Univ. of Hawai‘i at Manoa, Honolulu, USA, henric@hawaii.edu
2. Ecole Normale Supérieure de Lyon,& INRIA, France
{FannyDufosse|Yves.Robert|Frederic.Vivien}@ens-lyon.fr
3. University of Tennessee Knoxville, USA

***Abstract*—Platforms that comprise volatile processors, such as desktop grids, have been traditionally used for executing independent-task applications. In this work we study the scheduling of tightly-coupled iterative master-worker applications onto volatile processors. The main challenge is that workers must be simultaneously available for the application to make progress. We consider two additional complications: one should take into account that workers can become temporarily reclaimed and, for data-intensive applications, one should account for the limited bandwidth between the master and the workers.**

In this context, our first contribution is a theoretical study of the scheduling problem in its off-line version, i.e., when processor availability is known in advance. Even in this case the problem is NP-hard. Our second contribution is an analytical approximation of the expectation of the time needed by a set of workers to complete a set of tasks and of the probability of success of this computation. This approximation relies on a Markovian assumption for the temporal availability of processors. Our third contribution is a set of heuristics, some of which use the above approximation to favor reliable processors in a sensible manner. We evaluate these heuristics in simulation. We identify some heuristics that significantly outperform their competitors and derive heuristic design guidelines.

I. Introduction

In this paper we study the problem of scheduling parallel applications onto volatile processors. We target typical scientific iterative applications in which a master process parallelizes the execution of each iteration across worker processes. Each iteration requires the execution of a fixed number of tasks, with a global synchronization at the end of each iteration.

We consider a platform that consists of processors that alternate between periods of availability and periods of unavailability. When available each processor runs a worker process, and a master process can choose to enroll a subset of these workers to participate in the application execution. Worker unavailability can be due to software faults, in which case unavailability may last only the time of a reboot. A hardware failure can lead to a longer unavailability period, until a repair is completed and followed by a reboot. We consider a third source of processor unavailability, which comes from cycle-stealing scenarios: when a processor is contributed to the platform by an individual owner, this owner can reclaim it at any time without notice for some unknown length of time. A difference here is that the processor is merely preempted (as opposed to being terminated) until the processor is no longer reclaimed. A worker process on this processor can later resume its computation. Accordingly, we use a 3-state availability model: UP (available), $DOWN$ (crashed, computation is lost) and $RECLAIMED$ (preempted, but computation can resume later). Our platform model also accounts for the fact that, due to bandwidth limitation, the master is only able to communicate simultaneously with a limited number of workers (to send them the application program as well as task data). This limitation corresponds to the bounded multi-port model [1]. It turns out that limiting the communication capacity of the master dramatically complicates the design of scheduling strategies. But without this limitation, it would be in principle possible to enroll thousands of new processors at each iteration, which is simply not feasible in practice even if these many processors are available.

Given the above application and platform models, and given a deadline (typically expressed in hours or days), the scheduling problem under study is that of maximizing the expected number of application iterations successfully completed before the deadline. Informally, during each iteration, one must use the “best” processors among those that are simultaneously UP; these could be the fastest ones, or those expected to remain UP for the longest time. In addition, with processors failing, becoming reclaimed, and becoming UP again later, one has to decide when and how to change the set of currently enrolled processors. Each such change comes at a price: first, the application program needs to be sent to newly enrolled processors, thereby consuming some of the master's bandwidth; second, and more importantly, iteration computation that was only partially completed is lost due to the tight coupling of tasks.

Our contribution in this work is threefold. First, we determine the complexity of the off-line scheduling problem, i.e., when processor availability is known in advance. Even with such knowledge the problem is NP-hard. Second, we compute approximations of the expectation of the time needed by a set of processors to complete a set of tasks and of the probability that this computation succeeds. These approximations provide a sound basis for making sensible scheduling decisions. Third, we design several on-line heuristics that we evaluate in simulation. Some of these contributions assume Markovian processor availability, which is not representative of real-world platforms but provides a tractable framework for obtaining theoretical and experimental results in laboratory conditions. Due to lack of space we refer the reader to the companion research report [2] for a worked-out example, related work, statements and proofs of all NP-completeness results, and complete experimental results.

