



Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

***Overlapping Computations and
Communications with I/O in Wavefront
Algorithms***

Eddy Caron,
Frédéric Desprez,
Frédéric Suter

December 2004

Research Report N° 2004-58

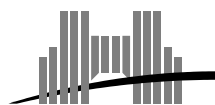
École Normale Supérieure de Lyon

46 Allée d'Italie, 69364 Lyon Cedex 07, France

Téléphone : +33(0)4.72.72.80.37

Télécopieur : +33(0)4.72.72.80.80

Adresse électronique : lip@ens-lyon.fr



INRIA



Overlapping Computations and Communications with I/O in Wavefront Algorithms

Eddy Caron, Frédéric Desprez, Frédéric Suter

December 2004

Abstract

Several numerical computation algorithms exhibit dependences that lead to a wavefront in the computation. Depending on the data distribution chosen, pipelining communication and computation can be the only way to avoid a sequential execution of the parallel code. The computation grain has to be wisely chosen to obtain at the same time a maximum parallelism and a small communication overhead. On the other hand, when the size of data exceeds the memory capacity of the target platform, data have to be stored on disk. The concept of *out-of-core* computation aims at minimizing the impact of the I/O needed to compute on such data. It has been applied successfully on several linear algebra applications. In this paper we apply *out-of-core* techniques to wavefront algorithms. The originality of our approach is to overlap computation, communication, and I/O. An original strategy is proposed using several memory blocks accessed in a cyclic manner. The resulting pipeline algorithm achieves a saturation of the disk resource which is the bottleneck in *out-of-core* algorithms.

Keywords: Out-of-Core, pipeline, wavefront algorithm, overlap.

Résumé

Plusieurs algorithmes de calcul numérique exhibent des dépendances qui entraînent un front d'onde dans le calcul. Selon la distribution de données choisie, pipeliner les communications et les calculs peut être le seul moyen d'éviter une exécution séquentielle du code parallèle. Le grain de calcul doit être choisi intelligemment pour obtenir dans le même temps un maximum de parallélisme et un faible surcoût de communication. D'un autre côté, lorsque la taille des données excède la capacité mémoire de la plate-forme cible, les données doivent être stockées sur disque. Le concept de calcul *out-of-core* tend à minimiser l'impact des entrées/sorties nécessaires à un calcul sur de telles données. Ce concept a déjà été appliqué avec succès sur plusieurs applications d'algèbre linéaire. Dans cet article, nous appliquons des techniques *out-of-core* aux algorithmes par vagues. L'originalité de notre approche est de recouvrir le calcul, les communications et les entrées/sorties. Nous proposons une stratégie originale utilisant plusieurs blocs de mémoire accédés cycliquement. L'algorithme pipeliné qui en résulte parvient à saturer la ressource disque qui constitue le goulot d'étranglement des algorithmes *out-of-core*.

Mots-clés: Out-of-Core, pipeline, algorithme par vague, recouvrement.

1 Introduction

Parallel distributed memory machines improve performance and memory capacity but their use adds an overhead due to the communications. Consequently, to obtain programs that perform and scale well, this overhead has to be minimized. The choice of a good data distribution is of course the first step to reduce the number and the size of communications. Furthermore, the communication layer should have the lowest latency possible to allow small messages to be sent. Usually, this latency is reduced at the hardware level but increases dramatically as several software layers are added. Buffering also increases the communication overhead. These overheads can be lowered inside the communication library itself. Thereafter, remaining communications should be hidden as much as possible. Depending on the dependences within the code, asynchronous communications are useful to overlap communications with computations. The call to the communication routine (send or receive) will be made as soon as possible in the code. A wait routine will then check for the completion of the communication. Unfortunately, this is not always possible because of dependences between computations and communications. Macro-pipelining methods make overlap feasible by reordering loops [13] and adding pipeline loops. These techniques can be used for several applications with wavefront computations like the ADI [15, 19, 18], Gauss-Seidel [2], SOR [16], or the Sweep3D [10, 21] algorithms.

Figure 1 presents a macro-pipeline wavefront algorithm working on two-dimensional data. Many wavefront algorithms use two phases. On the first phase, a wavefront is started following horizontal dependences (top of Figure 1). Thanks to the block-row distribution, this first wave can be computed with no communication. On the second wave (following vertical dependences), two solutions can be chosen. First, the matrix can be redistributed using a transposition to avoid further communication in the second wave. However, this global operation often adds a great overhead. An interesting solution consists in using macro-pipeline techniques. If a pipeline loop is added (bottom of Figure 1), the execution is pipelined and communications and computations can also be overlapped. The computation grain has to be carefully chosen depending of the communication/computation ratio and the magnitude of communication latency [9].

When the size of data exceeds the memory capacity of the target platform we have to manage these data carefully. Indeed the classic use of the virtual memory manager of the operating system may induce a tremendous loss of performance [6]. As large data are stored on disk, it is necessary to introduce the new concept of *out-of-core* computation, the goal of which is to minimize the impact of the I/O needed to perform a computation on such data [17]. The main idea of this concept is to maximize the usage of the data while they are loaded in memory.

Out-of-core algorithms do not use pipelining techniques because usually the cost of accessing data stored on disks is so high that no gain can be obtained. However, as the speed of disks and disk interfaces increases, this cost can be lowered and macro-pipeline techniques can be applied to out-of-core wavefront algorithms.

Our contribution is the following. We present an algorithm combining pipelining and out-of-code techniques for wavefront applications. Our strategy is based upon the use of three different memory blocks to be able to overlap communication and computation with I/O. Our goal is to saturate the disk resource while lowering the overhead of communications. We present a generic *out-of-core* wavefront algorithm for which no assumptions were made neither on the computation complexity nor on the frontier sizes. Our theoretical results are

```

do k = 1, K
  do i = 1, M
    do j = 1, N
      computation
    enddo
  enddo
enddo

do k = 1, K
  if (myId > 0)
    send(myId-1, Top Frontier)
  if (myId < P-1)
    recv (myId+1, Top Frontier)
  do jj = 1, N, NB
    jjmax = jj+NB-1
    if (myId > 0)
      recv (myId-1, Bottom Frontier)
    do j = jj, jjmax
      do i = 1, M/P
        computation
      enddo
    enddo
    if (myId < P-1)
      send(myId+1, Bottom Frontier)
    enddo
  enddo
enddo

```

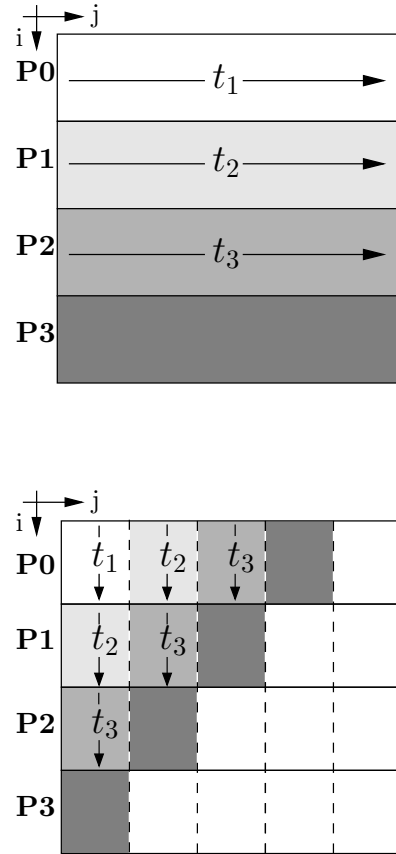


Figure 1: Pipelined loop nest and execution scheme for a block-row distribution.

corroborated by experiments. We also show that it is not necessary to fill the memory to have good performance. It is better to find the appropriate tradeoff between requirements of both pipelining and *out-of-core* techniques.

The rest of this paper is structured as follows. In Section 2, we present our general scheme for overlapping communications, computations, and I/Os. In Section 3, we explain our generic algorithm for pipelined wavefront algorithms. Finally, before a conclusion and some hints for future work, in Section 5, we present our experiments on a cluster of SMP processors connected through a Fast Ethernet.

2 Overlapping Computation and Communication with I/O

Before detailing the main idea of our *out-of-core* wavefront algorithm, we explain why and when classical *in-core* techniques can not be applied. We also give a sketch of a theoretical *in-core* wavefront algorithm applied on an *out-of-core* dataset, *i.e.*, stored on disk before and after the computation. We claim that such an execution is equivalent, in terms of execution time, to the execution of an *out-of-core* wavefront algorithm without overlap. Finally we discuss the feasibility of the overlapping of computations and communications with I/O, and

then detail the main contribution of this paper.

