

# On-line adaptive parallel prefix computation

Jean-Louis Roch, Daouda Traoré, and Julien Bernard\*

Équipe MOAIS (CNRS-INRIA-INPG-UJF)  
Laboratoire d'Informatique de Grenoble  
38330 Montbonnot Saint Martin, France

[Julien.Bernard, Jean-Louis.Roch, Daouda.Traore]@imag.fr,  
<http://moais.imag.fr>

**Abstract.** We consider parallel prefix computation on processors of different and possibly changing speeds. Extending previous works on identical processors, we provide a lower bound for this problem. We introduce a new adaptive algorithm which is based on the on-line recursive coupling of an optimal sequential algorithm and a parallel one, non-optimal but recursive and fine-grain. The coupling relies on a work-stealing scheduling. Its theoretical performance is analysed on  $p$  processors of different and changing speeds. It is close to the lower bound both on identical processors and close to the lower bound for processors of changing speeds. Experiments performed on an eight-processor machine confirms this theoretical result.

## 1 Introduction

Given  $x_0, x_1, \dots, x_n$ , the prefix problem is to compute the  $n$  products  $\pi_k = x_0 \circ x_1 \circ \dots \circ x_k$  for  $1 \leq k \leq n$ , where  $\circ$  is an associative operation. Prefix computation is a common operation in many algorithms including the evaluation of polynomials and modular additions [3], packing problems, loop parallelization [9].

The iterative sequential prefix computation requires  $n$  operations  $\circ$ . However, any parallel prefix circuit of depth  $d$  contains at least  $2n - d$  operations  $\circ$  (see section 3). The minimal parallel time is  $\Omega(\log n)$  on a machine without concurrent write. Ladner and Fischer [7] proposed a parallel algorithm which takes a time of  $2 \log n$  and  $2n$  operations. Fich [4] proved that any algorithm of time  $\log n$  requires  $4n$  operations. Then Ladner-Fischer's algorithm is fine grain and asymptotically optimal on  $\frac{n}{\log n}$  processors. It can be scheduled on  $p < \frac{n}{\log n}$  identical processors in time  $\frac{2n}{p} + O(\log n)$ . Since it carries out  $2n$  operations, it is not optimal for a fixed  $p$ . Nicolau and Wang [9] showed that a strict lower bound for the parallel time on  $p$  identical processors is  $\left\lceil \frac{2n}{p+1} \right\rceil$  for  $n \geq \frac{p(p+1)}{2}$ . They provided an algorithm, based on a cutting in  $(p+1)$  blocks and a pipeline between blocks, which reaches this lower bound. Most implementations either on

---

\* This work is supported by the French government ANR ARA-SSIA BGPR/SafeScale and a France-Mali grant.

dedicated distributed architectures or circuits [3] are based on an off-line block partitioning, with a block size depending on  $p$ .

The drawback of such optimal algorithms for a fixed number  $p$  of processors is that the number of operations is at least  $2\frac{p}{p+1}$  times greater than the number  $n$  of operations performed by an optimal sequential algorithm. Thus, although optimal on  $p$  processors, those algorithms are not efficient on a machine with processors of different and possibly changing speeds. This is the practical case for a multi-processor machine concurrently used by several users, since the load of the processors varies during the execution. In this case, the scheduling must be on-line.

To resolve this problem, we use an on-line work-stealing (see section 2) implemented in Kaapi [6,8]. Bender and Rabin [1] extended this work-stealing to processors of changing speeds: they analyze the time of an algorithm with respect to  $\Pi_{ave}$ , the average speed per processor. On this model, we provide in section 3 a lower bound  $\frac{2n}{p \cdot \Pi_{ave} + \Pi_{max}}$  for computation time of parallel prefix, where  $\Pi_{max}$  is the maximal speed of a processor during execution. We prove this bound is tight on uniform processors: for  $n \gg p$ , a block algorithm – with an off-line partitioning based on the relative speeds of the processors – reaches it.

