©Saxe-Coburg Publications, 2013.
F. Magoulès, (Editor),
Patterns for Parallel Programming on GPUs
Saxe-Coburg Publications, Stirlingshire, Scotland, 111-149.

Chapter 5

Optimization methodology for Parallel Programming of Homogeneous or Hybrid Clusters

S. Vialle[1,3] and S. Contassot-Vivier[2,3]

[1]*Supelec & UMI GT-CNRS 2958, Metz, France*
[2]*Université Lorraine, Loria, UMR 7503, Nancy, France*
[3]*AlGorille INRIA Project Team, Nancy, France*

Abstract

This chapter proposes a study of the optimization process of parallel applications to be run on modern architectures (multi-core CPU nodes with GPUs). Different optimization schemes are proposed for overlapping computations with communications, and for computation kernels.

Development methodologies are introduced to obtain different optimization degrees and specific criteria are defined to help developers find the most suitable degree of optimization according to the considered application and parallel system. According to our experience in industrial collaborations, we analyze both performance and code complexity increase. This last point is an important issue, especially in the industry, as it directly impacts development and maintenance costs.

Complete experiments are performed to evaluate the different variants of a benchmark application that consists of a dense matrix product. In those experiments, different runtime parameters and cluster configurations are tested. Then, the results are analyzed to evaluate the interest of the different optimization degrees as well as to validate the interest of the proposed optimization methodology.

Keywords: message passing, multithreading on multicore, vectorization on GPU, communication-computation overlapping, computing kernel optimization, deployment.

5.1 Motivations and Objectives

During recent decades, parallel computing has known a great development. Great improvements have been made on the software side (efficient standard parallel libraries for communications [1] or thread management [2]) and on the hardware side (increase in the number of cores, development of new devices like GPUs).

However, with the emergence of new types of parallel architecture, whose complexity has increased with the levels of explicit hierarchies, and the never-ending demand for efficiency by users (for intensive computations like physical simulations and so on), computer scientists are still faced with the challenge of optimally exploiting the power of the latest systems.

According to our past experiences in parallel design and developments, and the numerous traps we have observed, we propose in this chapter a didactic study of the design and implementation of a common scientific application. Our case-study deals with the very classical matrix product. Our objective is to detail the main choices a developer would have to face for the design, implementation and optimal use of such an application on a modern cluster.

5.1.1 Programming Modern Distributed and Parallel Architectures

Modern parallel architectures are mainly clusters of complex and powerful nodes, typically multicore CPUs sometimes enhanced with hardware accelerators like GPUs (often denoted as hybrid nodes). Although these architectures are cheap, they can lead to very high performances. This is why they are extensively used in large parallel systems. However, they include two or three different parallelism grains that require as much parallel programming paradigms. For example, we can implement parallel algorithms using:

- MPI alone, deploying (approximately) one MPI process per CPU core,
- MPI with OpenMP, deploying at least one MPI process per node and several OpenMP threads per MPI process,
- MPI with CUDA, to program a cluster of GPUs, deploying at least one MPI process per node, and running some grids of CUDA threads on GPUs from the MPI processes,
- MPI with OpenMP and CUDA, to program a cluster of nodes including both multicore CPUs and GPUs, deploying at least one MPI process per node, some CPU threads per MPI process, and running grids of GPU threads from one or several CPU threads.

The last configuration is the most complex one but it allows for the implementation of codes that can run on both CPU and GPU cores of each node. Moreover, it enables

the overlapping of CPU computations, GPU computations, CPU/GPU data transfers and inter-node communications. However, the cost of such advantages is a higher complexity to develop, debug and optimize a code including all these features. Finally, when running on some benchmark data sets, it can be very hard to validate and certify this type of codes. In the same way, the higher design, development and maintenance cost of such codes sometimes may be prohibitive in comparison to the gains obtained relative to a more simple (and thus cheaper) parallel software.

We propose in this chapter a methodology, composed of basic and generic development rules and implementation examples, to ease development of efficient multi-paradigm and multi-grain parallel codes on multicore CPU and manycore GPU clusters.

5.1.2 Benchmark Application

In order to ease the didactic description of our approach, we have chosen the very classical application of dense matrices product.

We denote A and B as two real square matrices of size $n \times n$ and we want to compute $C = A \times B$ on a parallel system containing P nodes. For this purpose, we adopt a classical algorithm on a ring topology of the nodes by distributing vertical strips of the B and C matrices (whose widths are n/P columns) over the nodes. In the same way, the A matrix is decomposed in horizontal strips (whose heights are n/P lines) that are initially distributed over the nodes. Then, on each node of the system, the local square sub-matrix of C (of size $\frac{n}{P} \times \frac{n}{P}$) corresponding to the local strips of A and B that are owned at that time is computed. Once this is done, the horizontal strips of A are cyclically shifted from one node to the following one in the ring, using MPI communications. Then, the local computations of other sub-matrices of C are done and so on until all the sub-matrices of C are computed. It can be deduced easily from the size of C ($n \times n$) and the number of nodes in the system (P), that P local multiplication and communication steps will be necessary to perform the whole computation of C and to return A in its initial state.

For clarity's sake, the initial distribution of the matrices is given in Figure 5.1, and the first four steps of the algorithmic process are illustrated in Figure 5.2.

In addition to that data distribution, the local part of matrix B on each node is transposed in order to optimize the memory accesses a little bit by having the same line size to parse between A and BT during the product. So, we obtain the basic algorithmic scheme, involving only CPU computations and synchronous communications, given in Algorithm 1.

In this algorithm, the communications are not explicitly described because there are several ways to implement such an operation, even in a synchronous/blocking mode. For example, with the MPI library, this can be efficiently achieved by the function `MPI_Sendrecv_replace`.

In fact, this point is one of the key issues in the optimization process of a parallel application. One aspect of the scope of this chapter is to look for the best option among the different possibilities either at the design level or at the implementation

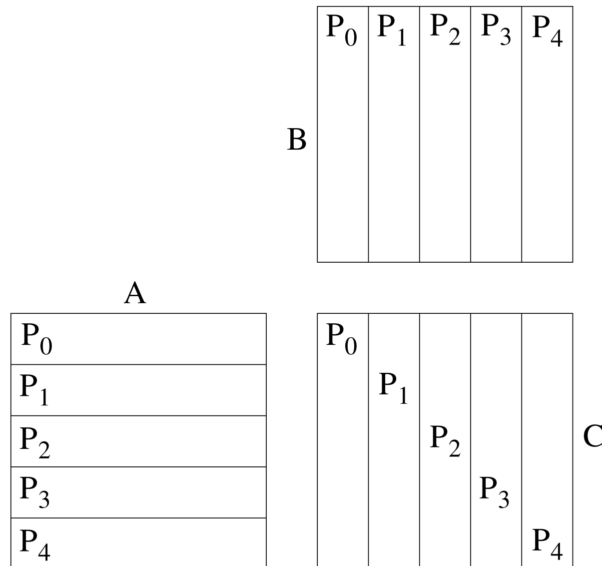

Figure 5.1: Initial distribution of the matrices over five nodes

one. Before exploring the ways the communications can be performed in Section 5.2, we briefly describe in the next subsection the parallel system that has been used to obtain all the experimental results that are reported within the chapter.

5.1.3 Experimental Context

The parallel system that has been used for the entire set of experiments presented in this chapter is a cluster composed of 16 nodes each including an Intel Nehalem quad core at 2.67Ghz, 6 Gb RAM and a NVIDIA GeForce GTX480 GPU. The interconnection network is a Gigabit Ethernet with a DELL Power Object 5324 switch.

Concerning the software environment, the OS is a Linux Fedora 64bits. The C compiler is the GNU C version 4.5.1 and the CUDA version is 4.2.

5.2 Interest and Difficulties of Computations and Communications Overlapping

The problem of overlapping computations and communications in parallel applications has been extensively studied in the last two decades, see for example [3–5,11,16], and is still an active research topic [6, 7, 9, 10, 14].

The obvious advantage of such optimization, when it can be ideally realized, is to completely hide one of the two actions (computations or communications) behind the other one in terms of execution time. A simple example is given in Figure 5.3 where each box corresponds to the computation of its label.

Nevertheless, the seizure of such gain often requires important modifications in the parallel algorithm and most often the overlapping is not complete but only partial.

