Efficient tiling for an ODE discrete integration program: redundant tasks instead of trapezoidal shaped-tiles

Fabrice Rastello
STMicroelectronics*
12 rue Jules Horowitz
38019 Grenoble cédex 1, France
Fabrice.Rastello@st.com

Thierry DAUXOIS
Ecole Normale Supérieur de Lyon
Laboratoire de physique
46 allée Italie 69007, France
Thierry.Dauxois@ens-lyon.fr

Abstract

In this paper, we present an efficient and simple solution to the parallelization of discrete integration programs of ordinary differential equations (ODE). The main technique used is known as loop tiling. To avoid the overhead due to code complexity and border effects, we introduce redundant tasks and we use non parallelepiped tiles. Thanks both to cache reuse ($\times 4.3$) and coarse granularity ($\times 24.5$), the speedup using 25 processors over the non-tiled sequential implementation is larger than 106.

We also present the draft of a fuzzy methodology to optimize the tile size and we illustrate it using real measurements for the communication cost and the execution time. In particular, we observe that the model of communication latencies over a Myrinet network is not as simple as is usually reported.

1. Introduction

The goal of this paper is to present a practical implementation of tiling methods to a real-life problem, namely the study of Lyapunov exponents of the Fermi-Pasta-Ulam (FPU) chain. The basic idea of tiling, also known as loop blocking, is to group elemental computation points into tiles that will be viewed as computational units. Hence, the *communication* time (which is usually proportional to the surface of the tile) decreases while the *computation* time (which is proportional to the volume of the tile) increases. Tiling is also a widely used technique to increase the granularity of computations and the locality of data references. The program that we aim to parallelize corresponds to the discrete integration of ordinary differential equations (ODE). Existing literature on tiling [6, 2, 1, 3, 13] provides a solution to this generic problem: a diamond-shaped blocking. The main drawback of this solution is the complexity of its implementation. In the concern of our study, there is an additional difficulty: the presence of high frequency synchronization barriers (see Section 2). Hence classical methods should be adapted using hierarchical tiling: this solution is illustrated in Figure 2. We present a solution which is simpler to implement yet still very efficient: we use redundant computational tasks (see Figure 4). This solution is particularly adapted to explicit integration schemes, but can also be applied to general multidimensional SOR programs with local dependences (like heat propagation or molecular dynamics), in addition to many image processing algorithms.

Also, automatic partitioning is complex since it should take into account both the volume and frequency of communications, in addition to data reuse, code complexity, etc. In this article we give the draft of a fuzzy methodology decomposed into different successive phases, sorted in decreasing importance order. This methodology is illustrated using our practical optimization problem. Finally, we are led to testing the validity of usual assumptions made for tiling optimizations, like the formulae used to express computation and communication times. In particular, we claim that linear communication models designed for small-scale message passing with perfectly synchronized communications [10], becomes strongly incorrect when communications are performed among data based computations.

The rest of this paper is decomposed into four major sections. The problem is described in Section 2. Different partitioning solutions are discussed in Section 3. The different steps to find optimal parameters are described in Section 4. Finally, Section 5 provides performance results, together with a short discussion on heterogeneous platforms. *This article corresponds to a short version of the corresponding research report [11].*

*This work has been supported by the ENS Lyon BQR.

2. Description of the code

The program we aim to parallelize here deals with the study of Lyapnuov exponents of the Fermi-Pasta-Ulam (FPU) chain. Roughly speaking this corresponds to study the degree of stochasticity of a one dimensional dynamical system. This part is devoted to the description of the initial code. Description of the FPU model, calculation of Lyapunov exponents, motivations and further details can be found in [11].

2.1. The sequential code

After the initialization, the program is made up of three consecutive phases: during the first phase ($0 \leq t \leq t_1$), the system evolves; during the second phase ($t_1 < t \leq t_2$), the system still evolves, in addition to the tangent space that is orthonormalized every $n_{ortho}dt$ iterations time; then the third phase ($t_2 < t \leq t_3 = T$) is identical to the second one but the calculation of the Lyapunov exponents is additionally performed.

Program 1. The sequential code.