In order to suit to processors with changing speeds, section 4 presents our new adaptive algorithm that performs an on-line block partitioning without assumption on the processor speeds. It is based on the recursive coupling of a sequential optimal algorithm and a fine grain parallel one which is scheduled by work-stealing. Its execution time, including on-line scheduling overhead, is  $T_p \leq \frac{2n}{(p+1) \cdot \Pi_{ave}} + O\left(\frac{\log n}{\Pi_{ave}}\right)$  which is close to the lower bound, and asymptotically optimal when processors are identical.

Finally, in section 6, we present experimental comparisons on a eight-processor machine between this algorithm and an optimal one with off-line static partitioning. Two cases are considered: dedicated processors and processors disturbed by additional processes. Even for small values of  $n$  (100) and of  $p$  (1 to 8), our adaptive algorithm has performances analogous to the optimal one when the machine is dedicated to the computation, and it is faster when the machine concurrently executes other processes (multiuser case), which is the practical case that motivates this work.

The coupling of two algorithms, a sequential one and a parallel one, is inspired by Daoudi *et al* [2] where it is applied to algorithms with the same number of operations on identical processors. However, its use for processors with different and possibly changing speeds, as well as the technique used to analyze its complexity are original. This technique is very general, we think that it can be applied to other problems.

## 2 Notations and on-line scheduling by work-stealing

Let  $W_\infty$  be the critical-path in number of operations for an execution on an unbounded number of processors;  $W$  is the total number of arithmetic operations

performed (work) by a given execution of the parallel algorithm. Note that  $W$  does not include scheduling operations but may depend both on the number of processors and the scheduling used for the considered execution. Let  $T_p$  be the execution time of the algorithm when scheduled on  $p$  physical processors, including scheduling overhead;

Cilk [5] and Kaapi [6,8] are parallel programming interfaces that support recursive parallelism and implement an on-line work-stealing scheduling based on the work first principle. The principle of work-stealing is simple. Each processor serially executes the tasks it has locally created according to a depth-first order. When a processor  $P_v$  becomes idle, it steals the oldest ready task (breadth first order) on a non-idle processor  $P_w$ , randomly chosen. For any series-parallel program with critical path  $W_\infty$  and work  $W$  on  $p$  identical processors, a work-stealing schedule ensures with high probability that  $T_p \leq \frac{W}{p} + O(W_\infty)$  [5,1].

Bender and Rabin [1] extended this theorem to heterogeneous processors of different and possibly changing speeds. The authors proposed a model that encompasses the practical use of a parallel architecture concurrently shared by various processes and users. Let  $\Pi_i(t)$  be the instantaneous speed of processor  $i$  at time  $t$ , measured as the number of operations  $\circ$  per unit of time. For a computation with duration  $T$ , let  $\Pi_{ave}$  be the average speed by processor:  $\Pi_{ave} = \frac{\sum_{t=1}^T \sum_{i=0}^{p-1} \Pi_i(t)}{p \cdot T}$ . In [1], a high utilisation schedule of factor  $\beta$  is used which is defined by the following property: if there are  $i < p$  idle processors, then the fastest idle processor is at most  $\beta$  times faster than the slowest busy processor. The parameter  $\beta$  can be tuned to optimize the performances of the system by reducing the number of migrations. Indeed, it is not even necessary to define a particular value of  $\beta$  [1] and in the sequel, we will assume  $\beta = O(1)$ .

To implement such a high utilisation schedule, the previous work-stealing is only modified in [1] when a processor  $P_v$  steals a work on an active processor  $P_w$  that has no ready work to be stolen in its local queue. Then, if  $P_w$  is slower than  $P_v$  by at least a  $\beta$  factor, then the work in progress on  $P_w$  is preempted and migrated on  $P_v$ . In the case when the processors speeds do not change too much, the following theorem bounds the execution time  $T_p$ .