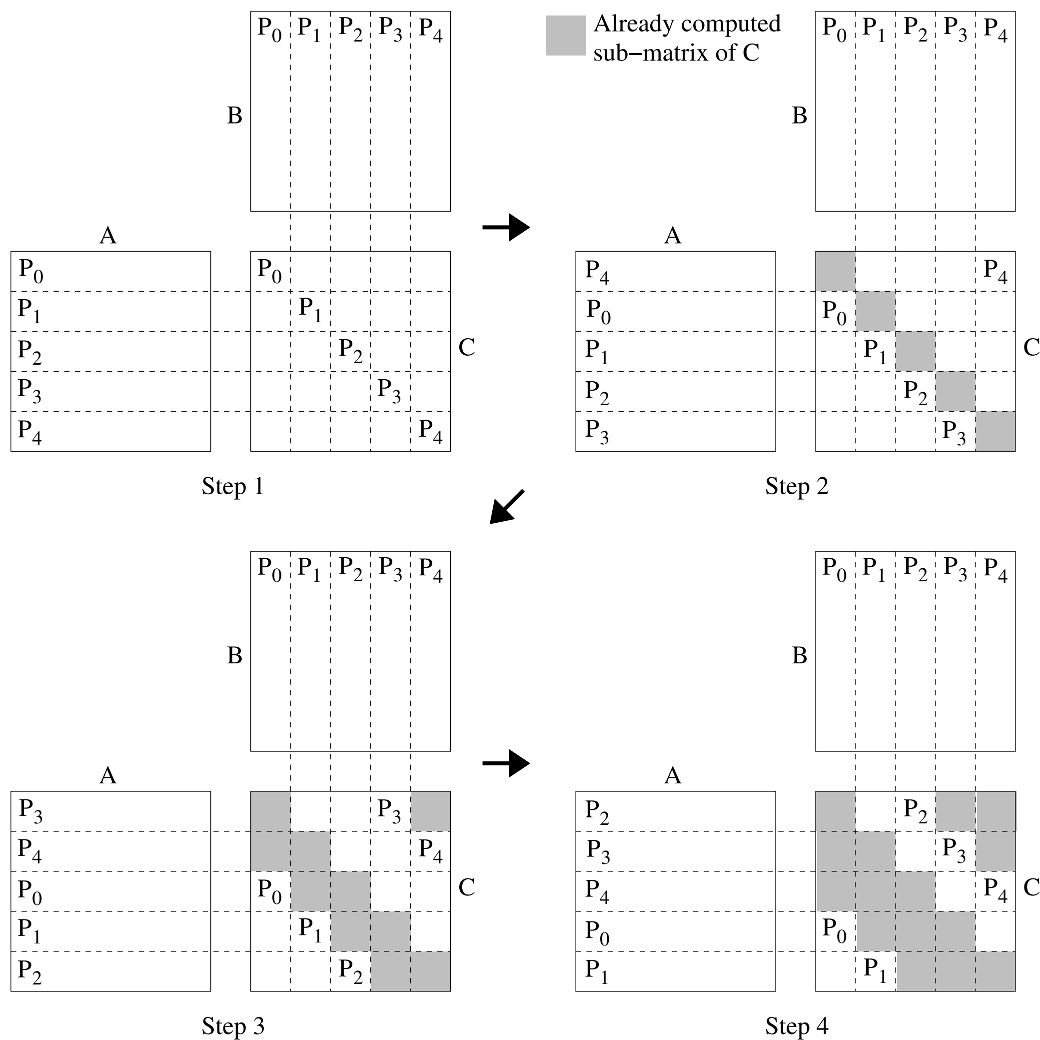

Figure 5.2: First four steps of the parallel process for 5 nodes

In some cases it is not even pertinent to try to overlap because the gains are much smaller than the effort of design and implementation. So, the first step when studying a potential overlapping of computations and communications is to check if it is worth doing it! This is what is discussed in the following paragraph.

5.2.1 Decision Criteria to Implement Overlapping

Ideally, when working on the development of a parallel application, one may want to obtain its maximal optimization in order to obtain the smallest execution time. However, in practice this is generally not what is done, save for simple cases where optimal designs and implementation are obvious. This is due to the ratio between the required effort to add a given enhancement and its gain over the application. Although this ratio is often a secondary criteria in academic research because fundamental studies aim at exploring all the potentiality of a given parallel problem, this becomes a major

Algorithm 1: Basic parallel algorithm for matrices product

Initial data: A, BT and C arrays are distributed over the nodes as in Figure 5.1.
BT is the transposed version of the local part of B on the current node.

```
1: for all node NodeId ∈ {0, ..., P − 1} do in parallel
2:   for step = 0 to P − 1 do
3:     // Loop over the global steps as in Figure 5.2
4:     LineOffset ← n/P × ((step + NodeId)%P)
5:     for i = 0 to n/P − 1 do
6:       // Computation of a square sub-matrix within the local vertical strip of C
7:       for j = 0 to n/P − 1 do
8:         val ← 0
9:         for k = 0 to n − 1 do
10:          val ← val + A[i][k] × BT[j][k]
11:        end for
12:        c[i + LineOffset][j] ← val
13:      end for
14:    end for
15:    Synchronous communications for cyclically shifting the strips of A
       over the nodes by one position to the right
16:    // The result of this operation is that:
17:    //  - the current local strip of A is sent to node (NodeId + 1)%P
18:    //  - the new local strip of A is received from node (P + NodeId − 1)%P
19:  end for
20: end for all
```

element in the industrial context. The main difference between these two contexts comes from the fact that the industry is directly linked to economic constraints. So, the difficulty to design, implement and maintain an application has an important impact over the cost of the application (conception time, number of people required and their competence level).

So, before bringing any improvement to an application, its level of pertinence must be measured. Moreover, the common approach (of good sense) consists of bringing the improvements with the highest gains in first and then following that with the improvements of decreasing gains. So, the improvements are sorted in decreasing order of their respective gains.

Now, let us define the optimization ratio of an application as 0% corresponding to no optimization (no improvement done) and 100% corresponding to the most efficient version of the application that can be made (all the possible improvements are done). According to our experience, we have been able to observe that when someone brings a series of improvements to an application by following their decreasing gain order, the difficulty tends to increase exponentially with the optimization ratio. Moreover, since the improvements are performed in decreasing order of their gains, the gain curve tends to slow down when increasing the optimization ratio. These facts are empirically illustrated in Figure 5.4.

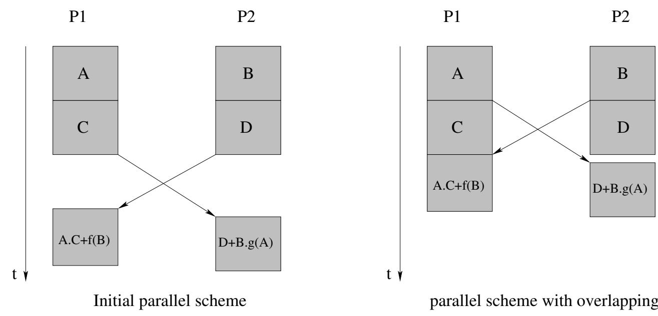

Figure 5.3: Simple case of overlapping of computations and communications

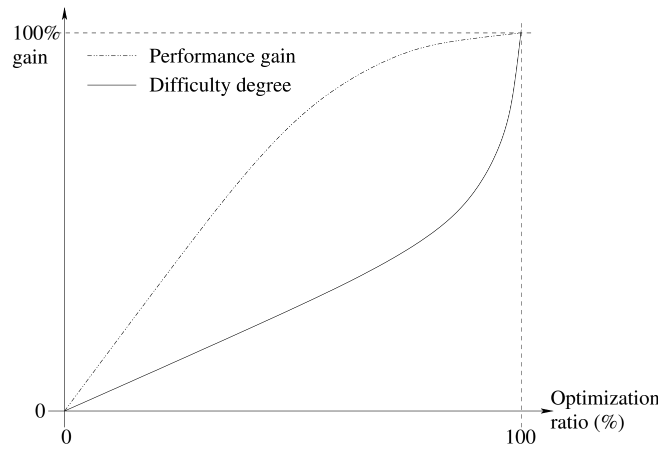

Figure 5.4: Difficulty degree and performance gain evolution in function of the optimization ratio

So, a decision criterion for bringing improvements to an application can be deduced by finding a specific threshold inside the optimization ratio interval where a satisfying trade-off between the difficulty degree and the overall gain is achieved. Another way to get a decision criterion is to consider each improvement separately and to compare its ratio between its estimated difficulty degree and its potential gain with a given threshold.

However, those criteria require achievement of a good idea of the difficulty and gain curves. Although it is quite subjective to evaluate the difficulty curve, the gain curve can be obtained quite easily by monitoring the time consumption of every part of the application and by ordering them in decreasing order.

In a parallel application, we can distinguish two main types of time consumption. The former is related to the computations and the latter concerns the communications. Concerning the computations, the monitoring generally consists of determining in which functions of the program the majority of the computation time is passed. This allows the designers/programmers to focus their optimization/parallelization effort on the most time consuming parts of the application. In the same way, the monitoring of the communications provides the time passed in each communication phase and this information directly influences the use of an overlapping technique.