Initialization
do $t = 0, T : dt$
  if $t > t_1$ {phase 2 or 3}
    do $k = 0, 3$ {4 steps of McLachlan-Atela's alg.}
      doall $j \in \{1, \ldots, N_{lyap}\}$
        { Evolution of tangent space }
        dovect $i \in \{0, \ldots, H-1\}$
          $\delta v_i^j = \delta v_i^j + c_k \times g_k(x_{i-1}, x_i, x_{i+1}, \delta x_{i-1}^j, \delta x_i^j, \delta x_{i+1}^j)$
          $\delta x_i^j = \delta x_i^j + d_k \times \delta v_i^j$
    if $\frac{t}{dt} \equiv 0 \; [n_{ortho}]$
      Gram_Schmidt $\begin{pmatrix} \delta x^1 & \ldots & \delta x^N \\ \delta v^1 & \ldots & \delta v^N \end{pmatrix}$
      if $t > t_2$ {phase 3} then
        update the N_{lyap} Lyapunov's exponents
  do $k = 0, 3$
    {Evolution of the system}
    dovect $i \in \{0, \ldots, H-1\}$
      $v_i = v_i + c_k \times f_k(x_{i-1}, x_i, x_{i+1})$
      $x_i = x_i + d_k \times v_i$

Here, H represents the number of particles; N_{lyap}, represents the number of Lyapunov's exponents; c_k and d_k are coefficients of McLachlan-Atela's algorithm [9]; f and g are functions obtained from the equations of motion [11] and their derivated forms.

Note that in practical, $t_3 \gg (t_1 \,\&\, t_2)$, i.e. the main time is spent on phase 3. In addition, as the time for updating Lyapunov's exponents is negligible, phase 2 and phase 3 can be considered identical. Consequently, we will focus our attention on phase 2 only.

2.2. The task graph

A task graph is an oriented graph, where vertices represent tasks, and edges represent dependences. Because the task graph of our application is rather complex , we present in Figure 1 a simplified (see [11] for further details) and reduced version of it.

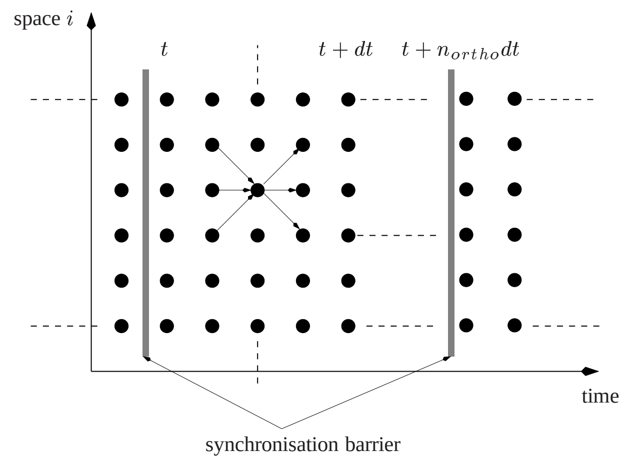

Figure 1. Reduced tasks graph of our program: graphs of tangent space and main system calculations have been superposed. Reduced dependences are then $(1,1)^t$, $(1,0)^t$ et $(1,-1)^t$. To avoid the use of an additional temporary array, the four phases of the perturbations evaluation for time $t+dt$ that depends on the value of the system at time t are considered as atomic.

3. The parallelization: the strategies

3.1. Tiling the iteration space

Rather than providing a detailed motivation for tiling, we refer the reader to the papers by Calland, Dongarra, and Robert [2] and by Högsted, Carter and Ferrante [8], which provide a review of the existing literature (see also [11]). Most of the work amounts to partitioning the iteration space of a uniform loop nest into tiles whose shape and size are optimized according to some criteria (such as the communication-to-computation ratio). Once the tile shape and size are defined, it remains to distribute the tiles to physical processors and to compute the final scheduling.

In our case, since the Gram-schmidt ortho-normalization can be seen as a synchronization barrier, the domain to be tiled is a wide strip (usually $n_{orth} \lesssim 25$) of width $4 \times n_{orth}$. Considering that $h = H/P$ might be large compared to n_{ortho}, the domain should be partitioned into parallelogram-shaped tiles. Those super-tiles should be sub-tiled into 2 triangles and parallelograms (see Fig. 2). As motivated in [11], this method is not appropriate. We propose an alternative solution, using dummy-tasks technique, which is still asymptotically optimal. The next paragraphs are devoted to the description of those two solutions.