**Theorem 1.** (see theorem 6 and 8 in [1]) *With high probability, the number of successful steal operations is  $O(p \cdot W_\infty)$  and the execution time  $T_p$  is bounded by*

$$T_p \leq \frac{W}{p \cdot \Pi_{ave}} + O\left(\frac{W_\infty}{\Pi_{ave}}\right).$$

The next section stands a lower bound for parallel prefix on this model.

### 3 Lower bound for parallel prefix on processors with varying speeds

In this paragraph, a lower bound is first given for  $p$  processors with varying speeds. Then, an off-line algorithm is provided that proves this lower bound is

tight on  $p$  processors with constant and known speeds  $\Pi_i$ . The next theorem stands the lower bound with respect to  $\Pi_{ave}$  and also to the maximal speed  $\Pi_{max}$  of all processors:  $\Pi_{max} = \max_{i=0, \dots, p-1; t=1, \dots, T} \Pi_i(t)$ .

**Theorem 2.** *A lower bound on the time  $T_p$  of any parallel prefix computation on  $p$  processors with average speed  $\Pi_{ave}$  and maximal speed  $\Pi_{max}$  is*

$$T_p \geq \frac{2n}{p \cdot \Pi_{ave} + \Pi_{max}}.$$

*Proof.* Let  $G$  be the computation DAG representing the execution of the parallel algorithm.  $G$  has  $n+1$  leaves corresponding to the inputs  $(x_i)_{i=0, \dots, n}$ ; each internal node matches an operation  $\circ$  with two inputs. Let  $A$  be the predecessor graph of a node that computes the output  $\pi_n$ . In  $\pi_n$ , each input  $x_i$  is operand of exactly one operation  $\circ$ ; then  $A$  is a binary tree with  $n+1$  leaves. Thus  $A$  contains exactly  $\#A = n$  operations  $\circ$ . Besides, let  $d$  be the depth of  $G$  and  $B$  be the complementary DAG of  $A$  in  $G$ . Since any prefix  $\pi_i$  is a successor of the leaf  $x_o$  and the depth of  $A$  is at most  $d$ , at most  $d$  prefixes are computed in  $A$ ; thus  $B$  computes at least  $n-d$  prefixes and then contains at least  $\#B = n-d$  operations  $\circ$ . As a consequence, the number  $\#G = \#A + \#B$  of nodes  $\circ$  in  $G$  is at least  $2n-d$  (note that this first part of the proof is similar to the one established in [4], theorem 2, for restricted prefix circuits of depth  $d = \log n$ ).

During  $T_p$ , at most  $p \cdot \Pi_{ave}$  operations  $\circ$  are computed; then,  $p \cdot \Pi_{ave} \cdot T_p \geq 2n-d$ . Besides, since  $G$  has a critical path with  $d$  operations  $\circ$ ,  $\Pi_{max} \cdot T_p \geq d$ . Putting things together gives  $(p \cdot \Pi_{ave} + \Pi_{max}) \cdot T_p \geq 2n$ .  $\square$

To prove that this lower bound is tight, we now introduce a parallel algorithm that reaches it in the restricted case where processors have uniform known speeds. For the sake of simplicity, the algorithm is first explicited in the case of  $p$  identical processors, and after extended to processors with uniform speeds. On  $p$  identical processors  $(P_i)_{i=0, \dots, p-1}$ , the algorithm is based on a partitioning of the  $n+1$  entries  $(x_i)_{i=0, \dots, n}$  in  $p+1$  blocks  $B_0, \dots, B_p$  of approximately the same size. To simplify, we suppose that each block  $B_i$  contains  $K = \frac{n}{p+1}$  consecutive elements.

**Step 1** In parallel for  $i = 0, \dots, p-1$ , we compute on processor  $i$  the sequential prefix of the block  $B_i$ . Let  $\alpha_i$  denote the last prefix of the block  $B_i$ . We notice that the prefixes  $(\pi_j)_{j=1, \dots, K}$  of the block  $B_0$  are thus computed.

