On-Line Scheduling of Parallelizable Jobs

Christophe Rapine[1], Isaac D. Scherson[2] and Denis Trystram[1]

[1] LMC - IMAG, Domaine Universitaire, BP 53, 38041 GRENOBLE cedex 9, France
[2] University of California, Irvine, Department of Information and Computer Science, Irvine, CA 92717-3425, U.S.A.

Abstract. We consider the problem of efficiently executing a set of parallel jobs on a parallel machine by effectively scheduling the jobs on the computer's resources. This problem is one of optimization of resource utilization by parallel computing programs and/or the management of multi-users requests on a distributed system. We assume that each job is parallelizable and can be executed on any number of processors. Various on-line scheduling strategies of time/space sharing are presented here. The goal is to assign jobs to processors in space and time such that the total execution time is optimized.

1 Introduction

1.1 Description of the Computational Model

A model of computation is a description of a computer together with its program execution rules. The interest of a model is to provide a high level and abstract vision of the real computer world such that the design and the theoretical development and analysis of algorithms and of their complexity are possible. The most commonly adopted parallel computer model is the PRAM model [4]. A PRAM architecture is a *synchronous* system consisting of a global shared-memory and an unbounded number of processors. Each processor owns a small local memory and is able to execute any basic instruction in one unit of time. After each basic computational step a synchronization is done and each processor can access a global variable (for reading or writing) in one time unit. This model provides a powerful basis for the theoretical analysis of algorithms. It allows in particular the classification of problems in term of their intrinsic parallelism. Basically, the parallel characterization of a problem $\mathcal{P}$ is based on the order of magnitude of the time $T_\infty(n)$ of the best known algorithm to solve an instance of size n. $T_\infty(n)$ provides a lower bound for the time to solve $\mathcal{P}$ in parallel. However, in addition to answer the question *how fast the problem can be solved ?*, one important characteristic is *how efficient is the parallelization ?*, i.e. what is the order of magnitude of the number of processors $P_\infty(n)$ needed to achieve $T_\infty(n)$. This is an essential parameter to deal with in order to implement an algorithm on an actual parallel system with limited resources. This is best represented by the ratio μ_∞ between the *work* of the PRAM algorithm, $\mathcal{W}_\infty(n) = T_\infty(n).P_\infty(n)$ and the total number of instructions in the algorithm $\mathcal{W}_{tot}$. In this paper we consider a physically-distributed logically-shared memory machine composed of m

identical processors linked by an interconnection network. In the light of modern fast processor technology, compared to the PRAM model, communications and synchronizations become the most expensive operations and must be considered as important as computations.

1.2 Model of Jobs

For reasons of efficiency, algorithms must be considered at a high level of *granularity*, i.e. elementary operations are to be grouped into *tasks*, linked by precedence constraints to form programs. A general approach leads to write a parallel algorithm for v virtual processors with $m \ll v \ll P_\infty$. The execution of the program needs a scheduling policy a, static and/or dynamic, which directly influences the execution time T_m of the program. We define the *work* of the application on m processors as the space-time product $\mathcal{W}_m = m \times T_m$. Ideally speaking, one may hope to obtain a parallel time equal to T_{seq}/m, where T_{seq} denotes the sequential execution time of the program. This would imply a conservation of the work. $\mathcal{W}_m = \mathcal{W}_{tot}$. Such a linear speed-up is hard to reach even with an optimal scheduling policy, due mainly to communication delays and the intrinsic non-parallelism of the algorithm (even in the PRAM model we may have $\mu_\infty \gg 1$). We introduce the ratio μ_m to represent the "penalty" of a parallel execution with respect to a sequential one and define it as:

$$\mu_m = \frac{\mathcal{W}_m}{\mathcal{W}_{tot}} = \frac{m.T_m}{T_{seq}}$$

1.3 Discussion on Inefficiency

The *Inefficiency* factor (μ_m) is an experimental parameter which reflects the quality of the parallel execution, even though it can be computed theoretically for an application and a scheduling. Let us remark that the parameter μ_m depends theoretically on n and on the algorithm itself. Practically speaking, most existing implementations of large industrial and/or academic scientific codes show a small constant upper bound. However, we can assume some general properties about the inefficiency that seem realistic. First, it is reasonable to assume that the work $\mathcal{W}_p$ is an increasing function of p. Hence inefficiency is also an increasing function of p : $\mu_p \leq \mu_{p+1}$ for any p. Moreover, we can also assume that the parallel time T_p is a non-increasing function of p, all the less for "reasonable" number of processors. Hence, $(p+1)T_{p+1} \leq (1+\frac{1}{p})p\, T_p \Rightarrow \mu_{p+1} \leq (1+\frac{1}{p})\mu_p$. Developing the inequality, for any number of processors $p, q \geq 1$ we get $\mu_{p+q} \leq \frac{p+q}{p}\mu_p$. In other words, $\frac{\mu_p}{p}$ is a non-increasing function of p. As a consequence, and as μ_1 is 1 by definition, μ_p is bounded by p, which intuitively means that in the worst case the execution of the application on p processors leads in fact to a sequential execution, i.e. the application is not parallel.