Such an improvement will depend initially on external parameters like the available time/budget to design and implement such optimization and the importance degree of the application performances. Those criteria have to be considered before any analysis of the pertinence of including overlapping in the application. Then, parameters related to the development and exploitation contexts of the application will play a role during the analysis. We can cite for example the software environment, the parallel system architecture and the number of available nodes for production.

Once the external parameters have been considered, the pertinence of the overlapping should be evaluated according to two criteria. The first one checks that the measure of the maximumpotential gain of the overlapping is large enough. That measure is obtained by selecting the minimum time consumption between computations and communications and by computing the ratio of this minimum to the total execution time of the application. The prior selection of the minimum comes from the fact that an ideal overlapping will at best completely *hide* (overlap) the shorter time consuming activity inside the other one. Thus, the time of that hidden activity will be removed from the total execution time. So, the potential gain is directly linked to the ratio between the computation and communication times. Its maximum value is reached when those two activities take approximately the same time. This is illustrated in Figure 5.5 where we consider, for clarity's sake, that there is no additional cost in the application other than the parallel computations and the communications. In this theoretical case, the maximum potential overlapping is 50% when both activities take the same time as their overlapping makes the application twice as fast (50% shorter). In real applications, the overall behaviour will be similar to the one depicted in this figure except that the maximum potential of overlapping will be under 50% due to constant additional costs in the application (initialization, post-treatments, termination, *etc.*).

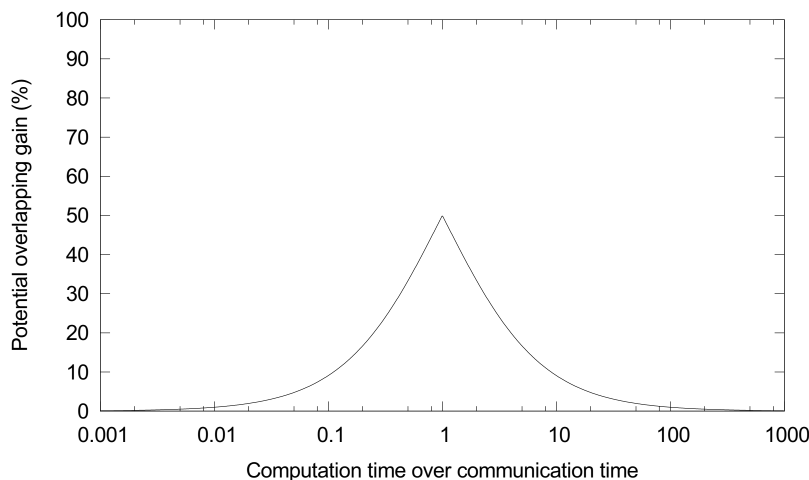

Figure 5.5: Potential overlapping gain according to the ratio between the computation time and the communication time in an application with no other costs

For example, if we consider an application where the computation and communication times are equal and correspond to 40% of the total execution each. Then, although

the potential gain will be maximal, it will be limited to 40% of the initial execution time. Now, let us imagine that the computation time of the application is 85% of its total execution time, and the communication time is only 1%. Then, although any ideal overlapping (if it exists) would completely hide the communications, it would obtain a maximal gain of only 1% of the total execution time. This is a very small gain according to the design/implementation effort required and the complexity increase of the source code. In such a case, it is very likely that the overlapping would be considered not pertinent.

In fact, that first criterion is fulfilled when the potential gain becomes large enough according to a given threshold, corresponding to the trade-off limit between optimization and design/implementation/maintenance cost. It provides an interesting filter but it is quite coarse due to the fact that it is totally theoretical and is based upon the hypothesis that an ideal overlapping can be found. However, this is generally not possible in practice and it is very common that only a part of the parallel computations and communications can actually be overlapped, reducing the final gain.

This is why a more subtle criterion is necessary once the first filter is passed. It is important to keep in mind that this second criterion should be checked only when the first criterion is verified because it requires a deeper analysis of the application algorithms and potentially a finer monitoring. It consists of evaluating the quality of the best overlapping scheme found by analyzing the parallel scheme of the application. This quality is measured by the percentage of the shortest activity that actually can be overlapped with the other activity. It represents, in some sense, the maximal degree of overlapping that can be achieved between computations and communications. The maximal quality (100%) corresponds to the case where one of the activities is completely overlapped by the other one.

Finally, the maximal global gain that can be expected (in percentage of the total execution time) is deduced by multiplying the first ratio (used in the first criterion) with that quality measure. The second criterion is fulfilled for sufficiently large values of this global gain.

For example, let us consider that 48% of the total execution time of an application is spent for computations and 46% for the communications. The first criterion is fulfilled since the ratio of the shortest activity is 46%. However, if the best overlapping scheme that is found can only overlap 10% of the communications with the computations (quality measure), then the actual maximal gain will be only $46\% \times 10\% = 4.6\%$, and the second criterion would probably not be satisfied.

In conclusion, we can say that the first criterion is used to decide whether a design analysis of the possible overlapping schemes has to be conducted or not, and the second one is used to decide whether it is worth implementing it or not.

If we apply this methodology to our matrices product application, we have to monitor the computation time, the communication time and the total execution time in order to check the first criterion. Such measures are given in Table 5.1 for a 4096×4096 matrix size and different numbers of nodes.

First of all, it can be observed that the communication time just increases slightly with the number of nodes. This is due to the fact that only the number of communica-

Number of nodes (P)	computation time (s)	communication time (s)	total exec time (s)	ratio of the shortest activity (%)
2	40.852	1.417	44.186	3.21
4	20.460	1.612	23.828	6.77
8	10.240	1.622	13.606	11.92
16	4.943	1.720	8.412	20.44

Table 5.1: Computation, communication and total times in the basic parallel matrices product for $n = 4096$, and potential gain of the overlapping

tions increases with the number of nodes but the global volume stays constant. This is not the same for the computation time as the parallelism is quite efficient and there is almost a linear decrease with the number of nodes. The total execution time shows a slower decrease than the computation time due to the inclusion of the communications, but also to the sequential parts of the program (initialization, finalization, *etc.*) that have generally near constant costs according to the problem size.

The very different benaviours of the computation and communication times imply that the ratio of the shortest activity (the communications in this case) increases with the number of nodes and reaches very significant percentages of the total execution time. According to these results, the pertinence of the overlapping is quite obvious for sufficiently large numbers of nodes. Effectively, an ideal overlapping should provide a gain of around 12% with 8 nodes and 20% with 16 nodes. Nevertheless, although this is not observable in these experiments, it must be kept in mind that there is always a threshold over the number of nodes beyond which the computation time becomes smaller than the communication time and, as explained in the first criterion description, the potential overlapping gain decreases.

At this point, the decision of implementing the overlapping still depends on the second criteria. A short analysis of the matrices product parallel algorithm (see Algorithm 1 and Figure 5.2) reveals that the communications performed at each step concern only some input data (a strip of A). So there is no strong constraint over the start time of those communications and they can be done during the computations of the C sub-matrices without involving any perturbation between the two activities. The only interaction between them may be concurrent read accesses on the A strip, but these types of accesses are quite efficiently managed by the current hardware and they would imply a negligible loss of time. Moreover, with the global communication time being smaller than the global computation time in Table 5.1 (a finer monitoring would confirm that it is also true when comparing them within each iteration of the global loop), it seems possible in this case to obtain an almost complete overlapping of the communications with the computations. Thus, the overlapping quality is close to 100% and the maximal actual gains are very close to the theoretical gains obtained in Table 5.1 (right column). So this parallel application is very well suited to the overlapping.

In the following paragraphs, we explore different overlapping strategies for our benchmark application. The MPI communication library is used in our implemen-

tations as it is currently the standard environment for parallel developments.

5.2.2 Attempting to Use Non-Blocking MPI Communications

When trying to implement overlapping in a parallel application, the first idea that comes to mind is to use the non-blocking communications that are available in the MPI library. The key idea in such communication operations is that they do not block till the communication is actually performed but they return immediately. It is a bit like a registering system in which the communications to be done are inserted on demand and removed as soon as they are performed. However, unlike the intuitive idea that the management of those asynchronous communications would be performed in an additional thread (and thus in parallel with the main process), the MPI standard does not specify any thread use for this and in general, the communications are managed explicitly at their initial request or whenever their completion is tested (including the waiting functions) [8]. So, in this context, the overlapping of communications and computations may not be fully parallel and thus may be inefficient.

According to our benchmark application, the overlapping scheme we obtain is given in Algorithm 2. As the communications are potentially performed during the computations, two versions of the local A matrix are required on each node to avoid concurrent read-write accesses. At each step, one version is used to perform the current computations and to send the current version of A to the next node while the other one is used for the reception of the next version of A from the previous node. For convenience in managing the two versions of A, they are placed in a global array of size two, implying an additional dimension (in first position) in the local array A. The "..." in the parameters of the call to the local computations function corresponds to additional parameters that are not relevant here.