**Step 2** We compute the  $p-1$  prefixes  $\beta_0 = \alpha_0$ ,  $\beta_1 = \alpha_0 \circ \alpha_1, \dots, \beta_{p-1} = \alpha_1 \circ \dots \circ \alpha_{p-1}$  of values  $\alpha_0, \dots, \alpha_{p-1}$ .

**Step 3** On processor 0, we compute the product by  $\beta_{p-1}$  of each element of the block  $B_p$  to obtain the prefixes  $\pi_{pK}, \dots, \pi_n$ . And, in parallel for  $i = 1, \dots, p-1$ , we compute on processor  $i$  the product by  $\beta_{i-1}$  of each element of the block  $B_i$ . We notice that these products are independant, even if they are made sequentially. All the prefixes  $\pi_i$  thus are obtained.

The execution time of this algorithm is  $2K + p - 1 \simeq \frac{2n}{p+1}$ , thus asymptotically optimal. Its number of operations  $2n - (2k + p - 1)$  is strictly optimal because it

reaches the lower bound  $2n - d$ . Moreover, we notice that by taking  $K = 2$  and by executing step 2 in a recursive way, it is the algorithm of Ladner and Fisher [7] which takes  $W = 2n$  operations  $\circ$  with a critical path  $W_\infty = 2 \log_2 N$ .

We now extend the previous algorithm to the case of  $p$  processors with uniform speeds  $\Pi_{max} = \Pi_0 \leq \Pi_1 \leq \dots \leq \Pi_{p-1}$  by tuning the block sizes in the partitioning. Let  $n_0 = \frac{n}{1+p \cdot \Pi_{ave} \cdot \Pi_{max}^{-1}}$ . Blocks  $B_0$  and  $B_p$ , each of size  $n_0$ , are assigned to processor 0. For  $1 \leq i \leq p-1$ , the processor  $i$  is assigned a block  $B_i$  of size  $n_0 \frac{\pi_i}{\Pi_{max}}$ . We also have  $2n_0 + \sum_{i=1}^{p-1} n_i = n$ . Step 1 and 3 takes a time  $\frac{n_0}{\Pi_{max}} = \frac{n_i}{\Pi_i} = \frac{n}{\Pi_{max} + p \cdot \Pi_{ave}}$ . So the whole time is  $\frac{2n}{\Pi_{max} + p \cdot \Pi_{ave}} + p$ , then asymptotically equal to the lower bound of theorem 2.

However, this algorithm assumes that relative speeds of the processors are known. It is not suited to the case of processors with varying speeds. In the next section, we present an on-line parallel algorithm that adapts automatically to the speeds of the processors by work-stealing.

## 4 Parallel adaptive algorithm

Our parallel algorithm with adaptive grain is based on the coupling of two algorithms: a sequential process  $P_s$  which sequentially computes prefixes and minimizes the number of operations and a variant of the preceding parallel algorithm, but with fine grain and scheduled by work-stealing on the  $p-1$  other processes. Initially, the process  $P_s$  starts the prefix computation of 1 to  $n$ . Let  $a = \frac{n}{p+1}$  and  $b = \frac{p}{p+1}n$ , the prefixes of 1 to  $a$  and  $b$  to  $n$  will be computed by this process  $P_s$ . However, the interval of indices  $[a, b]$  can be stolen and cut out recursively by processes  $P_v$  that become inactive. The algorithms for  $P_s$  and processes  $P_v$  are as follows:

*Sequential algorithm on process  $P_s$*

1.  $P_s$  sequentially computes the prefixes starting from index 1 (i.e.  $\pi_1$ ), until an index  $u_1$  such that the interval  $[u_1, u_2]$  of indices was stolen by a process  $P_v$ .
2.  $P_s$  preempts  $P_v$  and recovers the last index  $k \leq u_2$  computed by  $P_v$ , which thus already computed  $r_{u_1} = x_{u_1}, r_{u_1+1} = r_{u_1} \circ x_{u_1+1}, \dots, r_k = r_{k-1} \circ x_k$ .  $P_s$  sends the value  $\pi_{u_1-1}$  to  $P_v$  and starts again  $P_v$  (see below).
3.  $P_s$  computes  $\pi_k = \pi_{u_1-1} \circ r_k$ . Then it takes again the sequential computation of the prefix of  $k+1$  to  $n$  starting from  $k+1$  while returning at step 1. We speak about jump operation. For each jump operation,  $P_s$  makes an operation  $\circ$ .
4.  $P_s$  stops when it computed  $\pi_n$  (the prefixes of indices of  $b$  to  $n$  cannot be stolen). After having computed  $\pi_n$ , it becomes a thief process and executes the algorithm  $P_v$ .

*Parallel algorithm on  $p-1$  processes  $P_v$*

- When it is preempted by  $P_s$  (see algorithm of  $P_s$ ),  $P_v$  already computed partial prefix locally  $r_{u_1}, \dots, r_{u_k}$  of interval  $[u_1, u_k]$ . It then receives the value of the last prefix  $\beta = \pi_{u_1-1}$  computed by  $P_s$ . It then finalizes the interval  $[u_1, u_k]$  by computing the products  $\pi_i = \beta \circ r_i$  for  $u_1 \leq i \leq u_k$ . These products are parallel. On inactivity of another process thief, a half of these computations remaining to be made on  $P_v$  in this interval can then be stolen.
- When it is inactive, process  $P_v$  chooses a processor until finding an active process  $P_w$ . It can be either  $P_s$  or another thief process. If the victim is  $P_s$ , the steal is possible only if  $P_s$  has a remaining interval of indices ranging between  $a$  and  $b$ .
  1.  $P_v$  cuts the stealable interval on  $P_w$  in two parts.  $P_v$  extracts the right part  $[u_1, u_2]$  of the interval and steals it. The left part remains on  $P_w$ .
  2.  $P_v$  starts computation on the stolen interval  $[u_1, u_2]$ . It can be either a computation of a local prefix (i.e.  $r_{u_1} = x_{u_1}, r_{u_1+1} = r_{u_1} \circ x_{u_1+1}, \dots$ ) or the finalization of computations of prefixes starting from already computed values  $r_k$  (i.e.  $\pi_{u_1+1} = \pi_{u_1} \circ r_{u_1+1}, \pi_{u_1+2} = \pi_{u_1} \circ r_{u_1+2}, \dots$ ).

The program stops when all the processors are inactive. The main point of this algorithm is that a process that become slow will be preempted by the sequential process or will be stolen by a parallel process. The following section analyzes the complexity of this adaptive algorithm.

## 5 Asymptotic optimality of the adaptive algorithm

We use the modified work-stealing schedule (theorem 1) to execute the adaptive algorithm for the computation of parallel prefixes on  $p$  processors of changing speed. As in [1], we assume that the speed of the processor vary within a constant factor: there is a constant  $c \geq 1$  such that  $\max_{i,t} II_i(t) \leq c \cdot \min_{i,t} II_i(t)$ .

**Theorem 3.** *With high probability,*

$$T_p \leq \frac{2n}{(p+1)II_{ave}} + O\left(\frac{\log n}{II_{ave}}\right) \sim_{n \rightarrow \infty} \frac{2n}{(p+1)II_{ave}}.$$

*Proof.* For the analysis, we cut out the execution in two successive phases,  $\phi_1$  and  $\phi_2$ . The phase  $\phi_1$  is until  $P_s$  has computed  $\pi_n$ . Then, the phase  $\phi_2$  starts when  $P_s$  becomes a work-stealer. Let  $n_{seq}$  (resp.  $j$ ) be the number of prefixes (resp. jumps) computed by  $P_s$  during  $\phi_1$ . Let  $x$  (resp.  $y$ ) be the number of final prefix computed by the other processes (work-stealers) in  $\phi_1$  (resp.  $\phi_2$ ). Then  $n = n_{seq} + j + x + y$  and  $W = n_{seq} + 2j + 2x + 2y$ . Let  $I_1$  (resp.  $I_2$ ) be the total number of operations performed by idle processors during  $\phi_1$  (resp.  $\phi_2$ ).