1.4 Competitivity

The factor μ_p is a performance characteristic of the parallel execution of a program. It can be easily computed *a posteriori* after an execution in the same way than speedup or efficiency. A more theoretical and precise performance guarantee is the *competitive ratio*: an algorithm is said to be $\rho(m)$ competitive if for any instance $\mathcal{I}$ to be scheduled on m processors, $T_m(\mathcal{I}) \leq \rho(m).T_m^*(\mathcal{I})$, where $T_m(\mathcal{I})$ denotes the time of the execution produced by the algorithm and $T_m^*(\mathcal{I})$ is the optimal time on m processors. Our goal is, given multiprocessor tasks individually characterized by an inefficiency μ_p when executed on p processors, to determine the competitive ratio for the whole system.

2 Scheduling Independent Jobs

We define an application as a set of jobs linked by precedence constraints. Most parallel programming paradigms assume that a parallel program produces independent subproblems, to be treated concurrently. Consider the following problem: *Given a m-processor parallel machine and a set of N independent parallelizable jobs $(J_i)_{1,N}$ whose duration is not known until they terminate, what competitive ratio can we guarantee for the execution?* We assume that each job J_i has a certain inefficiency μ_p^i when executed on p processors and that it can be preempted. The question can be restated as: *Given a set of inefficiencies , what is the competitive ratio we may hope for from an on-line scheduling strategy?* In the following, we denote by $T^*(\mathcal{I})$ the optimal execution time and by $T_a(\mathcal{I})$ the one of the scheduling strategy a. The competitive ratio ρ_a^* of a is defined as the maximum on all the instances $\mathcal{I}$ of the fraction $T_a(\mathcal{I})/T^*(\mathcal{I})$. Another interesting performance guarantee is the comparison between $T_a(\mathcal{I})$ and the optimal execution time $T_S^*(\mathcal{I})$ when the parallelization of the jobs is not allowed, i.e. jobs are computed one at a time on single processors. We denote by ρ_a^{seq} the maximum on $\mathcal{I}$ of $T_a(\mathcal{I})/T_S^*(\mathcal{I})$. This competitive ratio will highlight the (possible) gain to allow the multiprocessing of jobs compared to the classical approach we present below where jobs are purely sequential.

2.1 Two Extreme Strategies: Graham and Gang Scheduling

The scheduling of multiprocessor jobs has recently become a matter of much research and study. In off-line scheduling theory, several articles have been published by J.Błażewicz and al. [2, 1], considering multiprocessor tasks but each one requiring a *fixed* number of processors for execution. If we look at the field of on-line scheduling, the most important contributions focused on the the case of one-processor tasks. In this classical approach any task requires only one processor for execution. The most famous algorithm is due to Graham [3] and consists simply in a greedy affectation of the tasks to the processors such that a processor cannot be idle if there is a remaining unscheduled job. Its competitive ratio is $2 - \frac{1}{m}$, which is the best possible ratio for any deterministic scheduling strategy when the computation costs of the tasks are not known till their completion. At the other end of the spectrum, the Gang scheduling analyzed recently in Scherson [5] assumes multiprocessor jobs and schedule them allowing the whole machine, m processors, to all the tasks. For the *Happiness* function defined by Scherson, the Gang strategy is proved to be the best one for preemptive multiprocessor job scheduling. Certainly, if we are concerned with the competitive ratio, the best strategy will be a compromise between allowing one processor per job (Graham) and the whole machine to any job (Gang). In particular we hope that multiprocessing of the job may improve the competitive ratio $(2 - \frac{1}{m})$ of Graham. In the following, let denote $\mu_p = \max_{1,N} \mu_p^i$.