Figure 2 shows the memory consumption of wavefront algorithms when data are distributed following a row distribution on a ring of P processors. Each processor owns M/P rows of the data. Each row contains N elements. In case of an *out-of-core* data, this $(M/P) \times N$ block is stored on disk. The bottom part of this Figure means that processors P_1 to P_{P-1} have to send some of the first rows of their partition to their left neighbor in the ring. This part coming from another processor is needed to perform the computation. We denote this block of rows as the *Top Frontier (TF)*. Depending on the applied wavefront algorithm, the number of rows of *TF* may vary. For instance, if we perform a mean filter with a 3×3 neighbor kernel, only one row is needed. Moreover it is common in wavefront algorithms to perform this communication in one step before beginning the computing of an iteration.

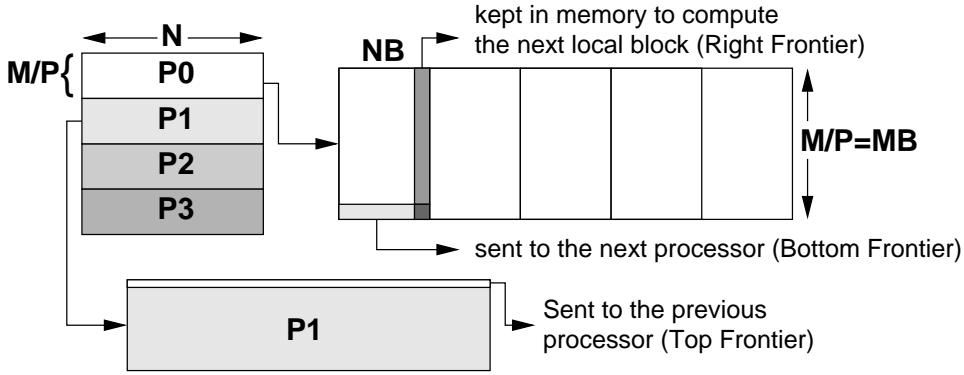


Figure 2: Memory consumption of wavefront algorithms.

The right part of Figure 2 describes which part of the $(M/P) \times N$ block of data stored on disk is actually loaded in memory. As said in the introduction, a coarse grain wavefront approach will divide the computation of an iteration in $\lceil N/NB \rceil$ steps. A $MB \times NB$ block will thus only be loaded in memory at a given moment. Therefore two parts of this block have to be distinguished. Both of them correspond to updated data that can not be written on disk at the end of the step. The first one, denoted as *Right Frontier (RF)*, is needed to compute the next step of the current iteration on the same processor. The second one is the *Bottom Frontier (BF)*, that has to be sent to the right neighbor to allow it to update its data. As for *TF*, the thickness of *RF* and *BF* directly depends on the applied wavefront algorithm.

$$Max_elt = BF + TF + RF + MB \times NB. \quad (1)$$

Equation 1 gives the maximal number of elements that it is possible to load in memory without swapping. This bound is driven by the memory consumption of processors P_1 to P_{P-2} which have to store the *TF* coming from their right neighbor and the *BF* coming from their left neighbor. The size of the 3 frontiers (Top, Bottom and Right) come from the application. MB is obtained by dividing the number of rows of the data by the number of processors of the platform. NB is then the only remaining parameter of this equation we can modify to tune our *out-of-core* wavefront algorithm. If Mem is the available memory space, Equation 2 gives an upper bound for NB denoted as NB_{max} .

$$NB_{max} = \left\lfloor \frac{Mem - (BF + TF + RF)}{MB} \right\rfloor. \quad (2)$$

To update a dataset stored on disk using a wavefront algorithm, there is at least one read and one write of the file describing this data, even if the target platform has enough memory to store the data *in-core*. In such a case, a single iteration of an *in-core* wavefront algorithm can be divided into the following steps: (i) load all data into memory; (ii) update the data using an algorithm similar to the one presented in Figure 1; (iii) write the modified data on disk.

The main objective of our *out-of-core* wavefront algorithm is to add a minimal I/O overhead to this lower bound. It is easy to propose a basic *out-of-core* approach with a similar complexity, without regard of the number of I/O latencies. The concept of this *out-of-core* algorithm is to first divide the data into *in-core* blocks and then for each block: (i) load it into memory; (ii) update it using an *in-core* wavefront algorithm; (iii) write the modified block on disk.

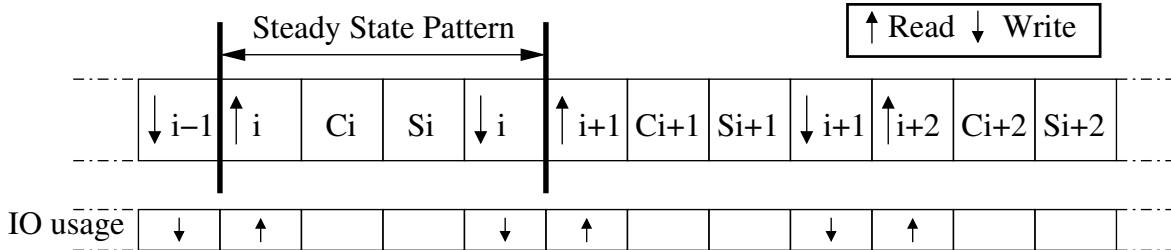


Figure 3: Succession of operations and I/O usage in a basic *out-of-core* wavefront algorithm.

The top part of Figure 3 represents the succession of operations (*Read* (\uparrow), *Compute* (C), *Send* (S) and *Write* (\downarrow)) that appear during the steady state part of the execution of a single iteration of the basic *out-of-core* algorithm. This is a normalized representation, I/O costs being greater than computation and communication costs by an order of magnitude. The labels (i , $i + 1$, ...) represent the indices of *in-core* blocks.

What we can see on this Figure is that gaps appear between I/O operations corresponding to the update of the block. The work presented in this paper aims at developing an original *out-of-core* wavefront algorithm where these idle times in the I/O usage are removed. To do so computations and communications have to be overlapped with I/O.

In the next section, we will give models of the different operations involved in our *out-of-core* wavefront algorithm and determine in which conditions such overlapping is possible.

To be able to overlap, we load 3 blocks of data into memory instead of one. These blocks of data are three times smaller than what is loaded by the basic *out-of-core* algorithm presented above, as memory is actually divided into three distinct memory blocks. While an I/O operation (a *Read* or a *Write*) is applied on the first memory block, the computation is performed on the data stored in the second one and a part of the data of the third memory block is communicated to an other processor. The problem is then to find a way to fill these memory blocks without introducing idle times in the I/O usage. The steady state part of a pattern satisfying this constraint is shown in Figure 4.

The top part of Figure 4 shows a normalized representation of the succession of operations that appear during the steady state part of the execution of a single iteration of our *out-of-core* wavefront algorithm. To remove the idle times in the I/O usage (represented by the bottom part of the Figure), the writing of an update block of data is delayed. The length of this delay allows our algorithm to perform two I/O operations. For instance, once the update

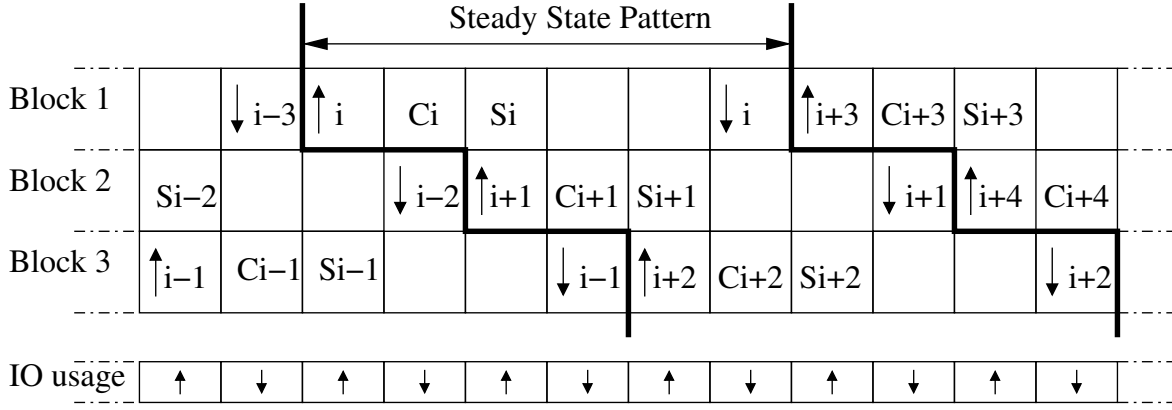


Figure 4: Succession of operations and I/O usage when 3 memory blocks are used.

of the i^{th} is finished, and before this block being written on disk, the $(i-1)^{th}$ block of data is written and the $(i+2)^{th}$ block is loaded into memory. We can also see that we have a circular use of the three memory blocks.