1. During the phase  $\phi_1$  of time  $T_p(\phi_1)$ , the sequential prefix algorithm is always executing on a processor and makes  $n_{seq} + j$  operations  $\circ$ . We note  $II_{seq}$  the average speed of this algorithm:  $II_{seq} = \frac{n_{seq} + j}{T_p(\phi_1)}$ .

At each unit of time, the  $p - 1$  others processors make the parallel part of the adaptative algorithm and make in total  $2x + y + j$  operations  $\circ$  and  $I_1$  inactivity operations. During  $\phi_1$ , the average speed per processor for this part of the algorithm is  $\Pi_{ave}(\phi_1) = \frac{2x+y+j+I_1}{(p-1) \cdot T_p(\phi_1)}$ .

- The total number of operations  $\circ$  in the phase  $\phi_1$  is  $W(\phi_1) = n_{seq} + 2j + 2x + y$ .
2. During the phase  $\phi_2$ , the sequential part of the adaptative algorithm is finished. The  $p$  processors finalize the  $y$  prefix computations that were anticipated in parallel and not finished in the phase  $\phi_1$ . During the phase  $\phi_2$ , the  $p$  processors make thus  $y$  operations  $\circ$  and  $I_2$  inactivity operations. The average speed per processor during  $\phi_2$  is  $\Pi_{ave}(\phi_2) = \frac{y+I_2}{p \cdot T_p(\phi_2)}$ .

For the sake of simplicity, we assume  $\Pi_{seq} = \Pi_{ave}(\phi_1) = \Pi_{ave}(\phi_2) = \Pi_{ave}$  (without loss of generality, since they are within a constant factor  $c$ ). During  $\phi_2$ , the processors which don't execute the sequential algorithm (i.e.  $p-1$  at each unit of time) make  $2x + y + J$  operations  $\circ$  with a critical path  $W_\infty(\phi_1) \leq 2 \cdot \log_2 n$  related to recursive cutting. By applying the theorem 1, we obtain  $T_p(\phi_1) \leq \frac{2x+y+j}{(p-1)\Pi_{ave}} + O\left(\frac{\log n}{\Pi_{ave}}\right)$ . Thus,  $T_p(\phi_1) = \frac{n_{seq}+j}{\Pi_{seq}} = \frac{n_{seq}+j}{\Pi_{ave}}$ . And then,  $(p+1)T_p(\phi_1) = (p-1)T_p(\phi_1) + 2T_p(\phi_1) \leq \frac{2n_{seq}+2x+y+3j}{\Pi_{ave}} + O\left((p-1)\frac{\log n}{\Pi_{ave}}\right)$ . As  $n = n_{seq} + x + y + j$ , We obtain:  $(p+1)\Pi_{ave}T_p(\phi_1) \leq 2n - y + j + O((p-1)\log n)$ . In addition, by applying theorem 1 to  $\phi_2$ , we obtain  $T_p(\phi_2) \leq \frac{y}{p\Pi_{ave}} + O\left(\frac{\log n}{\Pi_{ave}}\right)$ . Thus,  $(p+1)\Pi_{ave}T_p = (p+1)\Pi_{ave}T_p(\phi_1) + (p+1)\Pi_{ave}T_p(\phi_2) \leq 2n + j + \frac{y}{p} + O(p \log n)$ .