Lemma 1. *The competitive ratio of Gang scheduling is equal to μ_m.*

Proof. Consider any instance $\mathcal{I}$. Let a_i be the work of job J_i, and $\mathcal{W}_{tot}$ the total amount of work of the program. We have $T_{Gang} = \sum_{i=1}^{N} \frac{\mu_m^i}{m} a_i \leq \frac{\mu_m}{m} \mathcal{W}_{tot}$ Noticing that $\mathcal{W}_{tot}/m$ is always a lower bound of T^*, it follows that $\rho^*_{Gang} \leq \mu_m$. Consider now a particular instance consisting of m identical jobs of work a. Gang scheduling produces an execution time of $\mu_m a$, while assigning one processor per job produces a schedule of length exactly a. As this schedule is one-processor job, it proves that $\rho^{seq}_{Gang} \geq \mu_m$, which gives the result. □

Lemma 2. *For Graham scheduling, $\rho^{seq}_{Graham} = 2 - \frac{1}{m}$ and $\rho^*_{Graham} \geq \frac{m}{\mu_m}$*

Proof. Let consider the scheduling of a single job of size a. Any one-processor scheduling delivers an execution time of a, while the optimal one is reached giving the whole machine to the job. Thus $T^* = \frac{\mu_m}{m} a$ and so $\rho^*_{Graham} \geq \frac{m}{\mu_m}$. □

3 SCHEduling Multiprocessors Efficiently: SCHEME

A Graham scheduling breaks into two phases: in the first one all processors are busy, getting an efficiency of one. Inefficiency happens when less than m tasks remain. Our idea is to preempt these tasks and to schedule them with a Gang policy in order to avoid idle time.

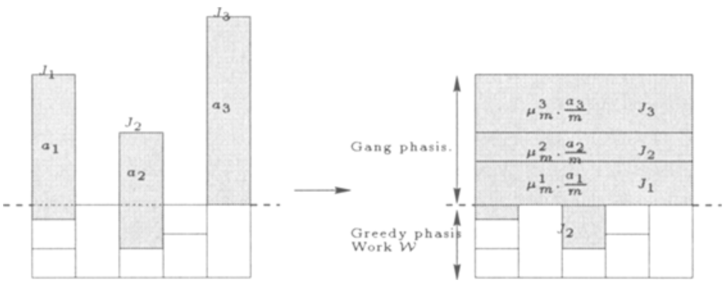
J_1
a_1
J_2
a_2
J_3
a_3
Gang phasis
Greedy phasis
Work W
$\mu_m^3 \cdot \frac{a_3}{m}$
J_3
$\mu_m^2 \cdot \frac{a_2}{m}$
J_2
$\mu_m^1 \cdot \frac{a_1}{m}$
J_1
2

Fig. 1. *Principle of a SCHEME scheduling*

3.1 Analysis of the Sequential Competitivity

Let's consider first time t in the Graham scheduling when less than m jobs are not completed. Let $J_1, \ldots, J_k$, $k \leq m-1$, be these unfinished jobs, and denote by $a_1, \ldots, a_k$ their remaining amount of work. Let $\mathcal{W} = m.t$ be the work executed between time 0 and t. We have the majoration $T_{sc}(m) \leq \frac{\mathcal{W}}{m} + \frac{\mu_m}{m} \sum_{i=1}^{i=k} a_i$. Introducing the total amount of work of the jobs, $\mathcal{W}_{tot} = \mathcal{W} + \sum_{i=1}^{i=k} a_i$, and the maximum amount of work remaining for a job, $a = max\{a_1, \ldots, a_k\}$, it follows that $T_{sc}(m) \leq \mathcal{W}_{tot}/m + \frac{\mu_m - 1}{m} \sum_{i=1}^{i=k} a_i \leq \mathcal{W}_{tot}/m + \frac{m-1}{m}(\mu_m - 1)a$

To obtain an expression of sequential competitivity , we need to minorate the one-processor optimal time. A lower bound is always $\mathcal{W}_{tot}/m$. As we consider a one-processor job scheduling, T_S^* is greater than the sequential time of any task. Hence we have $T_S^* > a$. We get the following lemma:

Lemma 3. *SCHEME has a sequential competitivity of* $(1 - \frac{1}{m})\mu_m + \frac{1}{m}$

Proof. It is sufficient to prove that the bound is tight. Consider the following instance composed of $m-1$ jobs of size m and m jobs of size 1. The optimal schedule allocate one processor per tasks of size m and assign all the small jobs to the last processor. It realizes an execution time $m = \mathcal{W}_{tot}/m$. A possible execution of SCHEME scheduling allocates first one small jobs per processor and then uses the gang strategy for the $m-1$ large jobs. It conducts to an execution time $T_{sc}(m) = 1 + (m-1)\mu_m$, which realizes the bound. □