If we only consider the I/O usage, we see that writes occur after reads. More precisely, the read of the i^{th} block of data is followed by the write of the $(i-2)^{th}$ block. Thanks to the overlapping of computation and communication with I/O, our algorithm achieves a saturation of the disk resource.

This strategy using three memory blocks implies modifications of Equations 1 and 2. The maximal number of elements that it is possible to load in memory without swapping is now given by Equation 3 and the upper bound NB_{max} is given by Equation 4.

$$Max_elt = BF + TF + RF + 3 \times MB \times NB. \quad (3)$$

$$NB_{max} = \left\lfloor \frac{Mem - (BF + TF + RF)}{3 \times MB} \right\rfloor. \quad (4)$$

In the next section, we will provide a model of the steady state part of our *out-of-core* wavefront algorithm and also detail the initialization and the end of it.

3 Modeling a Generic Out-of-Core Wavefront Algorithm

Our target platform is a cluster of PCs. This kind of platform is homogeneous in terms of computation capabilities, memory, disk and network. The input of the generic wavefront algorithm is a matrix.

In the remainder of this paper, we assume that all operations can be overlapped. The only exception is the overlapping of I/O and communication latencies with computation, as these operations are executed by the processor. They have to be performed sequentially. We now describe generic models for communication, I/O and computation.

Due to the communication scheme of wavefront algorithms, the chosen network topology is a ring. To model the communication costs, we chose the well known $\beta_c + L\tau_c$ model, where β_c and τ_c are the latency and the data transfer rate for a communication link and L the size of the message to transfer. The latency β_c can be divided into two components: β_S and β_R

which respectively are send and the receive latencies. In the remaining of this paper we denote the communication of a dataset of size L between two processors by the following function:

$$Comm(L) = \beta_R + \beta_S + L\tau_c. \quad (5)$$

It is very hard to find a analytical model that fits the actual experimental behavior of modern disks. One possible solution is to use the same kind of model as is used in communication modeling and to instantiate it with the theoretical latency and transfer rate provided by the disk vendor. The second possible solution is to use an existing I/O benchmark tool, but this task is not trivial as we discuss bellow.

In this section, we discuss the difficulty of estimating disk performance. As we have shown, a modelization based on I/O performance require to know the I/O throughput. A lot of tools seem available on the web and it seems easy to determine this information. However, we show here that is not so evident. On-line survey of tools to evaluate disk performance are given in the following section. We describe step by step the research of a suitable tool.

The first difficulty is to find an appropriate tool for our targeted platform, in our case Linux. Many existing tools lack Linux implementations (e.g. [23] or IOCALL [14]) or are simply outdated (e.g. DISKTEST [1]). They are benchmarks that measure OS performance and the system call interface for Unix system. However, I/O measurement needs to call system functions that reduce the portability of these tools even between Unix-like OS. Thus, these tools are inappopriate on our targeted platform for I/O analysis.

Bonnie v1.4 [4] is a tool to determine the speed of filesystems, OS caching, the underlying device and libc. The goal of bonnie is to make sure that these are real transfers between user space and the physical disk. We try different benchmarks, the sequential output and sequential input, with two approaches: write per-character and by block respectively corresponding to the `putc()` and `write()` evaluation. We take the average of ten executions for five file size (from 5GB to 9GB) and the results are shown in Table 1.

Sequential Input		Sequential Output	
per-character	block	per-character	block
0.79 MB/s	37.06 MB/s	26.50 MB/s	34.54 MB/s

Table 1: Results of bonnie benchmarks. The experiment is an average of ten execution for five file size (from 5GB to 9GB)

Due to the overhead of the filesystem and operating system layers, the result are poor in comparison to the vendor-specified disk transfer rate of our SCSI disks estimated to 160MB/s. Moreover, the result of sequential input per-character can be used. Nevertheless, it's not a surprise but we can notice that block access is better than per-character access. In the same manner, we compare this result with Bonnie++ 1.03a [3]. With the same experimental protocol, results are more stable.

In this experiment the problem linked to the modelization is double. First we can see the throughput is not the same for the read (Figure 5(a)) and write (Figure 5(b)) phase. Second, the matrix size has an impact on disk throughput. For the algorithm we read by block to increase the performance but that increases the impact of the matrix size. That means it's very hard to give an accurate value to the modelization for all case.

To validate this result we have used a last tool call IOzone [12]. IOzone is a filesystem benchmark tool. The benchmark generates and measures a variety of file operations

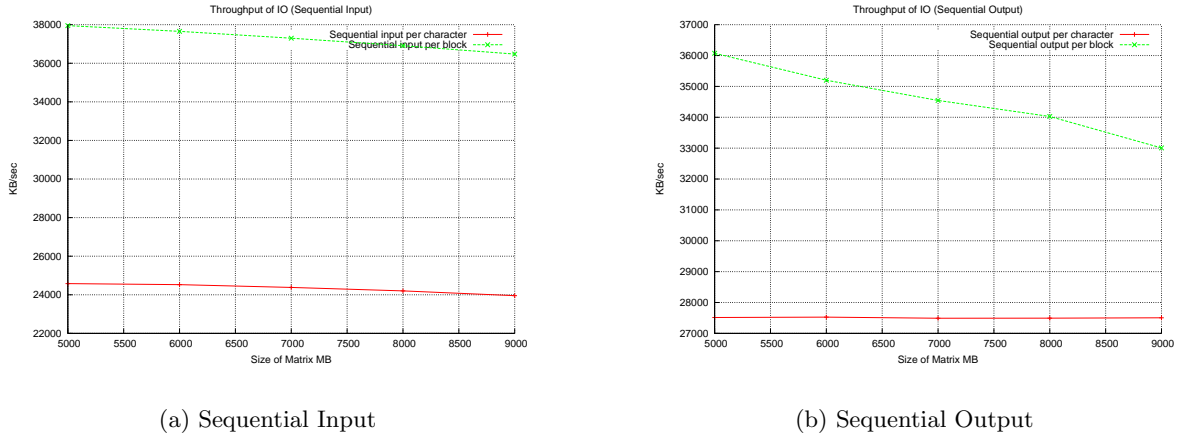


Figure 5: Bonnie++ Benchmark

(Read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read, pread, mmap, aio_read, aio_write). We have performed the same experiments: for sequential input, we consider `read()` and `fread()` function (Figure 6(a)), and for sequential output we consider `write()` and `fwrite()` function (Figure 6(b)). This tool gives different results than Bonnie++, but the throughput is in the same range of values. The results reinforce our previous conclusion : the file size has an impact on the throughput.

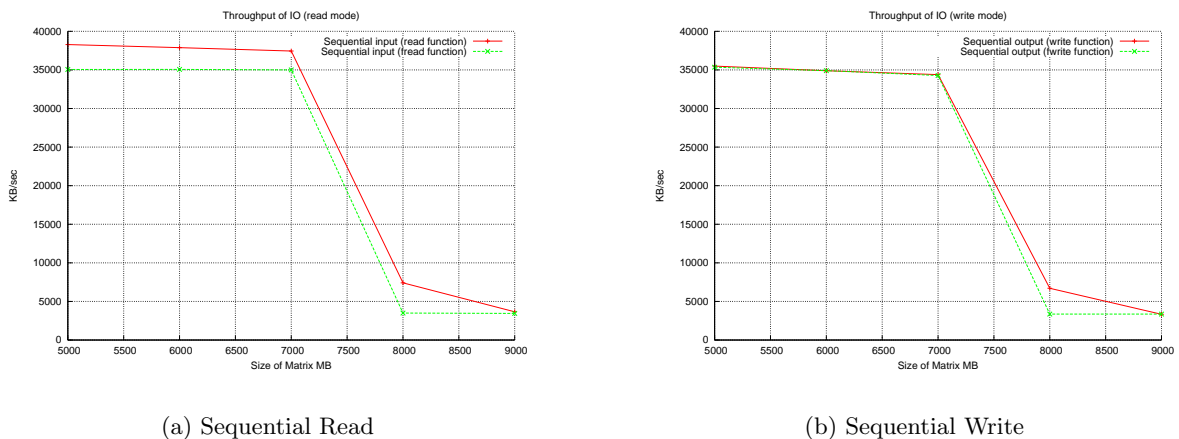


Figure 6: IOzone Benchmark

To increase the accuracy of modelization, the solution is to consider the throughput for a specific file size. That means the disk throughput is a dynamic value and not a static value.

