Processor-oblivious parallel stream computations

Julien Bernard Jean-Louis Roch Daouda Traore
{Julien.Bernard,Jean-Louis.Roch,Daouda.Traore}@imag.fr
Laboratoire d'Informatique de Grenoble
Équipe MOAIS (CNRS-INRIA-INPG-UJF)
38330 Montbonnot Saint Martin, France
http://moais.imag.fr/

Abstract

We study the problem of parallel stream computations on a multiprocessor architecture. Modelling the problem, we exhibit that any parallelisation introduces an arithmetic overhead related to intermediate copy operations. We provide lower bounds for the parallel stream computation on p processors of different speeds with two models, a strict model and a buffered model; to our knowledge, these are new results. We introduce a new parallel algorithm called processor-oblivious: it is based on the coupling of a fast sequential algorithm with a fine-grain parallel one that is scheduled by work-stealing. This algorithm is proved asymptotically optimal. We show that our algorithm has a good experimental behaviour.

1. Introduction

The amount and the variety of data communications has increased with the network capacities. Many computations must now be done on-the-fly in real time: compress, cipher, filter, etc. We refer to these computations as stream computations. While the network speed has long been the bottleneck, the stream computations may now slow down the communication: for example, a standard zlib compression can achieve a speed of 10 to 20 Mbps (depending on the machine) while an ADSL2+ can reach 24 Mbps. To benefit from the network improvements, a way to accelerate stream computations is to use the parallel multicore architectures.

A stream computation can naturally be described as a sequential process that reads blocks of data from the input device, computes the result and writes it to the output device. In the general case, the size of the output may differ from the size of the input and may not be known until the result has effectively been computed. This implies that if the computations are done in parallel, the result of block i can't be written to the output until block $i-1$ has been computed. So taking benefit of parallelism requires additional memory copies via temporary buffers. Those copies result in an arithmetic overhead proportional to the size of the memory space used in the buffers.

In section 2, we study this overhead and provide a lower bound for the time T_p of parallel computation on p identical processors of average speed Π in two situations: direct stream computation and buffered stream computation. To our knowledge, these bounds are new results.

Based on a work-stealing schedule [1, 2], we exhibit in section 3 a generic parallel stream computation algorithm, asymptotically achieving the lower bound. This algorithm generalises the recursive three-loop scheme introduced in [4] to stream computations. Only based on processor idleness, its main feature is to automatically adapt the number and the size of the blocks without any consideration neither on the number and speed of processors nor on the input data to process. This is why it is named as *processor-oblivious* [3].

2. Lower bounds for parallel stream computation

We consider a machine with p processors of average speed Π with an input device and an output device.

We denote R_I the bit rate of the input device, R_O the bit rate of the output device and R_C the computation bit rate. If $R_C > R_I$ or $R_C > R_O$, then the bottleneck is one of the I/O device and there is no need to improve the speed of the computation. So we suppose that $R_C < R_I$ and $R_C < R_O$.

We even suppose $R_C \ll R_I$ so that all the input bytes are available when desired (they can be buffered if necessary at no cost). In fact, this hypothesis is not too strong with our adaptive algorithm in section 3.2.

Then, we suppose we have two operations:

- The normal computation C which takes an element

from the input stream and computes it either in the output stream or in an intermediate result. We call ω_C the work of this operation.

- The overhead computation O which takes an intermediate result, computes the additional operation due to the parallelisation and puts it in the output stream. We call ω_O the work of this operation.

2.1. Direct stream computation and lower bound

In the model of *direct stream computation*, we consider that the output stream has a strictly sequential semantic: the stream must be written sequentially and only one process can write in the stream at each step.

Theorem 1 *The parallel time on p processors of a parallel direct stream computation is lower bounded by:*

$$T_p \geq \frac{n}{p.\Pi} \frac{\omega_C}{1-(1-\frac{1}{p})\frac{\omega_O}{\omega_C}}$$

Proof: Let a be the total number of input elements that have been directly computed in the output stream. So the number of overhead operations is $(n-a)$. On the one hand, if we consider the total work on the p processors, we have $p.\Pi.T_p \geq n.\omega_C + (n-a).\omega_O$ (1). On the other hand, if we take into account that the output stream must be written sequentially, we have $\Pi.T_p \geq a.\omega_C + (n-a).\omega_O$ (2). Using $(\omega_C - \omega_O).(1) + \omega_O.(2)$, we finally have: $(p.(\omega_C - \omega_O) + \omega_O).\Pi.T_p \geq n.{\omega_C}^2$

□

Let consider the two following border cases:

1. If $\omega_C \gg \omega_O$, then the lower bound is $T_p \geq \frac{n.\omega_C}{p.\Pi} = \frac{T_1}{p}$, which is the trivial bound.
2. If $\omega_C \sim \omega_O$, then the lower bound is $T_p \geq \frac{n.\omega_C}{\Pi} = T_1$, i.e. parallelism does not bring any speedup. This is due to the restrictive nature of the above stream model: the sequential semantic implies that writes are on the critical path.