3.2. The parallelogram solution

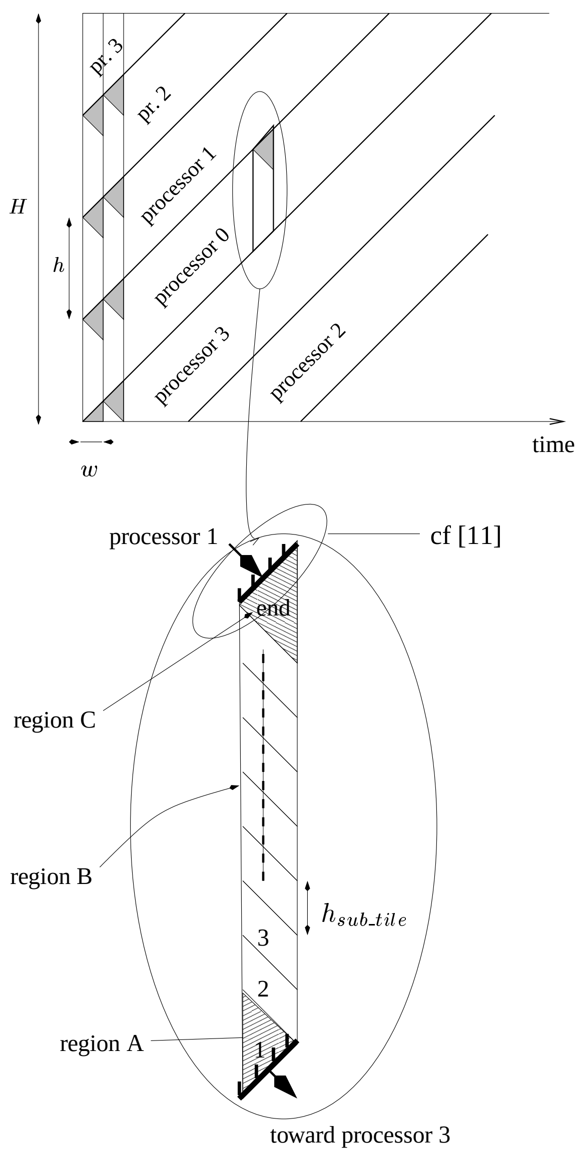

Figure 2. The parallelogram solution.

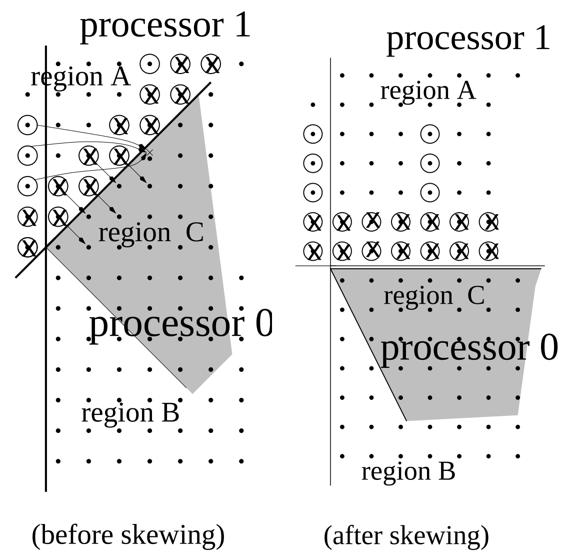

Figure 3. Part of the task graph of figure 2 before and after skewing of the iteration space. The data, necessary for the computation of region C by processor i, are represented with a circle for the main system (x_i & v_i) and with a cross for the perturbations (δx_i^j & δv_i^j).

Our parallelogram solution is illustrated in Figure 2. The iteration space is partitioned into strips along the diagonal direction. Then strips are distributed cyclically over the processors. Hence, between each ortho-normalization, each processor takes care of parallelogram-shaped tile. The main problem with this solution is, the complexity of communicated data set. Indeed, the presence of conditional branches inside the loop, partially due to borders effect, would imply a non-negligible overhead.

3.3. Second approach: rectangular tiling

As explained in Fig. 4, the iteration space is partitioned into P horizontal strips that are distributed over the P processors. Hence, between each ortho-normalization, each processor takes care of a rectangle. Because of dependences the computation of each rectangle requires the results from the computation of two triangles on the top and on the bottom. The choice here consists on computing those triangles by the processor (say i) itself. That corresponds to do *redundant work* ($w(w-1)$ more tasks for each tile) because those two triangles are also computed by the neighbored processors on top (processor $i+1$) and on bottom (processor $i-1$). Hence, before each step of computation, each processor should get from its neighboring processors the data (a vector on the top and on the bottom) useful for its computation. Then, parallelepiped-shaped tile can be tiled itself into a triangle and parallelogram-shaped sub-tiles of height h_{sous_tuile} to ensure locality. Sub-tiles are executed from the bottom to the top.