In both cases, theoretical (i.e. vendor information) or experimental result (i.e. benchmark tool) we can apply the same model based on equation 6.

$$Io(L) = \beta_{io} + L \tau_{io}. \tag{6}$$

To model computation costs, a common technique is to represent the complexity of the

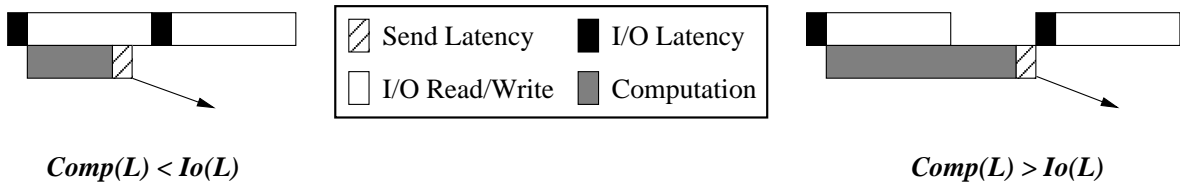
computation by a polynomial $f(L)$, where L is the size of the data, and then divide the expression by the speed of the processor. This speed is expressed in millions of floating operations executed by the processor in a second (Mflops). We thus have:

$$Comp(L) = \frac{f(L)}{speed}. \quad (7)$$

3.1 Generic Model

In this section we detail the different phases of our *out-of-core* wavefront algorithm, in the case where communication and computation operations can be overlapped with I/O operations, as assumed in the previous Section.

If we only consider the algorithmic constraints induced by our algorithm, overlapping is possible if and only if the time to update a block of data is less than the time to read or write it on disk. If not, gaps appear between I/O operations as shown below.



Assuming that target applications have an $\mathcal{O}(N^2)$ complexity, we claim that, on modern architectures, it is longer to read or write a block of data than performing such a computation on it.

We first detail one single iteration of our algorithm and then the case of K iterations. To do this presentation, we target a platform composed of $P = 4$ processors (P_0 to P_3). Each *in-core* block of data is of size $MB \times NB$, where NB is variable.

3.1.1 One Single Iteration

Classical *in-core* wavefront algorithms are composed of three phases: *synchronization*, *load* and *steady state*. To correctly handle I/O, we have to add a fourth phase to *flush* the memory and write the result on disk. We now detail and propose a model for each of these phases.

To be able to perform the computation on the first block of data, processors P_1 to P_3 have to read TF from disk (R_0) and send it to their left neighbor in the communication ring. All processors also have to load the first block into memory (R_1). The left part of Figure 7 shows the schedule of these operations. It should be noted that the initial data fetch operations on each processor are executed in parallel.

It has to be noticed that loading TF from disk is more expensive than loading a block of data. Indeed, due to implementation constraints, TF had to be read element by element, while a block of data correspond to a single read of $MB \times NB$ elements. We introduce a small optimization by performing R_1 before R_0 as shown in Figure 7. The NB first elements of TF are thus loaded during R_1 and do not have to be loaded a second time during R_0 .

We can see that the critical path of this phase is independent of the number of processors of the platform. For sake of readability, the optimization presented above do not appear in the model of the *synchronization* phase given by Equation 8, as it implies a negligible gain.

$$T_{synchro} = I_o(MB \times NB) + I_o(TF) + Comm(TF). \quad (8)$$

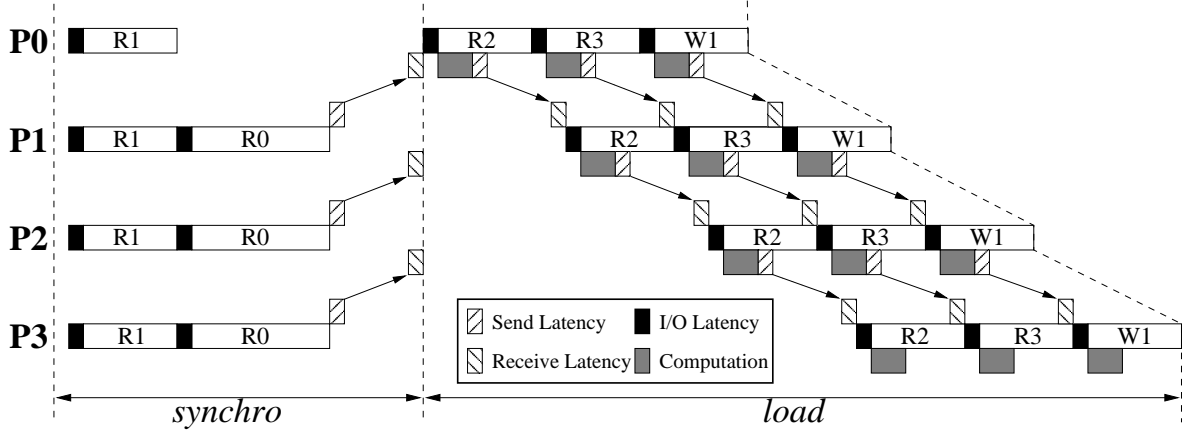


Figure 7: *Synchronization* and *load* phases.

In opposition to the *synchronization* phase, the *load* phase, shown by the right part of Figure 7, directly depends on the number of processors of the target platform. The critical path of this phase corresponds to the time processor P_3 waits before computing on its first block of data plus the time needed by this processor to reach a steady state. The time waited by processor P_3 can be decomposed as follows. Each processor between P_0 and P_2 initiates the fetch of its second block of data, then performs the computation on the first block and sends its BF to its right neighbor. Processor P_3 is eventually ready to compute on its first block of data once the reading of the second block is initiated.

We consider that a steady state is reached once the three memory blocks are filled and processors alternate *read* and *write* I/O operations. It takes three I/O operations (two reads (R_2 and R_3) and one write (W_1)) to processor $P - 1$ to reach such a steady state. The critical path of our *load* phase can then be modeled as:

$$T_{load} = (P - 1) (\beta_{io} + Comp(MB \times NB) + Comm(BF)) + 3 I_o(MB \times NB). \quad (9)$$

These two phases imply a tradeoff in the determination of the optimal NB , as no parallelism is exhibited between them. Indeed when usual *out-of-core* techniques try to use large data blocks to fill the memory, pipeline algorithms tend to use smaller blocks to reduce the time spent before reaching the steady state.

One could think that very simple and effective technique to improve performance is to adapt the size of NB depending on the current phase of the algorithm. Indeed, it seems to be more efficient to use a small block size, denoted nb , in the first two phases (to decrease the set-up time of the algorithm) and a larger one, NB in the last two phases (to reduce the number of steps and thus the number of I/O latencies). But experiments shown such a technique is inefficient as we can see in Figure 8. In this experiment, we apply an instance of our generic *out-of-core* wavefront algorithm (which will be detailed in Section 5) on a square matrix of size $N = 51200$.