The number  $j$  of jumps is lower than the number of successful stealing i.e.  $O(\log n)$  since  $W_\infty(\phi_1) \leq 2 \log n$  (theorem 1). Moreover, by using  $(p-1)(n_{seq} + j) = (2x + y + j + I_1)$  and  $n_{seq} \geq \frac{2n}{p+1}$ , we obtain  $y \leq I_1 \leq (p-2) \log n$ . Finally, we have:  $(p+1)\Pi_{ave}T_p \leq 2n + O(p \log n)$ .  $\square$

We can note that the proof and the theorem remain valid in the more general and realistic case where  $\Pi_{seq} \geq \Pi_{ave}(\phi_1)$  (the sequential algorithm is always executed by a processor faster than the average of the processors) and  $\Pi_{ave}(\phi_2) \geq \Pi_{ave}(\phi_1)$  (the sequential processor added in phase 2 is faster than the average of the other processors).

## 6 Experimental Results

We implemented the algorithms on a eight-processor SMP machine, with 31 GB of memory (Intel's Itanium-2 at 1.5 GHz) and in the multi-user context under the GNU/Linux 2.6.7 system. The adaptive and parallel algorithms are implemented with Kaapi [6,8].

The experiments consist in the computation of prefixes of 10000 elements (double) with a time of 1ms per operation  $\circ$  while varying  $p$  the number of processors from 1 to 8. The optimal sequential time of reference is 10s.

Tables 1 and 2 give the execution times obtained by the two parallel algorithms (with fixed grain on  $p$  processors and adaptive grain). For each experiment, we made 10 measurements and we kept the times of the fastest and the slowest execution and the average time of the 10 executions.

	Sequential	Static				Adaptive			
		p=2	p=4	p=6	p=8	p=2	p=4	p=6	p=8
Lower bound $\frac{2n}{(p+1)H_{ave}}$	10.00	6.67	4.00	2.86	2.22	6.67	4.00	2.86	2.22
Min	10.087	6.73	4.04	2.89	2.82	6.73	4.03	2.88	2.23
Avg	10.09	6.74	4.05	2.93	2.87	6.73	4.04	2.89	2.24
Max	10.09	6.75	4.06	3.00	2.99	6.73	4.04	2.89	2.25

**Table 1.** Comparison of the times of the three algorithms on  $p$  identical processors.

Table 1 compares the execution times when there are no other computations in progress on the processors. We notice that measurements of time are stable (variation between minimum and maximum lower than 6% for the algorithm with fixed grain and lower than 8% for the algorithm than adaptive grain). We check the optimality of the algorithm with fixed grain whose time is with less than 8% of the lower bound. Moreover, we check the optimality of the adaptive algorithm which is also less than 5% of the lower bound.

	Static				Adaptive			
	p=2	p=4	p=6	p=8	p=2	p=4	p=6	p=8
Lower bound $\frac{2n}{(p+1)H_{ave}}$	7.49	4.50	3.22	2.50	7.49	4.50	3.22	2.50
Min	8.34	7.33	4.97	3.67	7.55	6.03	3.77	2.94
Avg	9.97	8.15	5.60	4.05	9.21	7.23	4.47	3.34
Max	10.41	8.57	5.77	4.31	10.28	8.12	5.23	3.86

**Table 2.** Comparison of the times of the algorithms on  $p$  perturbed processors. Each column reports, the minimal, average and maximal times of 10 executions. For each of those 10 executions, the adaptive algorithm is the fastest

In table 2, additional processes of load are injected to disturb the load of the machine and to simulate the behavior of a real machine, disturbed by other users. In the aim of reproducibility, each experiment on  $p \leq 8$  processors is disturbed by  $9 - p$  artificial processes of duration larger than 10s. We can check in table 2 that the adaptive algorithm is at least 7% faster.

We note that the time are very changing but we observe that in the case of the minimum time, the adaptive algorithm is not so far from the lower theoretical bound ( $H_{ave} = \frac{8}{9}$ ). We think this is due to the scheduling of the system.