II. Models and assumptions

A. Application model

We consider an application that performs a sequence of iterations. Each iteration consists of executing m tasks and ends with a global synchronization. All m tasks are identical (in terms of computational cost) and communicate throughout the iteration execution. Therefore, all tasks must make progress at the same rate. If a task is terminated prematurely (due to a worker failure), all computation performed so far for the current iteration is lost, and the entire iteration has to be restarted. If a task is suspended (due to a worker becoming temporarily reclaimed), then the entire execution of the iteration is also suspended. Due to the global synchronization, there is no overlap between communication of task data and computation, but only between communication between tasks and computation. We thus consider that an iteration proceeds in two phases: a communication phase and a computation phase. Finally, before being able to compute, a worker must acquire the application code once (e.g., binary executable, byte code), of constant size V_{prog} in bytes, and the input data for each task and iteration, of constant size V_{data} in bytes. Data messages depend on tasks but have identical size.

B. Platform model

The platform comprises p processors, or workers. Worker P_q, $q = 1, \ldots, p$, can be in one of three states (UP, $\mathit{RECLAIMED}$ or DOWN), and transitions between these states occur for each processor at each time-slot independently of the other processors. More precisely:

- Any UP processor can become DOWN or $\mathit{RECLAIMED}$.
- Any UP or $\mathit{RECLAIMED}$ processor can become DOWN. It then loses the application program and all the data for its current tasks. If it was computing some of these tasks, these computations are lost.
- Any UP processor can become $\mathit{RECLAIMED}$. The processor does not lose any state. If it was receiving the application program or data for a task, the communication is temporarily suspended. If it was computing a task, the computation on *all* processors is temporarily suspended.

We denote by $\mathcal{S}_q$ the vector that gives the state of P_q at each time-slot starting with time-slot 0.

P_q can compute a task in w_q time-slots if it remains UP. If $w_q = w$ for each processor P_q, then the processors are homogeneous. The master has network bandwidth BW and communicates with a worker with bandwidth bw, meaning that we assume same capacity links from the master to each worker. Here we equate bandwidth with data transfer rate, acknowledging that in practice the data transfer rate is a fraction of the physical bandwidth. Let n_{prog} be the number of workers receiving the program at time t, and let n_{data} be the number of workers receiving the input data of a task at time t. The constraint on the master's bandwidth writes $n_{\text{prog}} + n_{\text{data}} \leq n_{\text{com}} = \lfloor BW/bw \rfloor$, where n_{com} is the maximum number of. processors that the master can communicate with (sending either program or input data) at each time-slot. Indeed, consider a worker on processor P_q that is communicating at time t. Either P_q is receiving the program, or it is receiving data for a task. In both cases, it does this at data transfer rate bw. Overall, the master can execute only a limited number n_{com} of such communications simultaneously. The time for a worker to receive the program is $T_{\text{prog}} = V_{\text{prog}}/bw$, and the time to receive the data is $T_{\text{data}} = V_{\text{data}}/bw$. For simplicity we assume that T_{prog} and T_{data} consist of integral numbers of time-slots. We also assume that the master is always UP, which can be enforced by using, for instance, two dedicated servers with a primary backup mechanism.

C. Application execution model

Let $\mathit{config}(t)$ denote the set of workers enrolled by the master, or *configuration*, at time t. The configuration is determined by an *application scheduler*, and in this work we propose algorithms to be used by this scheduler. To complete an iteration, enrolled workers must progress concurrently throughout the computations. One worker may be assigned several tasks and execute them concurrently if it has enough memory to do so. Formally, we define for each worker P_q a bound μ_q on the maximum number of tasks that it can execute concurrently. We assume that $\sum_{q=1}^{p} \mu_q \geq m$, otherwise the configuration cannot execute the application. The m tasks are mapped onto $k \leq m$ workers. Each enrolled worker P_q is assigned x_q tasks, where $\sum_{q=1}^{k} x_q = m$. To be able to compute their tasks, the k enrolled workers must have received the application program and all necessary data. More precisely: (i) each enrolled worker P_q must receive the program, unless it has received it at some previous time and has not be $DOWN$ since then; (ii) in addition, each worker P_q must receive a number x_q of data messages (one per task) from the master. Suppose that since the begin of the current iteration P_q has received x'_q data messages. At least $(x_q - x'_q)T_{\text{data}}$ time-slots are needed for this communication, and likely more since the master can be engaged in at most n_{com} concurrent communications.