In this version, we use persistent communications instead of the simple `Issend` and `Irecv` functions because this avoids multiple creations/releases of the involved MPI requests that would take place at each iteration of the main loop (over the `step` counter).

As mentioned above, although that overlapping scheme is operational, it may not provide the best performances (see Section 5.2.5) due to the fact that non-blocking communications are not multi-threaded. In the next subsection another scheme is proposed making use of separate threads for computations and communications.

5.2.3 Synchronous MPI Communications inside Dedicated Threads

In order to avoid the problem of the asynchronous MPI communications, a second possibility to implement the overlapping consists of using separate threads for the computations and the communications.

A simple and efficient way to do that is to use the OpenMP directives in conjunction with MPI. The principle is to place communications in one OpenOMP thread and the local computations in another one (possibly several for multi-core nodes). In the case of multiple computing threads, each of them performs only a part of the local

Algorithm 2: Overlapping scheme with non-blocking MPI communications

lstlisting

```
// Me is the number of the current node
// NbPE corresponds to P
// SIZE corresponds to n
// LOCAL_SIZE corresponds to n/P

// Status and requests for the non-blocking MPI communications
MPI_Status  StatusS[2];
MPI_Status  StatusR[2];
MPI_Request RequestS[2];
MPI_Request RequestR[2];

// Creation of persistent communications if more than one node
if (NbPE > 1) {
  MPI_Ssend_init(&A[0][0][0], LOCAL_SIZE*SIZE, MPI_DOUBLE, (Me+1)%NbPE, 0, ↩
      MPI_COMM_WORLD, &RequestS[0]);
  MPI_Recv_init(&A[1][0][0], LOCAL_SIZE*SIZE, MPI_DOUBLE, (Me-1+NbPE)%NbPE, 0, ↩
      MPI_COMM_WORLD, &RequestR[1]);

  MPI_Ssend_init(&A[1][0][0], LOCAL_SIZE*SIZE, MPI_DOUBLE, (Me+1)%NbPE, 0, ↩
      MPI_COMM_WORLD, &RequestS[1]);
  MPI_Recv_init(&A[0][0][0], LOCAL_SIZE*SIZE, MPI_DOUBLE, (Me-1+NbPE)%NbPE, 0, ↩
      MPI_COMM_WORLD, &RequestR[0]);
}

// Computation and circulation loop (same as line 2 in the basic scheme)
for (int step = 0; step < NbPE; step++) {

  // Index of the local version of A to use for the computations and sending
  int CurrentIdx = step%2;
  // Index of the local version of A to use for the reception
  int NextIdx = 1 - CurrentIdx;

  // Communications
  if (NbPE > 1) {
    MPI_Start(&RequestR[NextIdx]);
    MPI_Start(&RequestS[CurrentIdx]);
  }

  // Local computations
  KernelLocalProduct(step, CurrentIdx, ...);

  // Waiting for communications completion before going to next step
  if (NbPE > 1) {
    MPI_Wait(&RequestR[NextIdx], &StatusR[NextIdx]);
    MPI_Wait(&RequestS[CurrentIdx], &StatusS[CurrentIdx]);
  }
}

// Release of persistent communications requests
if (NbPE > 1) {
  MPI_Request_free(&RequestS[0]);
  MPI_Request_free(&RequestR[1]);

  MPI_Request_free(&RequestS[1]);
  MPI_Request_free(&RequestR[0]);
}
```

computations. Similarly to the global distribution of the computations over the nodes, the local subdivision over the computing threads is done in horizontal strips of the square sub-matrix of C that has to be computed at the current step of the main loop. This decomposition is illustrated in Figure 5.6. In respect of the communications, since they are performed in a separate thread, they are implicitly executed in parallel with the computations and there is no need to use asynchronous communications. This is interesting because it reduces the code complexity (no MPI requests management).

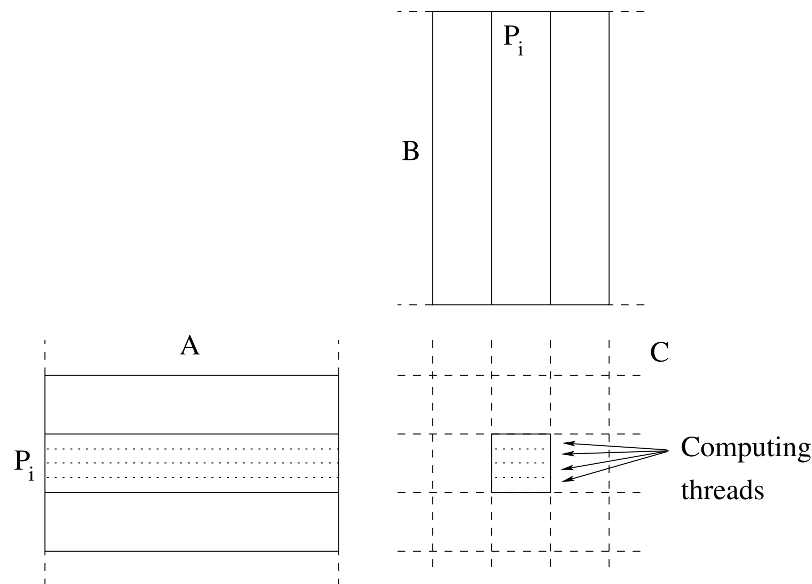

Figure 5.6: Decomposition of the local computations for multiple computing threads

In Algorithm 3 is given the most general version where there may be multiple computing threads. For each computing thread, the start and end indices (`infvalinc` and `supvalexc`) of its assigned horizontal sub-strip of C are computed. The synchronous `Sendrecv` MPI primitive is used in the communication thread.

In the two previous versions, only CPU computations are involved. However, it is currently possible to use additional devices such as GPU to speed up the computations. In the following subsection overlapping strategies in the context of GPU use are presented.

5.2.4 Overlapping MPI Communications and GPU Computations

The computations performed in our benchmark application are very well suited to the use of GPU due to their regularity. However, when using a GPU, data transfers are required to and from the GPU to the central memory of the node. As those transfers may influence the potential of the overlapping (whether it is included in it or not), we have added this information in our monitoring presented in Table 5.2. So there are two sub-columns for the computation time. The first one corresponds to the GPU

Algorithm 3: Overlapping scheme with separate computing and communication threads

lstlisting

```
MPI_Status status;

// Computation and circulation loop
#pragma omp parallel
{
  for (int step = 0; step < NbPE; step++) {
    int idx = step % 2;
    int nbthcalc = omp_get_num_threads()-1;

    // Computation threads
    if (omp_get_thread_num() < nbthcalc) {
      // Dynamic computation of the data range of the thread
      int q = LOCAL_SIZE / nbthcalc;
      int r = LOCAL_SIZE % nbthcalc;
      int infvalinc = q*omp_get_thread_num() + (omp_get_thread_num() < r ? ↩
          omp_get_thread_num() : r);
      int supvalexc = infvalinc + q + (omp_get_thread_num() < r ? 1 : 0);
      // Local computation
      KernelLocalProduct(step, idx, ... , infvalinc, supvalexc);

    // Input data circulation thread
    } else {
      if (NbPE > 1) {
       MPI_Sendrecv(&A[idx][0][0], LOCAL_SIZE*SIZE, MPI_DOUBLE, (Me+1)%NbPE, step,↩
            &A[1-idx][0][0], LOCAL_SIZE*SIZE, MPI_DOUBLE, (Me-1+NbPE)%NbPE, step, ↩
           MPI_COMM_WORLD, &status);
      }
    }

    // Synchronization barrier: wait all computation and comm achieved
    #pragma omp barrier
  }
} // end of parallel region
```

computation time without data transfers and the second one includes them. As a consequence, there are also two sub-columns for the ratio that respectively correspond to both variants of overlapping excluding or including the GPU data transfers.

Number of nodes (P)	GPU computation time (s)		communication time (s)	total exec time (s)	ratio of the shortest activity (%)	
	without transfers	including transfers			without transfers	including transfers
2	1.771	1.819	1.386	5.084	27.27	35.78
4	0.851	0.889	1.549	4.186	20.33	21.24
8	0.401	0.436	1.578	3.757	10.66	11.59
16	0.168	0.199	1.548	3.502	4.79	5.69

Table 5.2: GPU computation, communication and total times in the basic parallel matrices product for $n = 4096$, and potential gain from the overlapping

Although the global benaviours of the computation and communication times are almost the same as in the initial version (see Table 5.1), the computation times are much smaller than with CPU computations. In fact, there is a nearly constant speed up around 25-23 (respectively without and with data transfers) between the CPU computations and the GPU ones according to the number of nodes. This has a double impact over the ratio of potential overlapping. Firstly, the total execution time is much smaller than with CPU computations whereas the communication times are almost the same. So, the proportion of the communications in the total time is implicitly increased. Secondly, the computation time becomes much smaller than the communication time when the number of nodes increases. This tends to decrease the ratio of potential overlapping, as can be seen in the ratio columns of Table 5.2.