The next section consider a more general stream model that improves this second border case.

2.2. Buffered stream computation and lower bound

In the model of *buffered stream computation*, the output stream can be written at multiple places in the same time. If it's not possible in the reality, we can imagine an intermediate buffer (hence the name) where the data is stored and a special process that send the data sequentially in the output stream as soon as it is available. The buffer is not really a limitation as, in the reality, such systems already exist: buffers for packet-based network, write buffer for hard disks.

Theorem 2 *The parallel time on p processors of a parallel buffered stream computation is lower bounded by:*

$$T_p \geq \frac{n}{\Pi} \frac{\omega_C + \omega_O}{p + \frac{\omega_O}{\omega_C}}.$$

Proof: Let a be the total number of input elements that have been directly computed in the output stream with no overhead. On the one hand, those a input elements must have been sequentially processed (maybe by different processors, but in sequence): then $T_p.\Pi \geq a.\omega_C$ (1). On the other hand, the number of overhead operations is $(n-a)$. So, if we consider the total work on the p processors, we have: $p.T_p.\Pi \geq n.\omega_C + (n-a).\omega_O$ (2). Using $(2) + \frac{\omega_O}{\omega_C}.(1)$, we obtain $\left(p + \frac{\omega_O}{\omega_C}\right).T_p.\Pi \geq n.\omega_C + n.\omega_O$ which states the lower bound on T_p.

□

As an illustration, let us consider the two previous border cases:

1. If $\omega_C \gg \omega_O$, then the lower bound is $T_p \geq \frac{n.\omega_C}{p.\Pi} = \frac{T_1}{p}$, which is the trivial bound again.
2. If $\omega_C \sim \omega_O \sim \omega$, the lower bound is $T_p \geq \frac{n.(\omega_C+\omega_O)}{(p+1).\Pi} \simeq \frac{2.n.\omega}{(p+1).\Pi}$: this lower bound is indeed the same as the strict one for prefix computation [8, 9]. This is not surprising since prefix may be considered as a special case of stream computation.

In both cases, previous lower bounds exhibit potential speed-up for any $p \geq 2$.

3. Optimal on-line parallel stream computation

In this section, we first introduce the generic parallelisation of a stream computation based on work-stealing. Then we prove it achieves the previous lower bound.

3.1. Work-stealing and processor-oblivious algorithms

We still consider a machine with p processors $P_1, \ldots, P_p$ of average speed Π i.e. number of instructions per second per processor. Following Bender and Rabin [2], we call

work W of a computation the number of unit processor instructions it performs. So the time T to compute that operation on a single processor is $T = \frac{W}{\Pi}$.

We use the work-stealing scheduling paradigm [1] in which each processor executes its own task until it becomes idle and then, steals a fraction of the remaining work on a randomly chosen busy processor. This paradigm has been studied and implemented in Cilk [5, 2] and KAAPI [6, 7]. The following theorem stands an upper bound on the parallel time used for the execution of any series-parallel program with total work W and depth D (critical path in number of unit instructions) on p heterogeneous processors with average speed Π per processor:

Theorem 3 (theorem 6 and 8 in [2]) *With high probability, the number of steal attempts is $O(p.D)$ and the execution time T_p on p processors is bounded by:*

$$T_p \leq \frac{W}{p.\Pi} + O\left(\frac{D}{\Pi}\right).$$

This theorem ensures that the number of idle tops is $O(pD)$ which appears optimal when the arithmetic depth D is very small compared to W. However, this bound is only valid when the work W is constant whatever the number of processors is. This is not the case for stream computations in general: due to additional arithmetic overhead, performing parallel work increases the number of instructions.

3.2. Our processor-oblivious algorithm

Our processor oblivious parallel algorithm relies on the on-line coupling of two algorithms: a sequential algorithm is performed by a single process $\mathcal{S}$ and a parallel algorithm is performed by an unknown number of thief processes $\mathcal{T}$. Each process $\mathcal{T}$ acts as a co-processor to speed-up the sequential process $\mathcal{S}$. If there is only one processor available, this reduces to the sequential stream computation. The synchronisation between those processes is based on work-stealing.