Overall, the computation can start at a time t only if each of the k enrolled workers is in the UP state, has the program, has the data of all its allocated tasks, and has never been in the $DOWN$ state since receiving these messages. Because tasks must proceed in locked steps, the execution goes at the pace of the slowest worker. Hence the computation of an iteration requires $\max_q(x_q w_q)$ time-slots of concurrent computations (not necessarily consecutive, due to workers possibly being reclaimed). Consider the interval of time between time t_1 and time $t_2 = t_1 + \max_q(x_q w_q) + t' - 1$ for some t'. For the iteration to be successfully completed by time t_2, between t_1 and t_2 there must be $\max_q(x_q w_q)$ time-slots for which all enrolled workers are simultaneously UP, and there may be t' time-slots during which one or more workers are $RECLAIMED$.

The scheduler may choose a new configuration at each time t. If at least one worker in $config(t)$ becomes $DOWN$, the scheduler must select another configuration and restart the iteration from scratch. Even if all workers in $config(t)$ are UP, the scheduler may decide to change the configuration because more desirable (i.e., faster, more reliable) workers have become available. Let P_q be a newly enrolled worker at that point, i.e., $P_q \in config(t+1) \setminus config(t)$. P_q needs to receive the program unless it already has a copy of it and has not been $DOWN$ since receiving it. In all cases, P_q needs to receive task data, i.e., x_q messages of V_{data} bytes. This holds true even if P_q had been enrolled at time $t' < t$ but was un-enrolled since then. In other words, any interrupted communication must be resumed from scratch if the worker became $DOWN$ or was removed from the configuration.

III. Off-line complexity

The scheduling problem is to maximize the expected number of completed application iterations before time N, where N is a specified deadline. In the off-line version of this problem, one assumes full knowledge of future worker states. In other words, $\mathcal{S}_q[j]$ is known for $1 \leq q \leq p$ and $1 \leq j \leq N$. It is shown in [2] that the simplest off-line and deterministic versions of the problem are NP-hard.

IV. Analytical Approximations

In this section, we compute the expectation of the time needed by a configuration to compute a given workload conditioned on this computation being successful (i.e., with no worker becoming $DOWN$), as well as the probability of success. Intuitively, these quantities seem relevant for developing scheduling heuristics that account for the need for workers to be UP simultaneously, and for workers that can become temporarily $RECLAIMED$. To compute the above expectation and probability, we introduce a Markov model of processor availability. The availability of processor P_q is described by a 3-state recurrent aperiodic Markov chain, defined by 9 probabilities: $P_{i,j}^{(q)}$, with $i, j \in \{u, r, d\}$, is the probability for P_q to move from state i at time t to state j at time $t+1$, which does not depend on t.

A. Probability of success and expected duration of a computation

Consider a set S of workers all in the UP state at time 0. This set is assigned a workload that requires W time-slots of simultaneous computation. To complete this workload successfully, all the workers in S must be simultaneously UP during another $W-1$ time-slots. They can possibly become $RECLAIMED$ (thereby temporarily suspending the execution) but must never become $DOWN$ in between. What is the probability of the workload being completed? And, if it is successfully completed, what is the expectation of the number of time-slots until completion?

Definition 1: Knowing that all processors in a set S are UP at time-slot t_1, let $\mathbf{P}_{+}^{(\mathbf{S})}$ be the conditional probability that they will all be UP simultaneously at a later time-slot, without any of them going to the $DOWN$ state in between. Formally, knowing that $\forall P_q \in S$, $\mathcal{S}_q[t_1] = u$, $P_{+}^{(S)}$ is the conditional probability that there exists a time $t_2 > t_1$ such that $\forall P_q \in S$, $\mathcal{S}_q[t_2] = u$ and $\mathcal{S}_q[t] \neq d$ for $t_1 < t < t_2$.