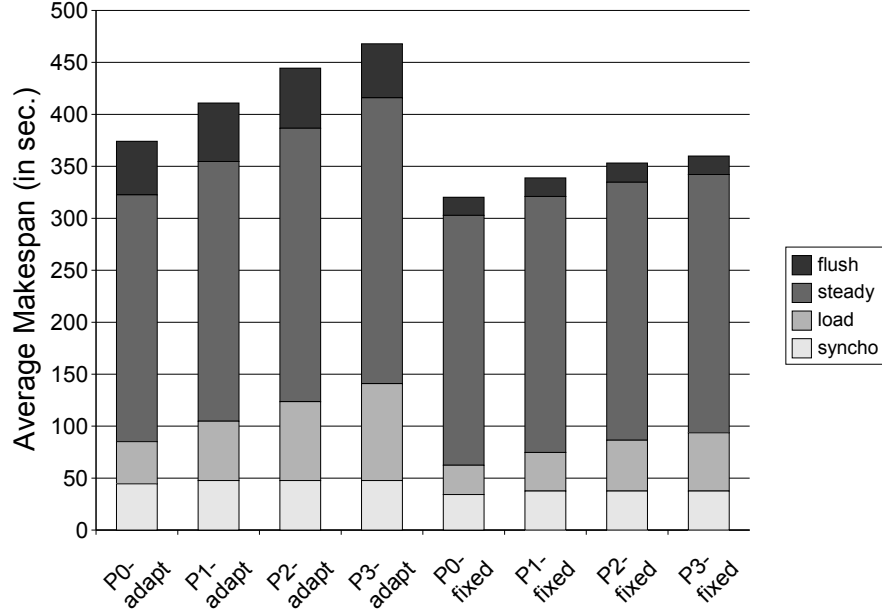


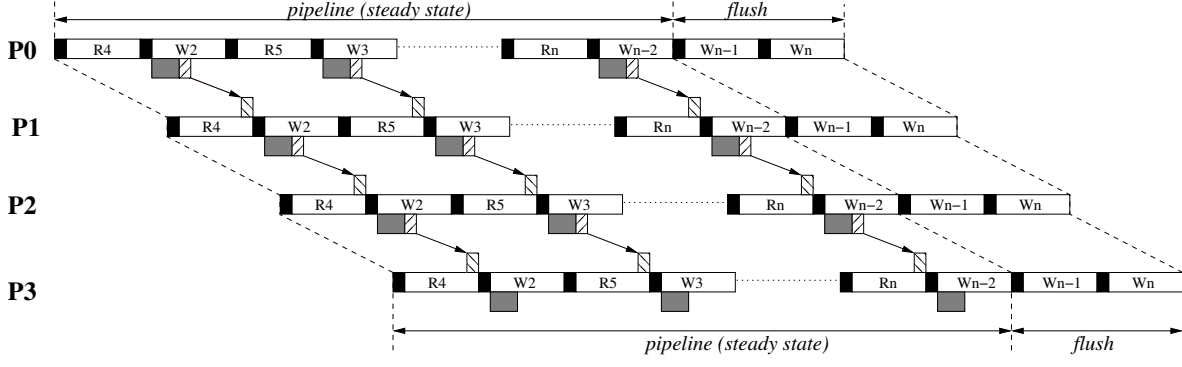
Figure 8: Comparison between executions with one and two block sizes.

The left part of Figure 8 shows the average makespan of the algorithm using two different block sizes ($nb = 512$ and $NB = 6336$) whereas in the right part a fixed block size ($NB = 3200$) is used. Note that nb is only used for the first block of data, as using it for the second and third block – used in the *load* phase – induces a significant modification of the *steady state* phase. We can see that the adaptive version does not achieve better performance neither for the global makespan nor in the first two phases. There are several explanations for this. Reducing the size of the first block does not reduce the time spent in the *synchronization* phase because of the optimization about *TF* presented above. For the *load* phase the time spent is higher with two block sizes as we read more elements than with a fixed block size (13184 vs. 9600). Although we may gain on the computation and the communication of the first block, the I/O operations are still predominant.

The steady state pattern we detail here differs slightly from the memory pattern given by Figure 4. In that Figure, the pattern covers the update of 3 blocks of data. In the left part of Figure 9, we consider that the pattern is composed of the reading of the i^{th} block of data, the computation on this block, the communication of its *BF* to the right neighbor and the writing of the $(i-2)^{th}$ block of data. This pattern corresponds to the update of a single block of data, as the writing of modified data is delayed as explained in Section 2.

The computation is overlapped with the writing. The communication is also overlapped with the writing on the sender side and with the reading of the $(i+1)^{th}$ on the receiver side. The critical path of this phase is then only composed of I/O operations. As three blocks of data have been updated in the *load* phase, and the computation is divided in N/NB steps, $(N/NB) - 3$ are updated in the *steady state* phase. This leads us to:

$$T_{steady\ state} = \left(\frac{N}{NB} - 3 \right) (2Io(MB \times NB)). \quad (10)$$

Figure 9: *Steady state* and *flush* phases.

As said before, each read operation of the *steady state* phase is followed by a write operation, but with a gap of two between the indexes of the blocks. This gap is due to the fact that three reads but only one write are performed before reaching the steady state. Once the computation has been applied on all blocks, two blocks remain in memory and have to be written (W_{n-1} and W_n) on disk. As we are still considering the critical path of our *out-of-core* wavefront algorithm, we modeled these writes on processor P_{P-1} , which finishes the algorithm.

$$T_{flush} = 2 I_o (MB \times NB). \quad (11)$$

A generic model our *out-of-core* wavefront algorithm is then given by Equation 12:

$$\begin{aligned}
 T_{total} &= T_{synchro} + T_{load} + T_{steady\ state} + T_{flush} \\
 &= I_o(TF) + Comm(TF) + I_o(MB \times NB) \\
 &\quad + (P - 1) (\beta_{io} + Comp(MB \times NB) + Comm(BF)) \\
 &\quad + 3 I_o(MB \times NB) + \left(\frac{N}{NB} - 3 \right) (2 I_o(MB \times NB)) \\
 &\quad + 2 I_o(MB \times NB) \\
 &= I_o(TF) + Comm(TF) \\
 &\quad + (P - 1) (\beta_{io} + Comp(MB \times NB) + Comm(BF)) \\
 &\quad + 2 \left(\frac{N}{NB} \right) I_o(MB \times NB) \quad (12)
 \end{aligned}$$

The last term of this model corresponds to the reading and writing of the whole matrix, performed in N/NB steps. The remainder of the equation represents the overhead of our *out-of-core* wavefront algorithm with regard to these mandatory I/O operations.

3.1.2 K Iterations

Most of the target applications of our *out-of-core* wavefront algorithm, presented in Section 1 perform the same computation on the whole dataset several times. This is expressed by the outer loop in Figure 1. In this Section, we propose a generic model of our *out-of-core* wavefront algorithm when K iterations are performed.

It has to be noticed that an *in-core* wavefront algorithm will read and write the data only once and perform the K iterations in between. In contrast, an *out-of-core* wavefront algorithm will have to read and write the data K times, as the data are modified by each iteration. The performance of an *out-of-core* execution with regard to an ideal *in-core* execution should then be worse by an order of K . But in the case of memory-bound systems, such *in-core* algorithms can not be applied. In [3], Caron *et al.* introduced the concept of *hole effect* that appears when the size of data exceeds the available memory and results in the collapse of the pagination system. This effect is mainly due to the memory manager policy used to select pages to write back when physical memory becomes insufficient. Usual memory managers use LRU (Least Recently Used) or FIFO like policies which are not well suited for linear accesses to memory as in data-parallel applications.

```
float V[N];
do j = 1, P
  do i = 1, N
    V[i] = f(i, j)
  enddo
enddo
```

Figure 10: Several linear accesses to the memory.

To illustrate this concept of *hole effect*, let consider the program shown on Figure 10. This program is very simple but sufficient to show what happens when an *in-core* wavefront algorithm is applied on data exceeding the available memory. In this program, there are P linear accesses to a vector V of N elements. Let M be the number of physical memory pages, and let B be the size of a page. Let us consider the situation where $\lceil N/B \rceil > M$, for instance $N = M \times B + 1$, and $V[1..N - 1]$ are initially in the physical memory. The access to $V[N]$ causes a page fault, and the LRU policy removes the page which contains $V[1]$ from physical memory. Unfortunately, it is the next page to be accessed. The next iteration will generate another page fault. This new page fault removes the page which contains $V[B + 1]$ from memory, *i.e.*, the next page to be accessed. This phenomenon, the *hole effect* occurs each time a new page is accessed. The number of disk accesses is equal to $2 \times \lceil N/B \rceil \times (P - 1) + 1$, and is independent of the physical memory size, whenever $N > M \times B$.

To implement an iterative *out-of-core* wavefront algorithm, the only way is to apply K times the pipeline exactly as described in Section 3.1.1. Indeed, before computing a new iteration, each processor has to send an updated version of its TF to its left neighbor. This implies that every block of data updated by a given iteration has to be written on disk before starting the next iteration.