From those results, we can conclude that the use of GPU is very interesting for absolute performances but it decreases the interest of overlapping when the number of nodes increases. In the case of our benchmark application, the first overlapping criterion may not be verified for 16 nodes and more. In practice, the decision to make a deeper analysis of the overlapping interest would depend, at this point, on the number of nodes planned to be used when exploiting the application. In the scope of our study, we consider a possible use of the application in any configuration of nodes. So, the analysis of the second criterion (maximal quality of the overlapping) is still pertinent due to the rather significant ratios obtained for small numbers of nodes.

In respect of the overlapping quality, an essential advantage of the GPU kernel calling mechanism in the context of overlapping lies in its naturally asynchronous nature. In fact, once the data have been transferred to the device, a kernel call from the CPU is non-blocking and allows it to perform other tasks during the kernel execution on the GPU without recourse to any additional thread. Obviously, this benaviour is possible because the CPU and the GPU are distinct and independent devices in terms of code execution (like any other co-processor like network or sound devices). Synchronization points can be explicitly inserted in the code if necessary. In respect of the inclusion of GPU transfers in the overlapping, the implementation is more complex

and requires multiple threads (see Section 5.2.4.2), but it can also be achieved quite efficiently.

Finally, it appears that the actual quality of an overlapping of GPU computations with communications should be high, and the second criterion is verified.

In the following two subsections, we propose different versions of overlapping GPU computations with inter-node communications. As the parallel system used for the experiments contains NVIDIA GPUs, the CUDA API is used in the proposed source codes.

5.2.4.1 Natural Overlapping of GPU Computations with Blocking MPI Communications

The first overlapping scheme is quite simple as it only concerns the GPU computations. It mainly consists of executing at each step of the global process, the data transfer of the current A strip to the GPU, the call of the GPU kernel and the inter-nodes communications of A strips. The asynchronous (non-blocking) nature of the GPU kernel calls makes an implicit overlapping of the kernel execution and the inter-node communications.

The corresponding overlapping scheme is given in Algorithm 4. It can be seen that there is no explicit synchronization of the GPU in this code. In fact, this is not required in that case because a single stream is used by default on the GPU and within one stream, only one kernel or data transfer can be executed at a time. Moreover, GPU data transfers are implicitly synchronous. Asynchronous transfers are possible but they imply strong constraints on the involved memory banks that significantly increase the code complexity. So this solution is not pertinent in our context.

Finally, it appears that in our algorithm each GPU data transfer or kernel call will be blocked while the previous one is not finished. Hence, the kernel call in the loop will be executed only once the transfer of A is completed, and the transfer of A in the next iteration will only begin once the kernel call in the previous iteration is finished. This benaviour ensures a valid execution of the main loop of the parallel process while preserving a simple source code.

Although this version should provide very good performances, there are still some possible improvements according to the data transfers to the GPU. As mentioned above, the GPU invocations being exclusive within a single stream, we have seen that the computation kernel and the transfer of A are sequentially scheduled. This is exactly the same for the initial data transfer before the main loop and the transfer of A in the first iteration. Moreover, the use of a synchronous MPI primitive and synchronous GPU transfers prevent the overlapping of the inter-node communications with the transfer of A at the next iteration or with the final data transfer after the loop. So, data transfers to and from the GPU are not included in this overlapping scheme. As a last step towards the complete optimization of the overlapping, a final version including data transfers is proposed in the following subsection.

Algorithm 4: Overlapping scheme with implicit asynchronous GPU kernel call

lstlisting

```
MPI_Status status;

// Transfer of the local strip of B and the node id to the GPU
gpuSetDataOnGPU();
// Computation and circulation loop
for (int step = 0; step < NbPE; step++) {
  int idx = step%2;
  // Transfer of the current local strip of A from the CPU to the GPU
  gpuSetAOnGPU(idx);
  // Computation
  gpuKernelLocalProduct(step,GPUKernelId); // Async call of the GPU kernel
  // Input data circulation
  if (NbPE > 1) {
    MPI_Sendrecv(&A[idx][0][0], LOCAL_SIZE*SIZE, MPI_DOUBLE, (Me+1)%NbPE, step, &A↩
        [1-idx][0][0], LOCAL_SIZE*SIZE, MPI_DOUBLE, (Me-1+NbPE)%NbPE, step, ↩
        MPI_COMM_WORLD, &status);
  }
}
// Get back results from the GPU to the CPU
gpuGetResultOnCPU();
```

5.2.4.2 Inserting the CPU/GPU Data Transfers in the Overlapping Mechanism

As can be seen in Table 5.2, the inclusion of the GPU data transfers in the overlapping increases its interest a little bit. Although it is not a major factor in the decision, it may push back the interest threshold over the number of nodes. So, we propose a final overlapping scheme including those transfers.

For this final optimization, the use of multiple streams on the GPU does not help due to the sequential dependency between the data transfers of A and the kernel calls. Moreover, as mentioned in the previous subsection, asynchronous GPU transfers induce a more complex memory management due to specific constraints. Asynchronous MPI primitives could also be used but, as mentioned previously, they are less efficient than separate threads due to the blocking synchronous data transfers of A to the GPU. In fact, the progress of the MPI communication would not be effective during the transfer of A, thus preventing an actual overlap of those two activities. Moreover, the initial and final transfers respectively to and from the GPU (before and after the main loop) could not be overlapped in such a scheme.

A more convenient and efficient solution is to use separate threads to perform on one hand, the GPU transfers and computations, and on the other hand, the inter-node communications. Such a scheme, using the OpenMP directives, is presented in Algorithm 5. It is quite similar to Algorithm 3 involving the use of OpenMP. However, the number of OpenMP threads is explicitly set to two in this lastest version: one for the GPU management and one for the communications.

There are specific GPU management rules when running multiple threads. Although a GPU device can be used by several threads at a time, each thread can use only one GPU device at a time. It is worth mentioning that it is very delicate to use one GPU device from several threads simultaneously and this should be done only when

Algorithm 5: Overlapping scheme with multiple threads

lstlisting
```
MPI_Status status;     // MPI asynchronous communication status

// Explicit creation of two threads (more is useless)
omp_set_num_threads(2);

// Computation and circulation loop: creation of the threads
#pragma omp parallel
{
  int thId = omp_get_thread_num();

  for (int step = 0; step < NbPE; step++) {
    int idx = step % 2;

    switch (thId) {

    // Computation thread
    case 0 :
      // Initialize GPU usage at step 0
      if (step == 0) {
        cudaSetDevice(0);     // Indicates that thread 0 uses the GPU 0 (optional)
        gpuSetDataOnGPU();    // Data transfer to the GPU
      }
      // Transfer of the current local version of A to the GPU
      gpuSetAOnGPU(idx);
      // Local computation
      gpuKernelLocalProduct(step, GPUKernelId);
      // Finalize GPU usage from thread 0 at last step
      if (step == NbPE-1) {
        gpuGetResultOnCPU(); // Get back the results from the CPU
      }
      break;

    // Communication thread
    case 1 :
      if (NbPE > 1) {
        MPI_Sendrecv(&A[idx][0][0], LOCAL_SIZE*SIZE, MPI_DOUBLE, (Me+1)%NbPE, step↩
             , &A[1-idx][0][0], LOCAL_SIZE*SIZE, MPI_DOUBLE, (Me-1+NbPE)%NbPE, step↩
             , MPI_COMM_WORLD, &status);
      }
      break;
    }

    // Synchronization barrier: wait for termination of both computation and ↩
        communication
    #pragma omp barrier
  }
} // end of parallel region: end of the threads
```

necessary. Mutual exclusion can be imposed between the threads using a same GPU by using lower level CUDA contexts. However, such exclusion mechanisms are not useful in our scheme and are outside the scope of this chapter.

Conversely, the specification of the GPU device used by each thread managing GPU computations would be required in the presence of several GPU devices per node. In fact, the scheme proposed in Algorithm 5 should be extended in the presence of several GPU devices by decomposing the local computations according to the number of available GPU devices and by assigning one thread per GPU device to manage their respective computations and data transfers. So, similarly to the scheme presented in Algorithm 3, each computing thread would manage a part of the locally distributed computations. The main difference is that the threads would do their assigned computations on distinct GPU devices instead of distinct CPU cores. In this case, the function `cudaSetDevice` must be used to indicate which GPU device will execute all the subsequent CUDA invocations made by the calling thread. This function can be called several times by the same thread and allows it to use different GPUs during its execution. With the latest versions of CUDA (4.2), it is not necessary to explicitly assign the GPU device to a thread when there is a single device inside the node. By default, all the threads will use this device.