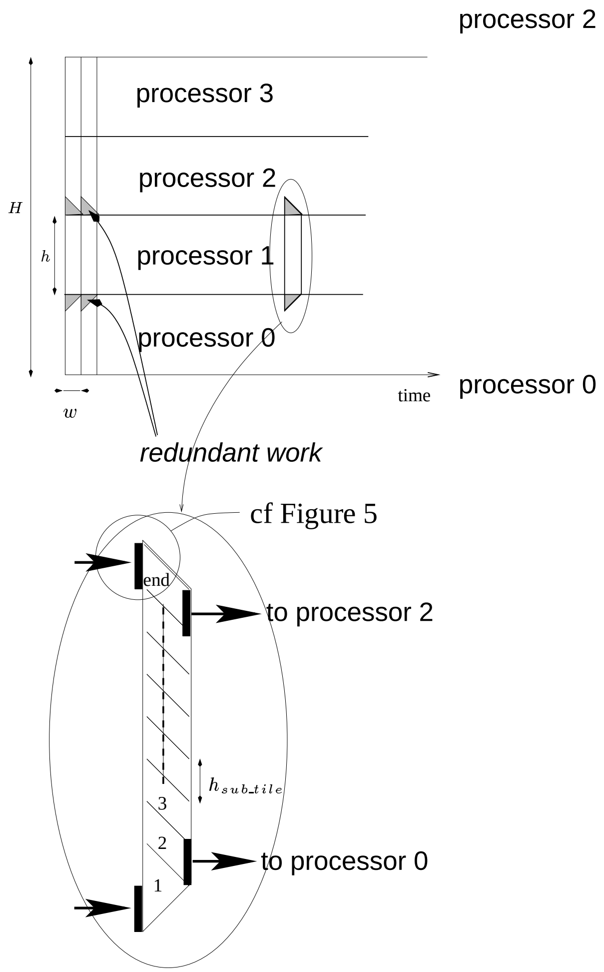

Figure 4. Rectangular tiling solution.

4. Optimal parameters

4.1. Draft of a fuzzy methodology

In the existing literature, most of the work amounts to partitioning the iteration space of a uniform loop nest into tiles whose shape and size are optimized according to some criteria (such as the communication-to-computation ratio). Shape is usually constrained by the set of dependences, and size is determined to minimize the latency implied by a coarse grain computation. Classical approaches aim to solve analytically this optimization problem entirely all at once. Instead, we propose to consider the different criteria separately and treat them in decreasing importance order:

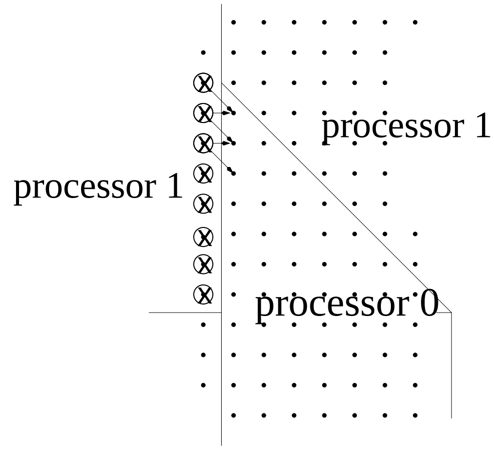

Figure 5. Part of the task graph of Figure 4. The data necessary for the computation are represented with a circle for the main system (x_i & v_i), and with a cross for the perturbations (δx_i^j & δv_i^j).

1. **Sequential execution time**: even on a sequential program, data reuse or absence of conditional branches can have a *huge* impact on the performances. Hence our first goal is to optimize the critical computational kernel performed by each processor. In this context: for data based computation, tiling should be performed at least across one direction; if possible, rectangular tiling should be preferred to trapezoidal ones: borders effects are minimized, loop bounds are simple.
As an example, in our implementation, we fixed tiles width to be greater than 20, and sub-tiles height to be lower than 100. This lead to a speedup over than 4.

2. **Express parallelism**: too large tiles can involve subsequent latencies, and tiling could kill the parallelism.

   - Many applications contain recurrences of length 1 over each direction. In this context, tiles should not be too large and then distributed cyclically or in a more sophisticated way [4] over the processors.
   - Rescheduling tasks inside tiles in addition to pipelined communications can sometimes restore the parallelism [12].
   - Finally, introducing redundant tasks as it is done here can also restore the parallelism and is sometimes better than skewing the iteration space.

3. **Communication overhead**: even for perfectly synchronized applications, communication overhead can be critical. It is not the case of our application here

where, in practical cases, H is large enough so that communication time becomes negligible compared to computation time. However, in many applications, communication volume per computation volume decreases with the size of the tiles [13]. Moreover, it is usually reported that for small-scale message passing, the communication latency is a linear function of the size of the messages. This is not true on the parallel support we used, but we can suppose that on some well configured distributed platforms, it might be better to send one time $2l$ bytes instead of two times l bytes (because of the startup). Then, coarse granularity should lead to better performances.

We call our approach a "fuzzy methodology" because the goal of each phase is not to minimize an analytical formula and to provide an exact result ("tiles size should be 3.567×0.45"). Instead, we need a result that is less constraint-full as possible. This idea is well illustrated by Figure 9.