Efficient scheduling by work-stealing When non-idle, a process is always computing a block $[a, b[$ of contiguous elements, storing its results in the final output buffered stream for $\mathcal{S}$ or in an intermediate local buffer for $\mathcal{T}$.

When a process $\mathcal{T}$ become idle after completion of its own block, it becomes a thief: it randomly chooses a victim processor and steals the last half part $\left[\frac{a+b}{2}, b\right[$ of the remaining block on the victim and starts computing it in a new temporary buffer.

The important point is that $\mathcal{S}$ strictly follows the sequential order in the stream: $\mathcal{S}$ always writes the elements in the output buffered stream at their final position. When $\mathcal{S}$ encounters a block $[a, b[$ that has been stolen and computed by a theft $\mathcal{T}$ in a temporary buffer $[u, v[$ of length L, it preempts the process $\mathcal{T}$. Then it performs a *jump operation* as follows. Let i be the current position $\mathcal{S}$ points to in the output stream, the temporary buffer $[u, v[$ can asynchronously be finalised (due to parallelisation) and copied in the output stream at position i. Meanwhile, the process $\mathcal{S}$ directly jumps to compute the block starting at $b + 1$, and writting directly its results in the output stream at position $i + L$.

Note that $\mathcal{S}$ never waits: except for preemptions (each performed in $O(1)$ time), all its operations are performed by the full sequential stream computation. When $\mathcal{S}$ reaches the end of the input stream, all computations are necessarily completed but there may be still to complete some finalisations and copies of intermediate buffers in the output stream. Then, the sequential process participates behaving as a theft like the other processes.

If there are n elements to compute, the depth of the computation is $D = \Theta(\log n)$ due to recursive splitting on successfull steals. Then, from theorem 3, this on-line assignment of elements ensures a well balanced load on p processors, while the number of synchronisations, $O(p \log n)$ steal requests, compares favourably to the work $W = \Omega(n)$ for n large enough.

However, work stealing introduces arithmetic and mutual exclusion overheads. In order to bound the overhead of parallelism on the stream computation, the algorithm is structured in three nested loops which are now detailed.

Bounding mutual exclusion overhead: microcroloop and nanoloop Two processes need to modify the bounds of the block $[a, b[$ assigned to a process. On the one hand, the process itself has to take a portion of the block to compute it sequentially: this is the `extract_seq()` operation. On the other hand, its thieves, when stealing, has to take a fraction of the block: this is the `extract_par()` operation. This two operations must be done in mutual exclusion. Each process execute two nested loops: the microloop and the nanoloop.

When a process has an empty interval, it performs `extract_par()` operations until finding a non idle victim processor. The `extract_par()` operation splits the victim block in two halves and steals the one that would have been performed last by the victim. It then enters the nanoloop.

To compute its local interval, the process performs `extract_seq()` to obtain a small subset of elements that it proceeds *non preemptively*, in sequence. The `extract_seq()` operation consists in extracting the first $\Theta(\log(b - a))$ elements of the block.

Due to the granularity choice $\Theta(\log(b - a))$, the depth D remains $\Theta(\log n)$ but the whole number of `extract_seq()` operations is now bounded by $\Theta\left(\frac{n}{\log n}\right)$. So the overhead of mutual exclusion becomes

$\Theta\left(\frac{n}{\log n}\right)$; it is hidden by the work $W = \Omega(n)$.

Bounding buffer copies overhead: amortised macroloop In order to bound the arithmetic overhead induced by copies, an amortised scheme is used. The global range $[0, n($ of the input stream is broken into several macrosteps of increasing size. So the global computation is structured in a sequential macroloop: the step k completes and step $k+1$ starts only when all computations and buffer copies related to step $k-1$ are completed. As a result, a theft process cannot steal a range in macrostep $k+1$ if all elements before the corresponding range has not been written in the output stream. Note that only thief processes $\mathcal{T}$ are synchronised by the macroloop. The process $\mathcal{S}$ do not participate to copies, except during the last macrostep.

Let s_k be the size of the k-th macrostep ($s_1 = O(1)$ is predefined) and let $n_k = \sum_{i=1}^{k} s_i$. We define $s_{k+1} = \frac{n_k}{\varepsilon(n_k)}$ with $\varepsilon(n) = o(n)$ and $\varepsilon(n) \to +\infty$ (typically, we will consider $\varepsilon(n) = \log n$ in the sequel) As a consequence, due to the sequential macroloop, the resulting depth of the computation becomes $D(n) = \varepsilon(n).\log n = \log^2 n$.