Definition 2: Let $\mathbf{E}^{(\mathbf{S})}(\mathbf{W})$ be the conditional expectation of the number of time-slots required by a set of processors S to complete a workload of size W knowing that all processors in S are UP at the current time-slot t_1 and none will become $DOWN$ before completing this workload. Formally, knowing that $\mathcal{S}_q[t_1] = u$, and that there exist $W-1$ time-slots $t_2 < t_3 < \cdots < t_W$, with $t_1 < t_2$, $\mathcal{S}_q[t_i] = u$ for $i \in [2, W]$, and $\mathcal{S}_q[t] \neq d$ for $t \in [t_1, t_W]$, $E^{(S)}(W)$ is the expectation of $t_W - t_1 + 1$ conditioned on success.

Theorem 4.1: It is possible to approximate the values of $P_{+}^{(S)}$ and $E^{(S)}(W)$ numerically up to an arbitrary precision ε in fully polynomial time.

B. Probability of success and expected duration of a communication

Similar approximations cannot be obtained for communications due to complexity added by the n_{com} constraint. Instead, we resort to a coarser approximation as explained hereafter. Let S be a set of enrolled workers. For worker $P_q \in S$, let n_q be the number of time-slots of communication needed to receive the application program and all the data of its allocated tasks. Suppose first that $|S| \leq n_{\text{com}}$. In this case, the expected communication time on worker P_q, E_q, can be estimated precisely reusing the result in the previous section: $E_q = E^{(P_q)}(n_q)$. We then estimate the expected communication time of the current configuration as $E_{comm}^{(S)} = \max_{P_q \in S}\{E^{(P_q)}(n_q)\}$. In the case $|S| \geq n_{\text{com}}$, obtaining an estimate close to the actual expected communication time seems out of reach. Instead, we use a coarser estimation: $E_{comm}^{(S)} = \max\left\{\max_{P_q \in S}\left\{E^{(P_q)}(n_q)\right\}, \frac{\sum_{P_q \in S} n_q}{n_{\text{com}}}\right\}$.

Let $P_{ND}^{(P_q)}(t)$ denote the probability that worker P_q that was UP at time t' does not become $DOWN$ between time t' and time $t' + t$. The probability of success is then estimated as $P_{comm}^{(S)} = \prod_{P_q \in S} P_{ND}^{(P_q)}(E_{comm}^{(S)})$. The expression for $P_{comm}^{(S)}$ does not take into account the time needed after the end of all communications for all workers to be UP simultaneously. The probability of success of an iteration is estimated by multiplying the probability of success of the communications and the probability of success of the computations.

V. On-line heuristics

We propose heuristics for solving the on-line version of the scheduling problem, i.e., assuming no knowledge of future processor states. Conceptually, we distinguish between two classes of heuristics. *Passive heuristics* conservatively keep current processors active as long as possible: the current configuration is changed only when one of the enrolled processors becomes $DOWN$. In this case, all previously executed work is lost. However, a worker that has not become $DOWN$ but has already received task data, can reuse that data if the scheduler reassigns tasks to it. *Proactive heuristics* allow for a complete reconfiguration even if no worker fails, possibly aborting ongoing computation if a better configuration is found. This makes it possible for an iteration to never complete. A criterion must thus be derived to decide whether and when such an aggressive reconfiguration is worthwhile. Our proactive heuristics are defined by a pair (criterion, passive heuristic). When a new configuration is computed using the heuristic, it is compared to the current configuration according to the criterion. If the new configuration is better than the current one, then it is launched, leading to new communications and task allocations. Otherwise, the execution continues with the current configuration for an additional time slot.

A. Passive heuristics

Passive heuristics assign tasks to workers, which must be in the UP state, one by one until m tasks are assigned. Each task is assigned to a worker according to a criterion that defines the heuristic. As described hereafter, we consider four different criteria: probability of success, expected completion time, estimated yield, and estimated apparent yield.