In our scheme, we have explicitly included this action because it is not implicit in all versions of CUDA, but also in a pedagogic goal to help the reader to keep in mind that the device choice may be necessary with multiple GPU devices.

Finally, this last scheme realizes a potentially complete overlapping and should obtain slightly better performances than the previous one. A global performance comparison of the different versions is presented in Section 5.4. However, experimental results focused on the overlapping are given and analyzed in the next section.

5.2.5 Experimental Comparison and Analysis of the Overlapping Schemes

We present in Table 5.3, a small synthesis of the experimental results obtained with the four overlapping versions. The times (in seconds) reported in this table correspond to the duration of the main computation/communication loop. They are averages of five executions after a first warm up execution. In fact, it is not rare to observe in a series of executions that the first one takes more time than the following ones. This is generally due to automatic energy-saving settings that reduce the frequencies of unused devices (cores or GPUs). So, the first execution is penalized by the time for those previously unused devices to get back to their maximal performance capabilities.

In connection with the first two schemes with CPU computations (Algorithms 2 and 3), several configurations are possible according to the number of computing threads used. In the table are presented the configurations providing the best performances. The first overlapping scheme uses a multi-threaded CPU kernel with 8 OpenMP threads and the second scheme uses 8 sequential CPU kernels in separate OpenMP threads plus another thread for the communications. It can be seen that the

Version	Algorithm 2	Algorithm 3	Algorithm 4	Algorithm 5
Comms	MPI async in main process	MPI sync in 1 thread	MPI sync in main process	MPI sync in 1 thread
Comps	1 CPU kernel using 8 threads	8 CPU kernels in 8 threads	1 GPU kernel with async call	1 GPU kernel + tansfers in 1 thread
# nodes (P)	Main computation-communication loop time (s)			
2	11.664	11.642	1.818	1.919
4	6.545	6.575	1.602	1.501
8	4.138	3.637	1.603	1.523
16	2.943	2.374	1.608	1.522

Table 5.3: Main loop time (s) for the four versions of overlapping with $n = 4096$

results between these two schemes are globally quite similar. However, the second scheme tends to be a bit more scalable when the number of nodes increases (up to 20% better). In fact, those two schemes have quite different benaviours.

As mentioned in Section 5.2.2, the overlapping of asynchronous MPI communications may be ineffective due to the lack of separation between the main process and the communications. In some cases, the communications are actually performed sequentially to the computations during the waiting task. A finer monitoring of this scheme has allowed us to observe that the total time of the main loop corresponds to the sum of the computation time and the communication waiting time at the end of the iterations. Moreover, the comparison with the total time of the main loop obtained with the basic version (Algorithm 1), in the same conditions, shows that they are practically identical. So there is no actual overlapping. A potential improvement would be to interleave asynchronous communication tests and computations as mentioned in [8]. Nevertheless, this cannot be a generic approach as it is not always possible to insert communication tests inside a computation kernel (typically when using a kernel from a library like BLAS).

The results of the second scheme are a bit better. Indeed, the loop times in this scheme are a little bit smaller than the ones in the first scheme and in the basic scheme, with a difference increasing with the number of nodes. This tends to indicate that there is an actual overlapping that seems to increase with the number of nodes. This benaviour can be explained by two reasons. The former is that for small numbers of nodes, the computation time is much longer than the communication time, yielding a low potential of overlapping and thus loop times similar to the first scheme. The latter, already pointed out in the comments of Table 5.1, is that computation times decrease faster than communication times when the number of nodes increases, thus increasing the potential of overlapping up to its maximum for a specific number of nodes (seemingly a little beyond 16 nodes for our first two overlapping schemes). Thus, when the number of nodes increases (up to some threshold not observable in our experiments) the potential overlapping in the second scheme becomes more and more important while there is still no actual overlapping in the first scheme. This explains the performance divergence of those two schemes.

In connection with the last two schemes in Table 5.3 (Algorithms 4 and 5), the first obvious result is that the use of GPUs provides an important gain in the computation time. Moreover, fine monitoring reveals that the overlapping is effective in both versions. This is mainly due to the practically complete hardware independence of the computation and communication activities, as they are performed on different devices. The two proposed schemes present very similar global benaviours and rapidly reach their performance limit with only four nodes. This comes from the fact that in these schemes, the computation time is smaller than the communication time for every multi-node configuration. Hence, their respective loop times are mainly governed by their communication times. Those times slightly decrease when the number of nodes increases due to the smaller data amount to send/receive on each node. But they rapidly reach their lower limit with just a few nodes. It is worth noticing that contrary to the overlapping schemes with CPU computations (Algorithms 2 and 3), the potential overlapping in these last two schemes (Algorithms 4 and 5) decreases when the number of nodes increases. The slight performance difference between these two schemes comes from the overlapping of the GPU transfers that is effective in Algorithm 5. Finally, the (bad) performance of this last scheme with two nodes is quite unexpected. The reasons of this phenomenon are not yet clear and should be the subject of further investigations.

These first experiments have allowed us to get a performance overview of the considered overlapping schemes and to analyze their respective global benaviours. A more detailed analysis of their efficiency is given in Section 5.4.2.

5.3 Impact of Optimization Degree in Computing Kernels

Optimizing computing kernels run on each computing node is a classic objective when developing a HPC code. On modern architectures it means: optimize serial code, parallelize the code on the different cores of a node, and attempt to use an accelerator (like GPU) when available. These optimization degrees are time consuming to develop, except when an adapted optimized library already exists. However, in any case, optimizing the computing kernel run on each node can have a great impact on distributed runs on a cluster.

5.3.1 Typical Degrees of Optimization

When running computations on modern CPU cores, we usually start designing a serial kernel, basically optimized (called *Ck0* in the following). Algorithm 1 illustrates the design of a basically optimized serial algorithm for one CPU core. We use a transposed TB matrix in order to improve cache memory usage and to decrease the number of cache misses. We achieve performance of 1.60 $Gflops$ on one core of our tested cluster (with Nehalem processors). But we can greatly increase this performance.

5.3.1.1 Optimization of CPU Computing Kernels

Algorithm 6 adds OpenMP multithreading to split the main computation loop run on each node (the loop over the lines of the local slice of A as in Figure 5.6). This is achieved very easily by adding just one compilation directive (one line) before the main loop. As mentioned in Section 5.2.2, two copies of A (indexed on the first dimension of the array) are used in order to avoid concurrent read-write accesses. Moreover, we introduced some local variables (PtA and PtB) to access A and TB elements with just one dimensional array indexes. However, this last optimization had no significant impact using the `gcc` compiler (that probably makes this type of optimization by itself).

This multithreaded version (named *Ck1*) achieves a performance close to $6.98\,Gflops$ on one node of our testbed cluster, running 8 threads. Different experiments have shown that best performances are achieved running 8 OpenMP threads on our Nehalem processor with 4 physical cores enhanced with hyperthreading. So, we have to run one thread per logical core (*i.e.* 8 threads).

Algorithm 6: Multithreaded CPU kernel (Ck1)

lstlisting
```
// Local slice of A matrix is stored in A[2][LOCAL_SIZE][SIZE].
// Local slice of transposed B Matrix is stored in TB[LOCAL_SIZE][SIZE].
// Local slice of resulting C Matrix is stored in C[SIZE][LOCAL_SIZE].

// At step "step", the processor compute the C block starting at line:
int OffsetLigneC = ((Me+step)*LOCAL_SIZE)%SIZE;

// OpenMP parallelization of the main loop on A[idx] lines
#pragma omp parallel for
for (int i = 0; i < LOCAL_SIZE; i++) {
   double *PtA = &A[idx][i][0];          // Ptr on the current A line
   for (int j = 0; j < LOCAL_SIZE; j++) {
      double *PtTB = &TB[j][0];          // Ptr on the current TB line
      double accu = 0.0;                 // Local accumulator of a new result
      // Compute a new value of the resulting C matrix
      for (int k = 0; k < SIZE; k++) {
         accu += PtA[k] * PtTB[k];
      }
      // Store the new result in C matrix
      C[i+OffsetLigneC][j] = accu;
   }
}
```

Algorithm 7 is another version (named *Ck2*) that uses the matrix-matrix product of the famous BLAS library (`cblas_dgemm` routine) to perform on each node the computation of the local sub-matrix of C. The BLAS function call requires several parameters, specifying the storage format of the A, B and C matrices as well as their respective line and column sizes. This library is well known to the HPC community (that never redevelops a matrix-matrix multiplication), and is generally supplied by constructors or by specialized communities. We achieved a performance close to $9.81\,Gflops$ on one core of our testbed cluster. We used the ATLAS version of the BLAS library, but with pure sequential implementation of each routine.