The next theorem states that this algorithm, independent from the number of processors and their relative speeds, reaches asymptotically the lower bound of theorem 2.

3.3. Asymptotic optimality of the processor-oblivious algorithm

Theorem 4 *The time T_p on p processors of the processor-oblivious stream compression algorithm verifies:*

$$T_p \leq \frac{n}{\Pi} \frac{\omega_C + \omega_O}{p + \frac{\omega_O}{\omega_C}} + O\left(\frac{n}{\varepsilon(n)}\right)$$

which is asymptotically ($n \to \infty$) optimal.

Proof: We cut the computation in two phases, ϕ_1 until the sequential process $\mathcal{S}$ computes the last byte of the last macrostep and then ϕ_2. At the end of ϕ_1, all the input buffer has been computed, either in the output buffer or in temporary buffers. By definition of the macrostep, the size of the last macrostep is $O\left(\frac{n}{\varepsilon(n)}\right) = o(n)$.

We call a the number of bytes that have been processed by $\mathcal{S}$ and b the number of bytes that have been copied from temporary buffers to the output buffer in phase ϕ_1. We also call j the number of jump operations (which have a work of ω_J) and i_k ($k \in [1,2]$) the number of idle tops of the parallel processes in phase ϕ_k (which have a work of ω_I).

During phase ϕ_1 of time $T_p(\phi_1)$: if we consider the sequential process which always runs, $\Pi.T_p(\phi_1) = a.\omega_C + j.\omega_J$; if we consider the parallel process, $(p-1).\Pi.T_p(\phi_1) = (n-a).\omega_C + b.\omega_O + i_1.\omega_I$. The $p-1$ parallel processes make a work with critical path $D(\phi_1) = O(\log n)$ due to recursive stealing. By applying theorem 3, $j = O((p-1).\log n)$ and $i_1 = O((p-1).\log n)$. Necessarily, we have $b \leq n - a$. So, from all this, $a \leq n.\frac{\omega_C+\omega_O}{p.\omega_C+\omega_O} + O(\log n)$ and then: $T_p(\phi_1) \leq \frac{n}{\Pi}\frac{\omega_C+\omega_O}{p+\frac{\omega_O}{\omega_C}} + O(\log n)$

During phase ϕ_2 of time $T_p(\phi_2)$, as said before, there are $n-a-b = O\left(\frac{n}{\varepsilon(n)}\right)$ bytes in the temporary buffers to copy, so $p.\Pi.T_p(\phi_2) = (n-a-b).\omega_O + i_2.\omega_I$. By applying theorem 3, $i_2 = O(\log(n-a-b)) = O\left(\log\left(\frac{n}{\varepsilon(n)}\right)\right) = O(\log n)$. So: $T_p(\phi_2) \leq \frac{1}{p.\Pi}.O\left(\frac{n}{\varepsilon(n)}\right).\omega_O + O(\log n)$

So joining both results leads to: $T_p \leq \frac{n}{\Pi}\frac{\omega_D+\omega_C}{p+\frac{\omega_C}{\omega_D}} + O\left(\frac{n}{\varepsilon(n)}\right)$

□

4. Experiments

To experiment the presented processor-oblivious algorithm, we implemented a filter function and made some experiments on a NUMA Bull machine with 8 Itanium 1.5GHz processors running GNU/Linux 2.6.12. We compared our implementation (`adaptive`) with a trivial parallel one (`parallel`) that equally share the input data between all the processors.

The overhead operation is just a copy and we measured $\omega_O = 7.8$ on this machine and the main computation is composed of several multiplications ($\omega_C = 207$) on elements that are filtered. The sequential time is about 700ms.

Figure 1 shows that `adaptive` is quite close to the optimal speedup. It also shows the performance of `parallel` decrease with more than 4 processors, this is due to the NUMA architecture. `adaptive` has a much better behaviour regarding this architecture.

Figure 2, that is the same experiment with very heterogeneous data (more elements are filtered at the end of the buffer), shows that the behaviour of `adaptive` is the same whatever the input whereas the behaviour of `parallel` depends on the input. This emphasizes the name *processor-oblivious* of our algorithm, its behaviour does not depend on the input data.