• **IP (Incremental: Probability of success) –** This heuristic attempts to find configurations with high probability of success. The next task is assigned to the worker such that the probability of success of all currently assigned tasks (including the new one) is maximized. More precisely, consider the set S of workers with at least one task already assigned. For each worker P_q, either in S or not, we compute the probability $P^{(S)}(q)$ of success of the communication and the computation if the additional task is assigned to P_q, using the results of Section IV: $P^{(S)}(q) = P^{(S \cup \{P_q\})}(W_q) \times P_{comm}^{(S \cup \{P_q\})}$ with W_q the maximal load in $S \cup \{P_q\}$ with an additional task on P_q. We assign the next task to worker P_{q_0}, with $q_0 = \text{ArgMax}\left\{P^{(S)}(q)\right\}$. This natural idea of the most reliable workers has been used for scheduling independent tasks in [3], [4], [5].

• **IE (Incremental: Expected completion time) –** This heuristic attempts to find fast configurations, without considering reliability. The next task is assigned to the worker that minimizes the expected execution time of the iteration. More precisely, consider the set S of workers with at least one task already assigned. For each worker P_q, either in S or not, we compute the expected communication time $E_{comm}^{(S \cup \{P_q\})}$ and the expected computation time $E^{(S \cup \{P_q\})}(W_q)$ with an additional task on P_q. We obtain the expected duration of the iteration $E^{(S)}(q) = E_{comm}^{(S \cup \{P_q\})} + E^{(S \cup \{P_q\})}(W_q)$. We assign the next task to worker P_{q_0}, with $q_0 = \text{ArgMin}\left\{E^{(S)}(q)\right\}$. This idea of picking the fatest workers has been used for scheduling independent tasks in [6].

• **IY (Incremental: Expected yield) –** This heuristic assigns the next task to the worker that maximizes the *yield* of the configuration. The yield is the expected value of the inverse of the execution time of the current iteration, which we estimate as follows. For a given configuration with probability of success P and expected completion time E for an iteration that has already been running for t time slots, the yield is estimated as $Y = \frac{P}{E+t}$. Intuitively, we expect the yield to achieve a trade-off between reliability (probability of success) and execution speed. Consider the set S of workers with at least one task already assigned. For each processor P_q, either in S

or not, we compute the expected yield with an additional task on P_q: let $P^{(S)}(q)$ be the probability computed for heuristic IP, $E^{(S)}(q)$ be the expected completion time computed for heuristic IE, and t be the time spent since the beginning of the current iteration. We assign the next task to worker P_{q_0}, with $q_0 = \text{ArgMax}\left\{\frac{P^{(S)}(q)}{t+E^{(S)}(q)}\right\}$. This general idea of trading off reliability for speed has been used in the context of independent tasks in many previous works [6], [4], [7], [8].

• **IAY (Incremental: Expected apparent yield) –** The yield takes into account the time already spent in the current iteration. It could be worthwhile to consider only future work, i.e., the remaining time until iteration completion. To this end we define the *apparent yield* as $AY = \frac{P}{E}$. Using the same notations as for heuristic IY, we assign the next task to processor P_{q_0} with $q_0 = \text{ArgMax}\left\{\frac{P^{(S)}(q)}{E^{(S)}(q)}\right\}$.

B. Proactive heuristics

Consider an application executing on a platform using a passive heuristic H and criterion C at some time t. The configuration $\mathit{config}(t-1)$ was selected by H at time $t' \leq t-1$ because of a configuration change due to a proactive decision, due to a worker becoming $DOWN$, or due to the beginning of a new iteration. Let $\mathit{config}_1 = \mathit{config}(t') = \mathit{config}(t-1)$. At time t', the configuration was measured by criterion C with value c'. Suppose that by time t no worker in this configuration has failed. Between t' and t, some work may have been done: some communications may be in process or completed, and computations may have started. Consequently, the measure of this configuration given by C should be updated to account for the progress between t' and t. Let c be the updated value of criterion C for the current configuration. At step t, a new configuration is computed from scratch using heuristic H, as if no task were allocated to any worker. Let config_2 be this new configuration and c_2 its measure by C. If $c \geq c_2$, then the current configuration at time $t-1$ is kept for another time-slot: $\mathit{config}(t) = \mathit{config}_1$. Otherwise, the current configuration is interrupted, and the new configuration is $\mathit{config}(t) = \mathit{config}_2$.