4.2. The parallel support

The platform is made of 2 Piles of PC interconnected with a Myrinet network namely PoPC and Pom. Those parallel resources are part of several other platforms from the Laboratory for High Performance Computing (see [7, 11] for more details). *The measurements presented in this section have been performed on Pom.*

4.3. The cache effect

The goal of this section is to show the importance of tiling on mono-processor, and to evaluate the optimal height for sub-tiles (h_{sub_tile}). For this purpose, we have performed performance measurements on a rectangular iteration space of size $100 \times H$, executed on a mono-processor without any ortho-normalization phase. As one can see on Figure 6, execution on a non-tiled program shows two levels of cache: there are three steps $H < H_{L1} = 300$, $500 < H < 3000$ and $H > 5000$ corresponding respectively to the cache L1, the cache L2 and the memory.

Hence, for the rest of the experiments, in the case $\frac{H}{P} > 200$, *the iteration space will be automatically tiled with tiles of size* $w_{sub\_tile} \times h_{sub\_tile} = w \times 100$.

We then performed tiling of our sequential program with parallelogram tiles of size $w \times h_{sub\_tile}$: we can remark a slight overhead (for $H > 3000$) corresponding to the initial load of data in the memory at the beginning of each tile.

4.4. Cache reuse versus redundant tasks

We have seen the impact of tiling on the computation time. Because of the size of cache L1, a good value for

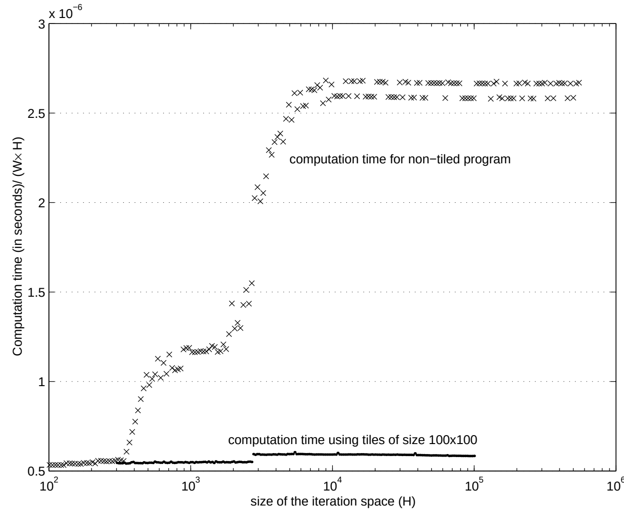

Figure 6. Time in seconds to perform one computational unit. For domains of size $H > 5000$, the speed-up is more than 4.3.

the tiles height is $h_{sub\_tile} = 100$. Experiments presented in this section show that a simple model (computation time=calculation time + memory access) provides a good approximation. However, we will see in the next section that the impact of communications per column is smaller than 200 times the average execution of one task. More formally,

$$\frac{\text{communication time for one column}}{\text{average computation time of one column}} < \frac{200}{h} = \frac{200P}{H}$$

Consequently, for domain size $H \geq 4000P$, communications will be negligible (5 %).

Hence, for locality purpose, tiles should be as large as possible. However, as we increase w, the amount of dummy tasks ($w \times (w-1)$) are increased and one should find the best trade-off between providing locality and computing less dummy tasks. Let us formalize this idea.

Let us denote by τ_{L2} (respectively τ_{Mem}) the average time necessary to fetch a data from cache L2 to cache L1 (respectively from the memory to cache L2); let us also denote by τ_{calc} the average calculation time of one task. Then, the computation time of a tile (rectangle of size $h \times w$ plus two triangles of height w) is given by the formula (where $\delta_{h>3000}$ is 1 if $h > 3000$ and 0 otherwise).

$$T_{calc}(h,w) \simeq hw \times \left((\tau_{L2} + \delta_{h>3000}\tau_{Mem}) \times \left(\frac{1}{w} + \frac{2}{h}\right) + \left(1 + \frac{1}{h}(w-1)\right)\tau_{calc} \right)$$

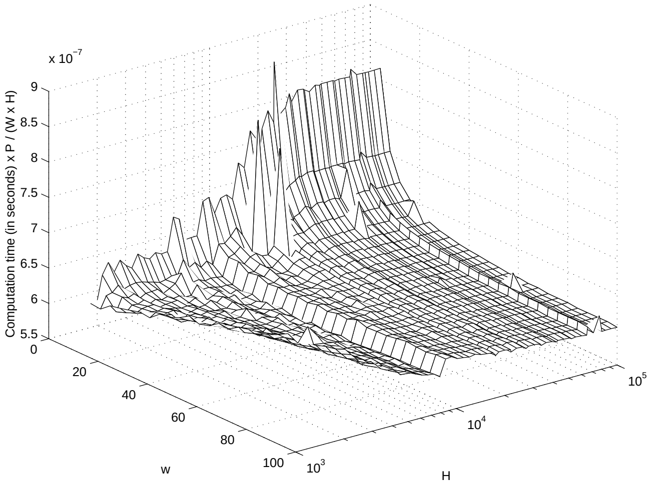

Figure 7. Influence of the width w on the total computation time (without any communications) versus the domain size H for parallel program executed on three processors.