5. Conclusion

Modelling stream computation as an algorithm that introduces memory management overhead (copy via intermediate buffers), we exhibit a lower bound $T_p \geq \frac{n}{\Pi}\frac{\omega_C+\omega_O}{p+\frac{\omega_O}{\omega_C}}$ where ω_C (resp. ω_O) is the elementary time to compute (resp. compute the overhead operations of) one byte of input stream. Then, using more processes than the number

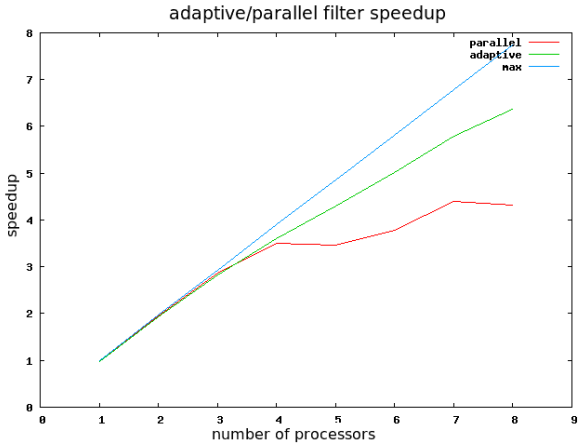

Figure 1. speedup of a filter on homogeneous data

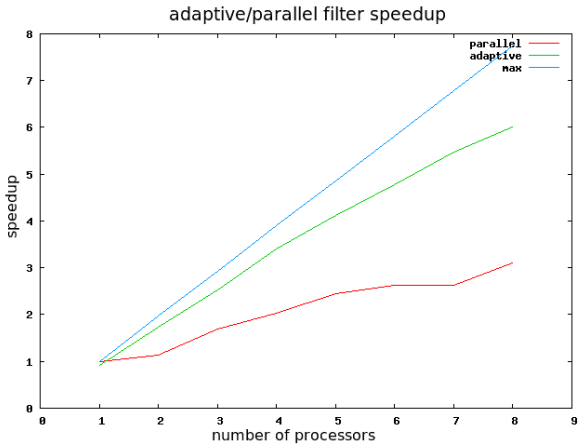

Figure 2. speedup of a filter on heterogeneous data

of processors introduces arithmetic overhead and decreases performances: parallel slackness is unsuited to this problem. This lower bound appears as a generalisation of the lower bound of parallel prefix computation [9]: in the case of prefix, $\omega_C = \omega_O$, both corresponding to a same arithmetic operation. For prefix, the bound has been proven tight [10]; for parallel stream computation, we only proved asymptotic optimality but we think that a similar algorithm to the one of Nicolau et al. could establish the strict optimality of the bound.

Based on an underlying work-stealing scheduling, we exhibit a processor-oblivious compression algorithm that asymptotically achieves this lower bound. Its main property is to execute a sequential compression on a given processor with no overhead: this processor is active during almost all the computation while the other ones always behaves as work-stealers. The algorithm behaves well whatever the charge of the processors or the complexity of input data.

References

[1] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2):115–144, 2001.

[2] M. A. Bender and M. O. Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to Cilk. *Theory of Computing Systems Special Issue on SPAA00*, 35:289–304, 2002.

[3] V. D. C. Cung, V. Danjean, J.-G. Dumas, T. Gautier, G. Huard, B. Raffin, C. Rapine, J.-L. Roch, and D. Trystram. Adaptive and hybrid algorithms: classification and illustration on triangular system solving. In J. Dumas, editor, *Transgressive Computing TC'2006*, pages 131–148, Granada, Spain, April 2006.

[4] V. Danjean, R. Gillard, S. Guelton, J.-L. Roch, and T. Roche. Adaptive loops with kaapi on multicore and grid: Applications in symmetric cryptography. In A. publishing, editor, *Parallel Symbolic Computation'07 (PASCO'07)*, London, Ontario, Canada, July 2007.

[5] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM Press.

[6] S. Jafar, T. Gautier, A. W. Krings, and J.-L. Roch. A checkpoint/recovery model for heterogeneous dataflow computations using work-stealing. In L. Springer-Verlag, editor, *EUROPAR'2005*, August 2005.

[7] MOAIS Project. KAAPI. `http://gforge.inria.fr/projects/kaapi/`, 2007.

[8] A. Nicolau and H. Wang. Optimal schedules for parallel prefix computation with bounded resources. In *PPOPP '91: Proceedings of the third ACM SIGPLAN symp. Principles and practice of parallel programming*, pages 1–10, New York, NY, USA, 1991. ACM Press.

[9] J.-L. Roch, D. Traore, and J. Bernard. On-line adaptive parallel prefix computation. In *Euro-Par 2006 Parallel Processing*, volume 4128 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[10] H. Wang, A. Nicolau, and K.-Y. S. Siu. The strict time lower bound and optimal schedules for parallel prefix with resource constraints. *IEEE Transactions on Computers*, 45(11):1257–1271, 1996.