Once processors P_1 to P_{P-1} have performed their last writes for iteration i , they begin the *synchronization* phase for iteration $i + 1$, *i.e.*, they read and transfer the TF . This synchronization phase is followed by the *load* phase for iteration $i + 1$. However, the critical path of the *transition* phase is different of that of *synchronization* and *load* phases. Indeed this path runs between processors P_{P-1} and P_{P-2} and leads to Equation 13.

$$T_{transition} = I_o(TF) + Comm(TF) + I_o(MB \times NB) + \beta_{io} + Comp(MB \times NB) + Comm(BF)$$

$$+3 I_o(MB \times NB). \quad (13)$$

A generic model our *out-of-core* wavefront algorithm when K iterations are performed is then given by Equation 14:

$$\begin{aligned} T_{total}^K &= T_{synchro} + T_{load} + K \times (T_{steady\ state} + T_{flush}) + (K - 1) \times T_{transition} \\ &= K (I_o(TF) + Comm(TF)) \\ &\quad + (P + K - 2) (\beta_{io} + Comp(MB \times NB) + Comm(BF)) \\ &\quad + 2K \left(\frac{N}{NB} \right) I_o(MB \times NB). \end{aligned} \quad (14)$$

As with Equation 12, the last term of Equation 14 corresponds to the K readings and writings of the whole matrix. It has to be noticed that even in the case of an *in-core* execution, TF has to be read and communicated between iterations as the data it contains has been updated. The remainder of this equation represents the time spent to load (or reload) the pipeline at each iteration.

4 Model Instantiation

We chose to instantiate our analytic model using a *out-of-core* wavefront algorithm similar to a mean filter applied to a square matrix ($M = N$ and so forth $MB = N/P$) of double precision elements. A convolution product is performed for each element of the initial matrix. Four neighbors (North, South, East and West) are used to compute the update. This implies that RF is of size $MB \times 1$, BF of size $1 \times NB$ and TF of size $1 \times N$. According to these values, we can rewrite Equations 12 and 14 only in terms of N , NB , P and parameters modeling the execution platform.

From these new equations we are able to determine an optimal block size, denoted as NB_{opt} , for the application of our *out-of-core* wavefront algorithm on a matrix of size N executed on a platform made of P processors. The chosen optimality criterion is the minimization of the makespan of our application. The computation of the optimal block size is given in the next two sections respectively for one and K iterations.

4.1 Optimal Block Size Computation for One Single Iteration

To be able to determine an optimal value for NB from Equation 12, we first have to replace the chosen values for BF , RF and TF . Then we use the models for communication, I/O and computation given in Equations 5, 6 and 7 respectively. This leads to Equation 15 given below.

$$\begin{aligned} T_{total} &= I_o(N) + Comm(N) \\ &\quad + (P - 1) \left(\beta_{io} + Comp \left(\frac{N \times NB}{P} \right) + Comm(NB) \right) \\ &\quad + 2 \left(\frac{N}{NB} \right) I_o \left(\frac{N \times NB}{P} \right) \\ &= \beta_{io} + N\tau_{io} + \beta_R + \beta_S + N\tau_c \end{aligned}$$

$$\begin{aligned}
& +(P-1) \left(\beta_{io} + \frac{5N \times NB}{P \times \text{speed}} + \beta_R + \beta_S + NB\tau_c \right) \\
& + 2 \left(\frac{N}{NB} \right) \left(\beta_{io} + \frac{(N \times NB) \tau_{io}}{P} \right) \\
& = P (\beta_R + \beta_S + \beta_{io}) + N (\tau_c + \tau_{io}) + \frac{2N^2 \tau_{io}}{P} \\
& + \frac{2N \beta_{io}}{NB} + NB \times (P-1) \left(\frac{5N}{P \times \text{speed}} + \tau_c \right). \tag{15}
\end{aligned}$$

Deriving Equation 15 with regard to NB we obtain the following derivative :

$$\frac{\partial}{\partial NB} T_{total} = (P-1) \left(\frac{5N}{P \times \text{speed}} + \tau_c \right) - \frac{2N \beta_{io}}{NB^2} = 0, \tag{16}$$

The positive solution of Equation 16 is:

$$NB_{opt} = \sqrt{\frac{2P \text{ speed } \beta_{io}}{(P-1)(5 + P \text{ speed } \tau_c)}}. \tag{17}$$

We can verify that this solution is actually a minimum for T_{total} by computing the secondary derivative :

$$\frac{\partial^2}{\partial NB^2} T_{total} = \frac{4N \beta_{io}}{NB^3}, \tag{18}$$

which is positive at this point.

4.2 Optimal Block Size Computation for K Iterations

Based on our one-iteration solution, we can rewrite Equation 14 to determine an optimal block size for K iterations. By the same process we obtain Equation 19 below.

$$\begin{aligned}
T_{total}^K & = K (Io(N) + Comm(N)) \\
& + (P+K-2) \left(\beta_{io} + Comp \left(\frac{N \times NB}{P} \right) + Comm(NB) \right) \\
& + 2K \left(\frac{N}{NB} \right) Io \left(\frac{N \times NB}{P} \right) \\
& = K (\beta_{io} + N\tau_{io} + \beta_R + \beta_S + N\tau_c) \\
& + (P+K-2) \left(\beta_{io} + \frac{5N \times NB}{P \times \text{speed}} + \beta_R + \beta_S + NB\tau_c \right) \\
& + 2K \left(\frac{N}{NB} \right) \left(\beta_{io} + \frac{(N \times NB) \tau_{io}}{P} \right) \\
& = (P+2K-2) (\beta_R + \beta_S + \beta_{io}) + NK (\tau_c + \tau_{io}) + \frac{2KN^2 \tau_{io}}{P} \\
& + \frac{2KN \beta_{io}}{NB} + NB \times (P+K-2) \left(\frac{5N}{P \times \text{speed}} + \tau_c \right). \tag{19}
\end{aligned}$$

Deriving Equation 15 with regard to NB we obtain the following derivative :

$$\frac{\partial}{\partial NB} T_{total}^K = (P + K - 2) \left(\frac{5N}{P \times speed} + \tau_c \right) - \frac{2KN\beta_{io}}{NB^2} = 0, \quad (20)$$

As was done in the single iteration analysis, we can verify that this solution is actually a minimum for T_{total}^K by computing the secondary derivative :

$$\frac{\partial^2}{\partial NB^2} T_{total}^K = \frac{4KN\beta_{io}}{NB^3}, \quad (21)$$

which is positive at this point.

5 Experiments

In this section, we present an experimental validation of our *out-of-core* wavefront algorithm. We use the algorithm described in section 4. Experiments are performed on 4 nodes of a 24 node SMP cluster. Each SMP node is a Dual-Pentium IV Xeon 2.6GHz with 2GB of memory and a 36GB SCSI disk (15000 RPM, 160MB/s SCSI channel). Nodes are connected through a Fast Ethernet network.

We ran two sets of experiments with different sizes of matrices. In the former we apply *in-core* and *out-of-core* wavefront algorithms on a matrix of size $N = 32 \times 1024 = 32768$. Distributing such a matrix on 4 processors leads to a memory footprint of 2.0GB per processor, which is not only the available memory space of a node of our cluster but also the 32-bit address space bound. However a matrix of this size can be considered as an *out-of-core* data as a part of the memory is used by operating system.

Unfortunately we are unable to inject realistic enough I/O values into our model to actually determine optimal block size and compare the model to real world experiments. However we compare our *out-of-core* algorithm to an *in-core* algorithm similar to the one presented in [18]. In that algorithm the whole matrix is loaded into memory before applying the wavefront scheme. This algorithm works only on a single block of memory. We let the virtual memory manager of the operating system handle the *out-of-core* data.

Figure 11 shows the average completion time of one single iteration of *in-core* and *out-of-core* wavefront algorithms for $N = 32768$. As we can see, applying *out-of-core* techniques can significantly improve performance (from a factor of 2.3 in the best case to a factor of 7.11 in the worst case). We can also notice that, for the *in-core* algorithm, using a large block size ravages performance as it increases the impact of the hole effect. In our *out-of-core* wavefront algorithm, the increase of block size has a less significant impact (less than 15%) but confirms that a balance between pipeline optimization and *out-of-core* memory usage has to be found to achieve high performance. The minimum completion time is achieved for blocks of size 1024. This leads to a memory consumption of 192MB, which is far less than the available 2GB.