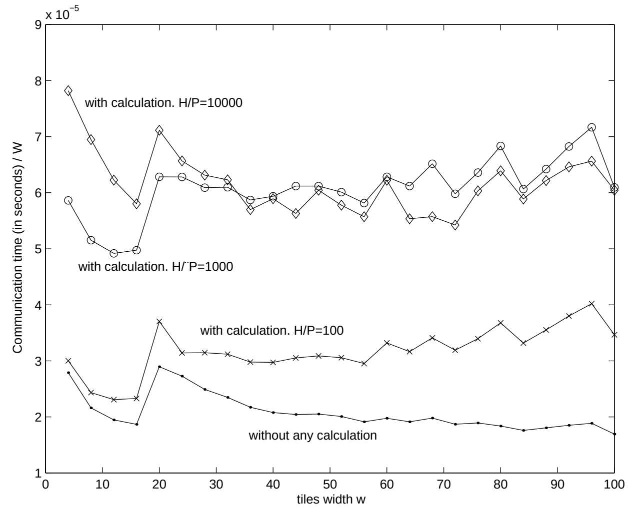

Figure 8. Evaluation of the communication overhead.

That can be experimentally observed on Figure 7 for values $w \gtrsim 15$. For $w \lesssim 15$, a more precise model of memory access than the simple formula $(\tau_{L2} + \delta_{h>3000}\tau_{Mem})$ would be necessary. But, as we will see on the next section, for h values for which w should be chosen as small, the communication overhead has a non-negligible impact. In this context, it becomes hazardous to formulate analytically the average execution time.

4.5. The communication overhead

In this paragraph, we study the impact of communications on the computation time. Because our study deals with small-scale message passing, we might use a simple model [10] for communication made of two components: the start-up β, and the communication itself $l\tau$ proportional to its size l. Usually, $\beta \gg \tau$, which motivates tiling because it increases the granularity of computation. Here, for a domain of size $W = 4T/dt$, tiled with tiles of size $w \times h = w \times H/P$, there are W/w groups of communications of size proportional to w. If we normalize by W, we obtain a communication cost per column of $\frac{\beta}{w} + \tau$. However, Fig. 8 do not confirm this approach: we see that communication latency is not a linear function of the message length. Also it increases when performed between data based computations. Further curves and discussions are provided in [11]:

4.6. The optimal parameters

From the previous remarks, we can summarize the general behavior of the program to the following points:

- For a small domain size $\frac{H}{P}$, the "redundant-work" overhead is very significant. Hence, w should be smaller than 20. In that context, larger the tiles, smaller the communication overhead, faster the calculation time (because of locality), and larger the redundant work overhead. Here, communication time and calculation time are predominant. The best tiles width is then $w = 16$.

- For very large domains, either the "redundant-work" overhead and the communication overhead are negligible. Hence, the best tiles width is $w = 4n_{ortho} = 100$.

- Between these two extreme cases, it is difficult to define a formal solution because of the irregularity of the communication overhead; the best solution might consist on testing all the possibilities (only 25 possibilities for w) with fixed $W = 4\frac{T}{dt} = 500$. Figure 9 reports the results. If we denote by $t_{calc}(H, w)$ the normalized computation time. Also, if for a given value of domain height H, we denote by w_{opt} the optimal tile's width: $t_{calc}(H, w_{opt}) = \min_w t_{calc}(H, w)$. Then, by allowing an error of 2%, the minimum bound corresponds to $\min\{w, t_{calc}(H, w) < t_{calc}(H, w_{opt}) \times 1.02\}$ and the maximum bound to $\max\{w, t_{calc}(H, w) < t_{calc}(H, w_{opt}) \times 1.02\}$.

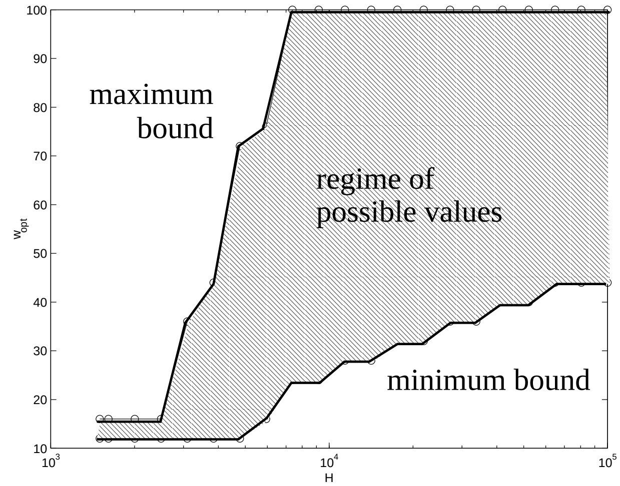

Figure 9. Optimal values of w versus H on $P = 6$ processors.