In conclusion, the adaptive algorithm brings a guaranteed performance when the machine is divided between several users, while adapting automatically to the available resources during the execution. Moreover its performance remains close to optimal even in the ideal case where the processors are all dedicated to the application. It thus appears to be more powerful than the sequential algorithm or than a fixed parallel algorithm.



This is confirmed by another experimentation where each elementary test corresponds to simultaneous launching in competition of the nine programs: adaptive algorithm on eight processors, sequential algorithm and the fixed parallel algorithm for the seven values  $p = 2, \dots, 8$  processors. Table 3 summarizes the results on a 10 test campaign. For 10 executions, the adaptive algorithm is always the fastest.

Its average time of execution is on average 19% times shorter than that of the optimal fixed parallel algorithm on 8 processors, with variations to 40% on one of the tests.

	Sequential	Static					Adaptive p=8
		p=2	p=4	p=6	p=7	p=8	
Min	20.53	18.70	16.41	13.93	13.54	12.25	10.79
Max	22.96	20.04	17.23	15.86	14.56	13.66	13.06
Avg	21.69	19.24	16.89	15.13	13.89	13.16	12.09
Median	22.00	19.26	16.96	15.12	13.76	13.12	12.18

**Table 3.** Comparison of the times of the 9 algorithms simultaneously launched – On the 10 executions of each test, the adaptive algorithm was the fastest

## 7 Conclusion

Motivated by the use of multi-processor machines shared between several users, we introduced a new parallel algorithm for the prefix computation which adapts automatically and dynamically to the available processors. This algorithm performs an asymptotically optimal number of operations. It is equivalent to that of the sequential algorithm when only one processor is available and to that of an optimal parallel algorithm when  $p$  identical processors are available. In the case of  $p$  variable processors speeds, its time is equivalent to that of an optimal algorithm on  $p$  identical processors speed equal to the average speeds. These theoretical results are validated by the experiments made on a SMP machine with 8 processors. A first perspective is to validate it on the national French heterogeneous grid GRID'5000 within the ANR BGPR-Safescale project.

More generally, our adaptive algorithm is based on the recursive and dynamic coupling of two algorithms, a sequential one, optimal in terms of number of operations, and a parallel one with a maximum degree of parallelism. Both the algorithm and its analysis are applied to the prefix computation, for which any parallel algorithm requires more operations than the sequential algorithm. However, we think that both this scheme and its theoretical analysis are more general and may apply to other problems, in particular for the resolution of exact linear systems.

## References

1. Bender, M.A., Rabin, M.O.: Online scheduling of parallel programs on heterogeneous systems with applications to cilk. *Theory Comput. Syst.* **35**(3) (2002) 289–304
2. Daoudi, E.M., Gautier, T., Kerfali, A., Revire, R., Roch, J.L.: Algorithmes parallèles à grain adaptatif et applications. *TSI* **24** (2005) 1–20
3. Dimitrakopoulos, G., Nikolos, D.: High-speed parallel-prefix vlsi ling adders. *IEEE Trans. Computers* **54**(2) (2005) 225–231
4. Fich, F.E.: New bounds for parallel prefix circuits. In: *STOC '83: Proceedings of the 15th ACM symp. Theory of computing*, New York, NY, USA, ACM Press (1983) 100–109
5. Frigo, M., Leiserson, C.E., Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*. (1998)
6. Jafar, S., Gautier, T., Krings, A.W., Roch, J.L.: A checkpoint/recovery model for heterogeneous dataflow computations using work-stealing. In Springer-Verlag, L., ed.: *EUROPAR'2005*, Lisboa, Portugal (2005)
7. Ladner, R., Fischer, M.: Parallel prefix computation. *J. ACM* **27**(4) (1980) 831–838
8. MOAIS Project: KAAPI homepage. <http://gforge.inria.fr/projects/kaapi/> (since 2005)
9. Wang, H., Nicolau, A., Siu, K.Y.S.: The strict time lower bound and optimal schedules for parallel prefix with resource constraints. *IEEE Trans. Comput.* **45**(11) (1996) 1257–1271