Figure 12 shows the average completion time of three iterations of *in-core* and *out-of-core* wavefront algorithms for matrices of same size. We can see that although the time to compute three iterations with our *out-of-core* algorithm is exactly three times longer than the time needed for one single iteration, it is not the same for the *in-core* algorithm. Although the data is read from disk only once in the *in-core* algorithm (versus three in the *out-of-core* version), letting the virtual memory manager handle an *out-of-core* data leads to very poor performance. Indeed there is a factor of 6 between the average makespans of both algorithms.

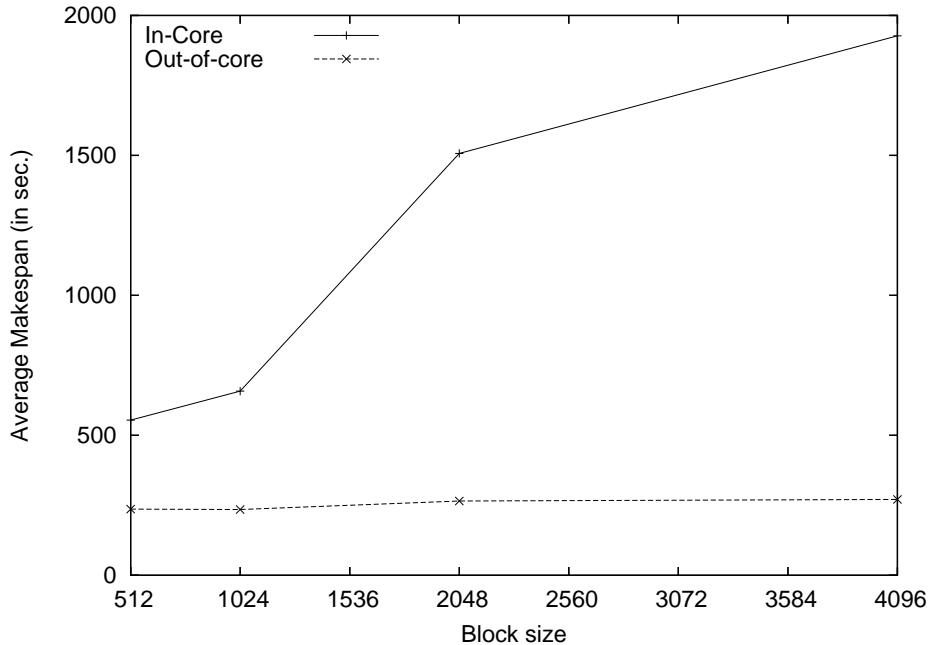


Figure 11: Average completion time of one single iteration of *in-core* and *out-of-core* wavefront algorithms for $N=32768$.

Another interesting point to notice is that the best performance for our *out-of-core* wavefront algorithm is achieved with blocks of size 2048 (corresponding to a memory consumption of 384 MB). This block size is twice as large as for one single iteration. This difference comes from the synchronization phase (where the *Top Frontier* is read) that does not depend on the block size and represents a significant part (27%) of the total time.

In the second set of experiments, we apply our algorithm on an even larger matrix of size $N = 50 \times 1024 = 51200$. For such a size, an *in-core* algorithm can not be applied at all on a 32-bit architecture as it exceeds the address space limit. Applying *out-of-core* techniques is thus mandatory.

Figure 13 presents the average completion of our *out-of-core* wavefront compared to the time needed to only read and then write the matrix. The overhead introduced by our algorithm may seem high but a part of it can be easily explained if we look further at the different steps of the algorithm and especially at the *synchronization* and *load* phases. Indeed the Read/Write time shown in Figure 13 does not include the time needed to load the *Top Frontier* into memory. As said in Section 3.1.1, this time can not be neglected as this frontier has to be read element-wise. Furthermore, the *load* phase is not parallel as each processor has to wait for a communication from its left neighbor before reading a block of data. Figure 14 shows the time needed for both phases on processor P_3 by our *out-of-core* algorithm and for our Read/Write competitor. As we can see, the main part of the overhead shown in Figure 13 comes from the first two phases. We also see that choosing a large block size, as is common in *out-of-core* techniques, is not a good solution when these techniques are combined with pipelining. Once again there is a tradeoff to be made between minimizing the time spent to load the pipeline and the minimizing the number of I/O steps.

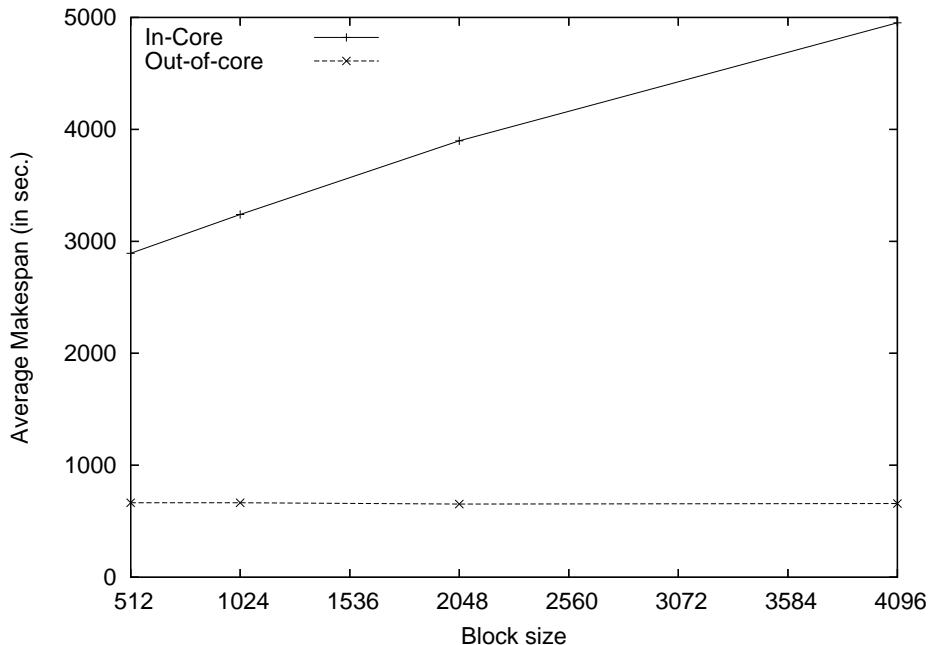


Figure 12: Average completion time of three iterations of *in-core* and *out-of-core* wavefront algorithms for $N=32768$.

6 Related Work

Macro-pipeline algorithms have been the subject of many recent research efforts. Most of them are related to the generation of efficient parallel codes from Fortran programs using annotations [18, 22]. Modelization of their behavior and the computation of optimal grain have also been studied either statically [9, 18] or dynamically [20]. The modelization of synchronous pipelined wavefront algorithms has been presented in [11] for SMP clusters. These techniques have been applied to several applications [2, 5, 15, 16, 19, 18] achieving high performance.

Concerning *out-of-core* computation, two approaches can be applied. The system approach [6] proposes a technique adapted to large amount of data to improve the virtual memory management from the operating system. The algorithmic approach [7] optimizes I/O in high-level linear algebra algorithms. The *out-of-core* computation is split in *in-core* blocks.

The only paper dealing with the overlap of communication, computation, and I/Os we found is the one from M.J. Clement and M.J. Quinn [8]. The chosen application is parallel sorting which makes slightly different constraints on the algorithm.

Our work is mainly based on the study and the extension of the generic model for macro-pipelined version of the ADI algorithm presented in [18].

7 Conclusion and Future Work

In this paper we presented an original combination of *out-of-core* and pipelining techniques applied to applications such as image processing to obtain an *out-of-core* wavefront algorithm.