Algorithm 7: Sequential BLAS CPU kernel (Ck2)

lstlisting
```
// idx: index of the 2D array of A[2][LOCAL_SIZE][SIZE] to read at current step.
// OffsetLigneC: starting line of the C block computed at current step.

cblas_dgemm(CblasRowMajor,  CblasNoTrans, CblasNoTrans, LOCAL_SIZE, LOCAL_SIZE, ↩
    SIZE, 1.0, &A[idx][0][0], SIZE, &B[0][0], LOCAL_SIZE, 0.0, &C[OffsetLigneC][0],↩
     LOCAL_SIZE);
```

Finally, Algorithm 8 illustrates a computing kernel still based on a BLAS library call, but applied to a subpart of the local slice of the A matrix: processing lines in the range `[InfValInc;SupValExc[` (kernel *Ck3*). This computing kernel is called from an OpenMP multithreaded algorithm (see Algorithm 3). Several OpenMP threads are run, each thread computes its `[InfValInc;SupValExc[` range and calls the *Ck3* kernel. Experiments have pointed out that the most efficient way of using kernel Ck3 was to run only one computing thread per physical core (and not relying on the hyperthreading mechanism). We reached $36.30\ Gflops$ on our testbed nodes.

Algorithm 8: BLAS CPU kernel to be used in a multithreaded scheme (Ck3)

lstlisting
```
// idx: index of the 2D array of A[2][LOCAL_SIZE][SIZE] to read at current step.
// OffsetLigneC: starting line of the C block computed at current step.
// InfValInc: index of the first line of A[idx] processed by the thread.
// SupValExc: index of the first line of A[idx] not processed by the thread.

cblas_dgemm(CblasRowMajor,  CblasNoTrans,  CblasNoTrans, (SupValExc - InfValInc),  ↩
    LOCAL_SIZE,  SIZE, 1.0,  &A[idx][InfValInc][0],  SIZE, &B[0][0],  LOCAL_SIZE, ↩
    0.0,  &C[OffsetLigneC+InfValInc][0],  LOCAL_SIZE);
```

5.3.1.2 Optimization of GPU Computing Kernels

Using a GPU allows for the achievement of higher performance when the computations are adapted to the *vector-like* architecture and programming model. Details about GPU programming (using CUDA framework for NVIDIA cards) is beyond the scope of this study. However, we aim to show that there are similarities with CPU programming according to the optimization process of computing kernels.

Algorithm 9 shows the source code of a basic CUDA kernel *Gk0* (top) and the source code of this kernel call (bottom). This short code is composed of very classical operations in a CUDA program. It is quite simple as it uses only the *global memory* of the GPU and it accesses data without fully respecting their *coalescence*. Those omissions may induce a performance degradation as a part of the memory bus bandwidth of the GPU is wasted and the thread scheduling may not be optimal. We achieved a performance of $38.80\ Gflops$ on the GTX480 GPU board of each node of our testbed.

If we go a step further in the optimization process, Algorithm 10 introduces a medium optimized GPU computing kernel (*Gk1*). It uses the global memory of the

Algorithm 9: Basic GPU kernel (Gk0)

lstlisting

```
// Definition of the GPU computing kernel
__global__ void MatrixProductKernel_Gk0(int step)
{
 int lig = blockIdx.y;
 int col = threadIdx.x + blockIdx.x * BLOCK_SIZE_X_K0;
 double res = 0.0;

 if (col < LOCAL_SIZE) {
   for (int k = 0; k < SIZE; k++) {
     res += GPU_A[lig][k] * GPU_B[k][col];
   }
   GPU_C[lig+(((GPU_Me+step)*LOCAL_SIZE)%SIZE)][col] = res;
 }
}
```

lstlisting

```
// Call of the GPU computing kernel Gk0

// Description of a block of threads
Db.x = BLOCK_SIZE_X_K0;
Db.y = 1;
Db.z = 1;
// Description of a grid of blocks
Dg.x = LOCAL_SIZE/BLOCK_SIZE_X_K0 + (LOCAL_SIZE%BLOCK_SIZE_X_K0 ? 1 : 0);
Dg.y = LOCAL_SIZE;
Dg.z = 1;

// Run the grid of blocks of threads with Gk0
MatrixProductKernel_Gk0<<<Dg,Db>>>(step);
```

GPU but also its *shared memory*: a fast memory used like a cache memory explicitly managed by the developer in the source code. In the previous GPU algorithm, another part of this fast memory was a real cache memory entirely managed by the GPU. Moreover, the Gk1 kernel has also coalescent data accesses that allow for a better scheduling of GPU threads. That kernel is to be used on a two-dimensional grid of thread blocks as presented in Algorithm 11. The `cudaFuncSetCacheConfig` function is used before the kernel call in order to increase the size of the shared memory that can be used by the kernel. This kernel source is longer and more complex than the first one. Not all developers can write or maintain this code. But it achieves a performance of $98.00\ Gflops$ on one node of our testbed (instead of $38.80\ Gflops$ with GK0 !).

Finally, Algorithm 12 introduces the usage of the highly optimized cuBLAS library: calling the routine `cublasDgemm` and a transposition kernel (user defined). The call to the cuBLAS library routine is close to the call to the BLAS library routine (on CPU), but the result is always stored in *column major* mode as in FORTRAN libraries. Then, it is necessary to transpose the resulting matrix at the end of each step before storing it in the entire local slice of the C matrix. This last GPU kernel is named *Gk2*. Calling the cuBLAS library is more complex than calling the BLAS library on a CPU, but it remains reasonable for GPU developers and it achieves a high performance of $154.00\ Gflops$ on one node of our testbed.

Algorithm 10: GPU kernel using shared memory (Gk1)

lstlisting

```
__global__ void MatrixProductKernel_Gk1(int step)
{
 __shared__ double sh_A[BLOCK_SIZE_Y_K1][BLOCK_SIZE_X_K1]; // Local "cache" of A
 __shared__ double sh_B[BLOCK_SIZE_X_K1][BLOCK_SIZE_X_K1]; // Local "cache" of B
 double res = 0.0;                                          // Local result storage
 int ligC = threadIdx.y + blockIdx.y*BLOCK_SIZE_Y_K1;      // Coordinates of the C
 int colC = threadIdx.x + blockIdx.x*BLOCK_SIZE_X_K1;      // elt computed
 int colA = threadIdx.x;                                    // Initial indexes of
 int ligB = threadIdx.y;                                    // A column and B line

 // For each step: process BLOCK_SIZE_X_K1 elt of the required A line and B column
 for (int step = 0; step < SIZE/BLOCK_SIZE_X_K1; step++) {
   // Load A data into the shared sh_A array
   if (ligC < LOCAL_SIZE) {
     sh_A[threadIdx.y][threadIdx.x] = GPU_A[ligC][colA];
     colA += BLOCK_SIZE_X_K1;
   }
   // Load B data into the shared sb_B array
   if (colC < LOCAL_SIZE) {
     int ligShB = threadIdx.y;
     for (int sstep = 0; sstep < BLOCK_SIZE_X_K1/BLOCK_SIZE_Y_K1; sstep++) {
       sh_B[ligShB][threadIdx.x] = GPU_B[ligB][colC];
       ligB += BLOCK_SIZE_Y_K1;
       ligShB += BLOCK_SIZE_Y_K1;
     }
   }
   // Wait for all threads having updated the A and B "cache memories"
   __syncthreads();
   // Update C value using A and B data uploaded into the shared memory
   if (ligC < LOCAL_SIZE && colC < LOCAL_SIZE)
     for (int k = 0; k < BLOCK_SIZE_X_K1; k++)
       res += sh_A[threadIdx.y][k] * sh_B[k][threadIdx.x];
   // Wait for all threads having finished to use current values in the A and B ↩
       caches
   __syncthreads();
 }
 // Last step: process the remaining elts of the required A line and B colum
 if (SIZE % BLOCK_SIZE_X_K1 != 0) {
   // Cache a last value of A
   if (ligC < LOCAL_SIZE && colA < SIZE)
     sh_A[threadIdx.y][threadIdx.x] = GPU_A[ligC][colA];
   // Cache some last values of B
   if (colC < LOCAL_SIZE) {
     int ligShB = threadIdx.y;
     for (int sstep = 0; sstep < BLOCK_SIZE_X_K1/BLOCK_SIZE_Y_K1 && ligB < SIZE; ↩
         sstep++) {
       sh_B[ligShB][threadIdx.x] = GPU_B[ligB][colC];
       ligB += BLOCK_SIZE_Y_K1;
       ligShB += BLOCK_SIZE_Y_K1;
     }
   }
   // Wait for all threads having updated the A and B "cache memories"
   __syncthreads();
   // Update C value with the last A and B values uploaded in the shared memory
   if (ligC < LOCAL_SIZE && colC < LOCAL_SIZE)
     for (int k = 0; k < SIZE % BLOCK_SIZE_X_K1; k++)
       res += sh_A[threadIdx.y][k] * sh_B[k][threadIdx.x];
 }
 // Store the final computed value into the C matrix variable
 if (ligC < LOCAL_SIZE && colC < LOCAL_SIZE)
   GPU_C[ligC][colC] = res;
}
```