3.2 Analysis of the Competitive Ratio

We determine here the competitive ratio of the SCHEME scheduling compared to an optimal multiprocessor jobs scheduling. Let consider an instance of the problem on m processors, and S^* an optimal schedule. We decompose S^* into temporal area slices $(A_i^*)_i$, each slice corresponding to the jobs compute between time t_{i-1} and t_i defined recursively as follow: $t_0 = 0$, and t_{i+1} is the maximal date such that in the time interval $[t_i, t_{i+1}[$ no job is preempted or completed. Notice that in the slice A_i^*, either all the jobs are computed on one processor, the slice is said "one-processored", or at least one job is executed on $p \geq 2$ processors. Let denote by A_i the area needed in the SCHEME schedule to compute the instructions of A_i^*. Notice that any piece of job in the SCHEME scheduling is either process on one or on m processors.

• Let A_i^* be a one-processored slice. Let denote by G_i the amount of instructions of A_i^* executed on m processors in A_i and by W_i the amount of instructions of A_i^* executed on one processor in A_i. By definition $A_i^* = G_i + W_i$. At least one of the jobs is entirely computed on one processor in the SCHEME schedule, hence $W_i \geq A_i^*/m$. In A_i, the area to compute the instruction of G_i is increased by a factor μ_m. Thus $A_i = \mu_m G_i + W_i = \mu_m A_i^* - (\mu_m - 1)W_i \leq \mu_m A_i^* - (\mu_m - 1)\frac{A_i^*}{m} \leq ((1 - \frac{1}{m})\mu_m + \frac{1}{m})A_i^*$. It follows that $A_i \leq \rho_{sc}^{seq}(m)A_i^*$.

• Let A_i^* be a multi-processored slice. There exists a job J computes on $p \geq 2$ processors in the slice. Let a be its amount of instruction and denote by

$B = A_i^* - \mu_p a$ the remaining area less J. In the SCHEME schedule, the area A_i needed to execute the instructions of A_i^* is at most $\mu_m(a+B) = \mu_m(A_i^* - (\mu_p - 1)a)$. Moreover $A_i^* \leq \frac{\mu_p}{p} a \times m$. Hence: $A_i \leq \mu_m(1 - \frac{\mu_p - 1}{\mu_p}\frac{p}{m})A_i^*$ The maximum ratio appears minimizing the term $\frac{\mu_p - 1}{\mu_p}p$. Noticing that p/μ_p and μ_p are increasing functions, maximal value is reached for $p = 2$. Hence: $A_i \leq (1 - \frac{2}{m} + \frac{2}{m\mu_2})\mu_m A_i^*$. We call the ratio factor $\rho_{sc}^{par}(m)$. For any slice A_i^* of S^*, we have the corresponding area in S bounded by $\max\{\rho_{sc}^{seq}(m),\ \rho_{sc}^{par}(m)\}.A_i^*$. Noticing that the SCHEME schedule contains no idle time, we get the following lemma:

Lemma 4. *The competitive ratio of SCHEME scheduling is equal to*

$$\rho_{sc}^*(m) = (1 - \frac{1}{m})\mu_m + \frac{1}{m}\max\{1, \frac{2-\mu_2}{\mu_2}\mu_m\}$$

Proof. To prove that the ratio is tight, consider the instance composed of $m-2$ jobs of size 1 and one job of size $(2/\mu_2)$. A possible schedule allocating one processor per unit job and two for the last job, completes at time 1. The SCHEME scheduling leads to time execution $\frac{\mu_m}{m}((m-2) + \frac{2}{\mu_2})$. □

4 Concluding Remarks

We are currently experimenting the mixed strategy SCHEME shown to be (theoretically) promising. Other more sophisticated scheduling strategies are also under investigation.

References

1. J. Błażewicz, P. Dell'Olmo, M. Drozdowski, and M.G. Speranza. Scheduling multiprocessors tasks on three dedicated processors. *Information Processing Letters*, 41:275–280, April 1992.
2. J. Błażewicz, M. Drabowski, and J. Węglarz. Scheduling multiprocessor tasks to minimize schedule length. *IEEE Transactions on Computers*, 35(5):389–393, 1986.
3. R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.
4. R.M. Karp. On the computational complexity of combinatorial problems. *Networks*, 5:45–68, 1975.
5. I. D. Scherson, R. Subramanian, V. L. M. Reis, and L. M. Campos. Scheduling computationally intensive data parallel programs. In *Placement dynamique et répartition de charge : application aux systèmes parallèles et répartis (École Française de Parallélisme, Réseaux et Système)*, pages 107–129. Inria, July 1996.