5. Performance results

In this section, we present performance results obtained on the clusters described in Section 4.2. Unfortunately, because no reservation system is available on this machine, it is extremely difficult to get all the processors together in a dedicated mode. Hence, we had to dynamically balance the loads on processors during the execution: the height h_j of the tile computed by processor P_j was chosen to verify the constraint $\forall j, T_{calc}(h_j, w, P_j) \simeq Constant$. The imperfection of the system (miss of reactivity) explains the irregularity observed on the curves. Moreover, because we were interested mainly in large domain size, we have restricted the experiments to a tiles' width of $w = 100$. Instead, smaller values would have provided better results on small iteration domains (cf Figure 9). However, results are still very good.

14 processors from Pom with an average cycle time $t_{seq}(pom) = 3.99 \times 10^{-7}$ and 11 processors from PoPC with an average cycle time $t_{seq}(popc) = 7.09 \times 10^{-7}$ have been used. In that case, nor the processors, neither the network are homogeneous. Hence, the work should be load balanced. In the other hand, we aim at executing the program on very large domains, and the communication overhead between two clusters is not large enough to justify taking into account the heterogeneity of the network. On a meta-computing context, partitioning should have be done hierarchically on each cluster (see [11] for a discussion), but here we chose to implement our program just as if the network was homogeneous.

Two curves are represented in Figure 10:

- The *average execution time per task* corresponds to the total execution time divided by the iteration domain size $W \times H = 500 \times H$.
- the *extrapolated optimal execution time* (horizontal line) is evaluated as follow: the minimum (over all possible domain size) average sequential execution time per tile is measured on one node from the Pom (denoted by $t_{seq}(pom)$), and on one node from the PoPC (denoted by $t_{seq}(popc)$). The average sequential time is then extrapolated by the formula

$$\frac{1}{\frac{14}{t_{seq}(pom)} + \frac{11}{t_{seq}(popc)}} \simeq 1.97 \times 10^{-8}.$$

Figure 11 represents the speedup for our parallel execution versus the extrapolated sequential execution.

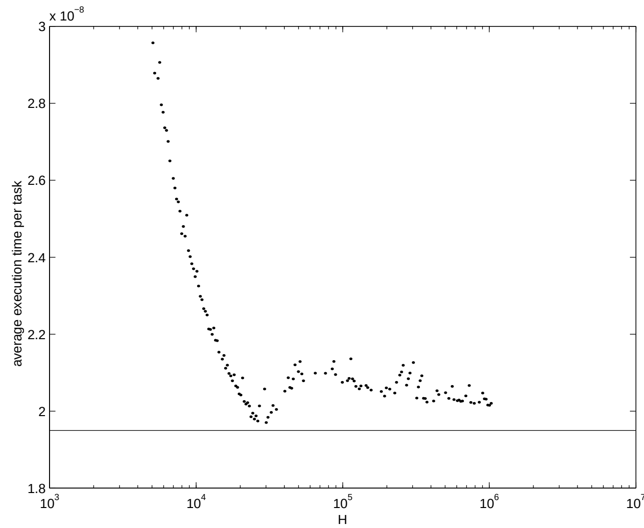

Figure 10. Execution time on 25 processors. The horizontal line represents the extrapolated optimal execution time as defined in the text.

The execution time curve (Figure 10) contains a step (around $H = 3000$) that corresponds to the the cache limit: because sequential execution time curve is roughly identical but scaled by a factor of $P = 25$, the ratio between sequential execution time and parallel one represented in Figure 10 contains a bump (that exceed the "optimal" value 25).

6. Conclusion

In this paper, we have applied tiling techniques to a practical problem of dynamical system theory, namely the numerical Lyapunov calculation. This work gave the opportunity to check the veracity of models often used for tiling optimizations. In particular communication cost models that