For certain criterion choices, a heuristic could diverge and continually change the configuration, even with workers that are reliably UP. To avoid this divergence, proactive criteria have to respect the following constraint: a given configuration that has been running for $t+1$ time-slots must be better for the proactive criterion than the same configuration running for t time slots. With this constraint, all possible configurations are ordered by their value for the selected criterion at the beginning of the iteration, and a lower-ranked configuration in this order cannot be chosen to replace the current configuration. As the number of possible configurations is finite, no proactive heuristic can diverge. The four criteria used to define passive heuristics in the previous section meet this constraint. However, AY (Apparent Yield) leads to many (unnecessary) configuration changes before converging, while the other criteria should be stable. Hence, for the proactive criterion C, we only retain P (Probability of success), E (Expected completion time) and Y (Expected yield). Any passive heuristic H can be used as the building block for a proactive heuristic. We thus obtain 3×4 proactive heuristics named C-H where $C \in$ {P, E, Y} and $H \in$ {IP, IE, IY, IAY}.

VI. Conclusion

Unlike previous work that has considered loosely-coupled master-worker applications, in this study a single processor failure can have a dramatic effect on application execution. The requirement that the application can progress only when all enrolled processors are simultaneously available, dramatically complicates all scheduling decisions. By assuming a Markov model of processor availability, we have proposed polynomial time approximation schemes to compute the expected completion time of a computation, and its probability of success. We have then proposed many heuristics that are easily defined as combinations of two among four sensible metrics: probability of success, expected completion time, expected yield and expected apparent yield. All these heuristics have been extensively evaluated in simulations, see [2]. The main conclusion is that a proactive heuristic that selects processors to maximize expected execution time and changes configuration based on yield or probability of success is very promising.

Acknowledgments. This work was supported in part by the ANR *RESCUE* project.

References

[1] B. Hong and V. K. Prasanna, "Adaptive allocation of independent tasks to maximize throughput," *IEEE TPDS*, vol. 18, no. 10, pp. 1420–1435, 2007.

[2] H. Casanova, F. Dufossé, Y. Robert, and F. Vivien, "Mapping Tightly-Coupled Applications on Volatile Resources," ENS Lyon, Tech. Rep., May 2012. [Online]. Available: http://hal-ens-lyon.archives-ouvertes.fr/ensl-00697621

[3] T. Estrada, D. Flores, M. Taufer, P. Teller, A. Kerstens, and D. Anderson, "The Effectiveness of Threshold-Based Scheduling Policies in BOINC Projects," in *Proc. of e-Science'06*, 2006.

[4] C. Anglano, J. Brevik, M. Canonico, D. Nurmi, and R. Wolski, "Fault-aware scheduling for Bag-of-Tasks applications on Desktop Grids," in *Proc. of Grid Computing*, 2006, pp. 56–63.

[5] J. Wingstrom and H. Casanova, "Probabilistic Allocation of Tasks on Desktop Grids," in *Proc. of PCGrid*, 2008.

[6] D. Kondo, A. Chien, and H. Casanova, "Resource Management for Rapid Application Turnaround on Enterprise Desktop Grids," in *Proc. of SC*, 2004.

[7] E. Byun, S. Choi, M. Baik, J. Gil, C. Park, and C. Hwang, "MJSA: Markov job scheduler based on availability in desktop grid computing environment," *FGCS*, vol. 23, pp. 616–622, 2007.

[8] E. Heien, D. Anderson, and K. Hagihara, "Computing Low Latency Batches with Unreliable Workers in Volunteer Computing Environments," *Journal of Grid Computing*, vol. 7, no. 4, pp. 501–518, 2009.