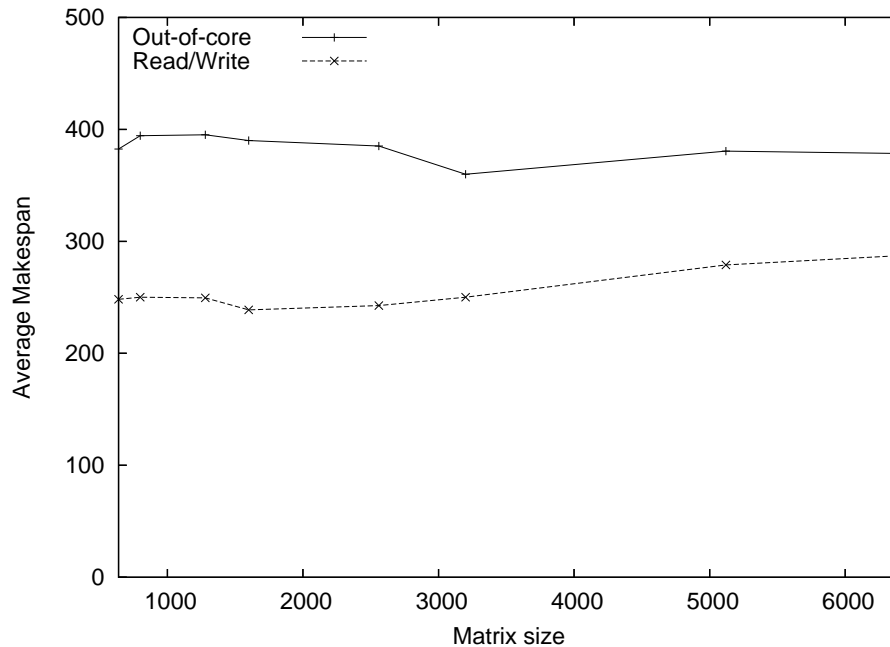


Figure 13: Average completion time of one single iteration of our *out-of-core* wavefront algorithm and of a Read/Write of the data for $N=51200$.

Our strategy is based upon the use of three different memory blocks to be able to overlap communication and computation with I/O. While we are reading (or writing) a block, we perform a computation on another and communicate the third. This scheme aims at achieving a saturation of the disk resource which is common in *out-of-core* computing.

We then presented a generic *out-of-core* wavefront algorithm for which no assumptions were made on either the computation complexity or on the frontier sizes. The frontiers are parts of the data that either have to be communicated to another processor or are kept in memory for computation. We produced an analytical model of our generic *out-of-core* wavefront algorithm to determine an optimal block size for a given target application. Experiments and disk performance estimation tools have shown us that actual I/O performance is sometimes far away from device specifications and highly dependent on the size of the data. Thus we were unable to inject realistic enough I/O values into our model to actually determine optimal block size and compare the model to experiments.

We gave however some experimental results on a particular instance of our generic algorithm corresponding to a mean filter applied on a square matrix. Experiments have shown that an *out-of-core* algorithm achieves better performance than an *in-core* one when dealing with an *out-of-core* version implying to let the virtual memory manager to handle memory usage and page swapping. We also presented results for data exceeding the address space limit of a 32-bit architecture. For such data, the use of *out-of-core* technique is mandatory. We also have shown that it is not necessary to fill the memory to have good performance. It is better to find the appropriate tradeoff between requirements of both pipelining and *out-of-core* techniques.

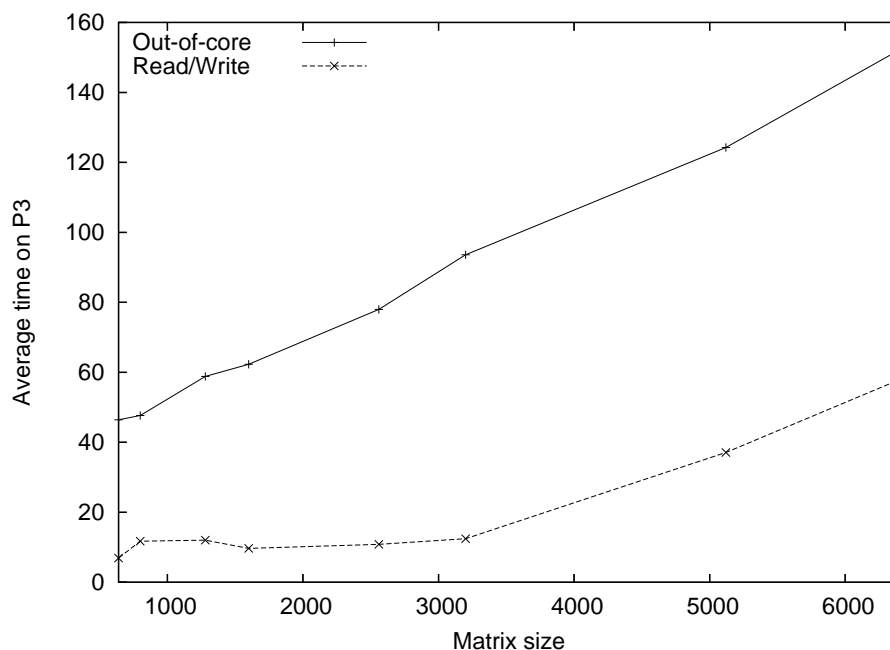


Figure 14: Time for synchronization and load phases of our *out-of-core* wavefront algorithm and of the Read/Write competitor.

References

- [1] B. Allen. Disktest. <http://www.acnc.com/benchmarks/disktest.tar.Z>, 1975.
- [2] S. Baden and S. J. Fink. Communication Overlap in Multi-tier Parallel Algorithms. In *Proceedings of SuperComputing'98*, Orlando, FL, November 1998.
- [3] Bonnie++. <http://www.coker.com.au/bonnie++/>.
- [4] T. Bray. Bonnie File System Benchmark. <http://www.textuality.com/bonnie/>, 1988. Revised version in 1996.
- [5] C. Calvin and F. Desprez. Minimizing Communication Overhead Using Pipelining for Multi-Dimensional FFT on Distributed Memory Machines. In D.J. Evans, H. Liddell J.R. Joubert, and D. Trystram, editors, *Parallel Computing'93*, pages 65–72. Elsevier Science Publishers B.V. (North-Holland), 1993.
- [6] E. Caron, O. Cozette, D. Lazure, and G. Utard. Virtual Memory Management in Data Parallel Applications. In *Proceedings of HPCN Europe'99*, volume 1593 of *Lecture Notes in Computer Science*, pages 1107–1116, Amsterdam, April 1999. Springer-Verlag.
- [7] E. Caron and G. Utard. Parallel Out-of-Core Matrix Inversion. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02)*, Fort Lauderdale, April 2002.
- [8] M. J. Clement and M. J. Quinn. Overlapping Computations, Communications and I/O in Parallel Sorting. *Journal of Parallel and Distributed Computing*, 28:162–172, 1995.

- [9] F. Desprez, P. Ramet, and J. Roman. Optimal Grain Size Computation for Pipelined Algorithms. In *Europar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 165–172. Springer Verlag, August 1996.
- [10] A. Hoisie, O. Lubeck, and H. Wasserman. Scalability Analysis of Multidimensional Wavefront Algorithms on Large-Scale SMP Clusters. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, February 1999.
- [11] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme. A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs. In *Proceedings of the 2000 International Conference on Parallel Processing (ICPP'00)*, pages 219–232, Toronto, August 2000.
- [12] IOzone. <http://www.iozone.org>.
- [13] K. Ishizaki, H. Komatsu, and T. Nakatani. A Loop Transformation Algorithm for Communication Overlapping. *International Journal of Parallel Programming*, 28(2):135–154, 2000.
- [14] T. Le and J. Stubbs. Iocall. <ftp://ftp.netsys.com/benchmark/iocall.c.gz>, 1990.
- [15] D. K. Lowenthal. Accurately Selecting Block Size at Runtime in Pipelined Parallel Programs. *International Journal of Parallel Programming*, 28(3):245–274, 2000.
- [16] T. Nguyen, M. Mills Strout, L. Carter, and J. Ferrante. Asynchronous Dynamic Load Balancing of Tiles. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, March 1999.
- [17] R. Oldfield and D. Kotz. Applications of Parallel I/O. Technical Report PCS-TR98-337, Dartmouth College, Computer Science, Hanover, NH, August 1998.
- [18] D. Palermo. *Compiler Techniques for Optimizing Communications and Data-distribution for Distributed Memory Multicomputers*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.
- [19] D. Palermo and P. Banerjee. Automatic selection of Dynamic Data Partitioning Schemes for Distributed-Memory Multicomputers. In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, August 1995.
- [20] B. Siegell and P. Steenkiste. Automatic Selection of Load Balancing Parameters Using Compile-Time and Run-Time Information. *Concurrency - Practice and Experience*, 9(4):275–317, 1997.
- [21] D. Sundaram-Stukel and M. K. Vernon. Predictive Analysis of a Wavefront Application Using LogGP. *ACM SIGPLAN Notices*, 34(8):141–150, 1999.
- [22] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Dept. of Computer Science, Rice University, January 1993.
- [23] B. Wolman and T. M. Olson. IOBENCH: a System Independent IO Benchmark. *SIGARCH Computer Architecture News*, 17(5):55–70, 1989.