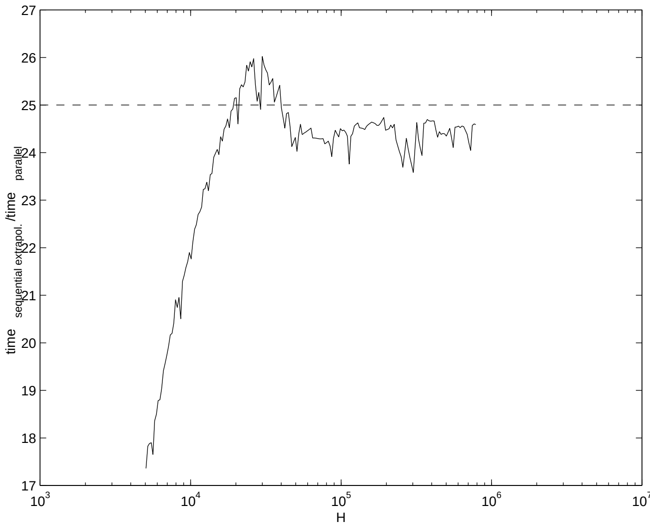

Figure 11. Speedup for our parallel execution on 25 processors versus the extrapolated sequential execution.

are a simple function of the communicated data volume l (like $\beta + l\tau$) have been shown inaccurate. The major objective of this article was to present a simple-to-implement yet very efficient solution for tiling the iteration space (cyclic and non-cyclic) of explicit integration schemes of differential equations. This solution can be applied on all algorithms that present a recurrence on all but one directions, which is the case, in particular, of some SOR and DSP algorithms. The optimization of the tile size has been decomposed into different successive phases, sorted in decreasing importance order, and which have been studied in this article. Our approach differs from classical ones that usually aim to solve tiling optimization problem in a single step. We strongly believe that our "fuzzy" approach should be used by future automatic partitioning algorithms. Finally, except maybe for the orthonormalization which is quite specific to our physical study, our implementation is scalable and well-suited for massively parallel platforms.

6.1. Acknowledgments

The authors thank Marie Rastello, Stefano Ruffo and Alessandro Torcini for very helpful discussions, Hervé Gilquin for its very efficient assistance when using the Sun-Enterprise of PSMN [5]. Finally, the authors would also like to thanks Michael Winter for his work on parallelepiped-shaped tiles.

References

[1] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? *Integration, the VLSI Journal*, 17:33–51, 1994.

[2] P. Calland, J. Dongarra, and Y. Robert. Tiling with limited resources. In L. Thiele, J. Fortes, K. Vissers, V. Taylor, T. Noll, and J. Teich, editors, *Application Specific Systems, Achitectures, and Processors, ASAP'97*, pages 229–238. IEEE Computer Society Press, 1997. Extended version available on the WEB at http://www.ens-lyon.fr/∼yrobert.

[3] P.-Y. Calland and T. Risset. Precise tiling for uniform loop nests. In P. Cappello et al., editors, *Application Specific Array Processors ASAP 95*, pages 330–337. IEEE Computer Society Press, 1995.

[4] D. Chavarra-Miranda, A. Darte, R. Fowler, and J. Mellor-Crummey. Generalized multipartitioning. In *Second Annual Los Alamos Computer Science Institute (LACSI) Sy mposisum*, Santa Fe, NM, Oct. 2001.

[5] P. S. de Modélisation Numérique. Psmn. World Wide Web document, URL: `http://psmn.ens-lyon.fr`.

[6] F. Desprez, J. Dongarra, F. Rastello, and Y. Robert. Determining the idle time of a tiling: new results. *Journal of Information Science and Engineering*, 14:167–190, 1998.

[7] L. for High Performance Computing. Lhpc. World Wide Web document, URL: `http://www.ens-lyon.fr/LHPC/ANGLAIS/choix.html`.

[8] K. Högstedt, L. Carter, and J. Ferrante. Determining the idle time of a tiling. In *Principles of Programming Languages*, pages 160–173. ACM Press, 1997. Extended version available as Technical Report UCSD-CS96-489, and on the WEB at http://www.cse.ucsd.edu/∼carter.

[9] R. I. MacLachlan and P. Atela. The accuracy of symplectic integrators. *Nonlinearity*, (5):541–562, 1992.

[10] L. Prylli, B. Tourancheau, and R. Westrelin. Modeling of a high speed network to maximize throughput performance: the experience of bip over myrinet. In H. R. Arabnia, editor, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, volume II, pages 341–349. CSREA Press.

[11] F. Rastello and T. Dauxois. Parallelization of the numerical lyapunov calculation for the fermi-pasta-ulam chain. Technical Report RR-01-42, LIP, ENS Lyon, France, November 2001. Available at `www.ens-lyon.fr/LIP/`.

[12] F. Rastello, A. Rao, and S. Pande. Optimal task scheduling to minimize inter-tile latencies. In *International Conference on Parallel Processing (ICPP'98)*, pages 172–179. IEEE Computer Society Press, 1998.

[13] F. Rastello and Y. Robert. Automatic partitioning of parallel loops with parallelepiped-shaped tiles. *IEEE Transaction on Parallel Distributed Systems*, 2001. to appear.