Algorithm 11: Gk1 GPU kernel call

lstlisting
```
// Description of a block of threads
Db.x = BLOCK_SIZE_X_K1;
Db.y = BLOCK_SIZE_Y_K1;
Db.z = 1;
// Description of a grid of blocks
Dg.x = LOCAL_SIZE/BLOCK_SIZE_X_K1 + (LOCAL_SIZE%BLOCK_SIZE_X_K1 ? 1 : 0);
Dg.y = LOCAL_SIZE/BLOCK_SIZE_Y_K1 + (LOCAL_SIZE%BLOCK_SIZE_Y_K1 ? 1 : 0);
Dg.z = 1;
// Maximize the size of the GPU shared memory for the Gk1 kernel
cudaFuncSetCacheConfig(MatrixProductKernel_Gk1, cudaFuncCachePreferShared);
// Run the grid of blocks of threads with the required kernel
MatrixProductKernel_Gk1<<<Dg,Db>>>(step);
```

Algorithm 12: cuBLAS based GPU kernel (Gk2)

lstlisting
```
// Compute AxB calling cuBLAS library
cublasDgemm(handle, CUBLAS_OP_T, CUBLAS_OP_T, LOCAL_SIZE, LOCAL_SIZE, SIZE, &alpha,↩
     Adr_GPU_A, SIZE, Adr_GPU_B, LOCAL_SIZE, &beta, Adr_GPU_R, LOCAL_SIZE);
// Transpose Column Major result into a part of the C matrix
// Description of a block of threads
DbT.x = BLOCK_SIZE_XY_TK0;
DbT.y = BLOCK_SIZE_XY_TK0;
DbT.z = 1;
// Description of a grid of blocks
DgT.x = LOCAL_SIZE/BLOCK_SIZE_XY_TK0 + (LOCAL_SIZE%BLOCK_SIZE_XY_TK0 ? 1 : 0);
DgT.y = LOCAL_SIZE/BLOCK_SIZE_XY_TK0 + (LOCAL_SIZE%BLOCK_SIZE_XY_TK0 ? 1 : 0);
DgT.z = 1;
// Run the transposition kernel on the grid of blocks of threads
TransposeKernel_v0<<<DgT,DbT>>>(Adr_GPU_R, Adr_GPU_C + ((step+Me)%NbPE) * ↩
    LOCAL_SIZE * LOCAL_SIZE, LOCAL_SIZE, LOCAL_SIZE);
```

5.3.2 Experimental Highlighting of the Kernel Optimization

CPU kernel version	Ck0 (basic optimization)	Ck1 (OpenMP multithreading)	Ck2 (Atlas/BLAS monothreaded)	Ck3 (Atlas/BLAS + OpenMP multithreading)
Gflops	1.60	6.98	9.81	36.30
Speedup	1.0	4.4	6.1	22.7

Table 5.4: Performances of the different CPU kernels

In Table 5.4 are presented the CPU kernels performances achieved on one node of our testbed, together with their respective speedups according to the first basic version (Ck0). A regular and significant improvement of the performance with the increase of the optimization degree can be seen. Moreover, we can observe that the C source code of the different CPU versions remains *reasonably simple*. Using the BLAS library is easy. So, it appears really interesting to develop these improvements on CPU, and specially the last one, calling a BLAS implementation and adding OpenMP multithreading. Theoretically, the Atlas implementation of the BLAS can be compiled in a multithreaded way in order to internally use the available cores on each node. However, our version of Atlas library was not multithreaded, and we had to explicitly manage multithreading via OpenMP.

GPU kernel version	Gk0 (basic)	Gk1 (medium optimized)	Gk2 (cuBLAS)
Gflops	38.80	98.00	154.00
Speedup vs GPU-basic	1.0	2.5	4.0
Speedup vs CPU-optim	1.1	2.7	4.2
Speedup vs CPU-basic	24.3	61.3	96.3

Table 5.5: Performances of the different GPU kernels

In Table 5.5 the performances obtained with the different GPU kernels are given. The optimized cuBLAS based kernel (*Gk2*) is 4.0 times faster than the basic GPU kernel, 4.2 times faster than the most optimized CPU kernel (based on the BLAS library and OpenMP multithreading), and 96.3 times faster than the basic and sequential CPU kernel. The final performance improvement of our matrix product kernel on one node is significant. However, it requires the learning of CUDA programming in order to use GPU accelerators. Moreover, if the problem to solve is more complex or more original than a simple dense matrix product, it is probable that no highly optimized libraries (like BLAS and cuBLAS) will be available. Then the developer will have to design and implement some highly optimized kernels by himself. This type of work requires a lot of expertise and a long development time, independently of the distribution on several computing nodes.

In the next section we investigate the impact of kernel optimization over the performance of a distributed version on a cluster, and we attempt to identify the right couples (kernel optimization, parallel scheme optimization).

5.3.3 Decision Chain for Optimization of Computing Kernels

A computing kernel optimization process can lead to long and expensive development. To enter this kind of process is an important decision. A chain of decision criteria has to be considered.

As explained in Section 5.2.1, the first criterion is the ratio of the considered kernel in the entire application. It is not very useful to spend great effort on optimization in a kernel that takes only 1% or 2% of the total application time. On the other hand, if the kernel represents a significant part of the total execution time, then we have to study the criteria introduced in the following paragraphs.

5.3.3.1 Technical Criterion

When considering a computing kernel, the first criterion to evaluate before entering a new optimization development step is whether its performance can be theoretically optimized or not. A possible approach is based on counting the computing operations and the memory accesses achieved by the kernel. This approach uses the concept of *arithmetic intensity* introduced by NVIDIA in [15]. We partially studied this approach in [13] and more deeply in a collaboration with the EDF company [12].

The main steps are:

1. Counting the number of floating point operations achieved by the kernel: n_{ops}.

2. Counting the number of memory accesses achieved by the kernel. However, we need a model of the architecture, or at least some hypothesis about the hierarchy and speeds of the different memories. For example, we may count only accesses to the main memory, not to cache memories. So, we need to make assumptions about the cache size and cache management. Finally, we obtain a number of accesses in function of the architecture: $n_{a,archi}$.

3. Computing the *arithmetic intensity*: $i_a = n_{ops}/n_{a,archi}$, the average number of operations achieved per memory access.

4. Comparing this value to the *critic arithmetic intensity*: i_c, defined as the ratio between the processor speed (flops) and the memory access speed (bandwidth).

5. Deducing the theoretical minimal execution time:

   - If $i_a > i_c$, the kernel is *cpu-bound*: it is limited by the computing speed of the processor, and the theoretical minimal execution time is: $t_{ideal} = n_{ops}/(processor_speed)$.

   - If $i_a < i_c$, the kernel is *memory-bound*: it is limited by the memory bandwith, and the theoretical minimal execution time is: $t_{ideal} = n_{a,archi}/(memory_bandwidth)$.

When the experimental execution time is sufficiently larger than the theoretical minimal time, then it is interesting to attempt to optimize the kernel code. Moreover, it is sometimes possible to design new algorithms with an arithmetic intensity (i_a) closer to the critical one (i_c), in order to better exploit both computing and memory access capabilities of the processor.

5.3.3.2 Required Expertise Criterion

The first steps of the CPU kernel optimization are *standard* and require basic computer science knowledge. The usual optimization includes data structure and data access design in order to read and/or write contiguous memory locations and to avoid cache misses. Then, a simple access to multithreading via libraries like OpenMP is possible for loops performing independent iterations. A software developer with basic knowledge about processor architectures and multithreading libraries can achieve this degree of serial optimization and multithreading. However, the next steps are much more complex, and very few developers are able to design optimized code like BLAS libraries. Moreover, using multithreading leads to particular data storage and data access optimization, requiring specific knowledge.

When using GPUs, the required level of expertise increases quickly. Designing algorithms and codes with *coalescent* memory accesses is the basic training of a CUDA developer, and is usually well understood. But, developing explicit caching algorithms to exploit the fast *shared memories* of the NVIDIA GPUs requires a higher level of expertise. Many developers will never reach this second level of expertise. However, GPUs do not (yet) support a complex OS with various tasks. It is possible to monitor and to control what happens on a GPU, and finally to acquire a very high level of expertise and to achieve very optimized kernels.

In any case, achieving computing kernel optimization requires an adapted level of expertise, that takes time to acquire. So, an important decision criterion for starting an optimization process or not, is the availability (and the cost) of this expertise inside the development team or the company. Obviously, when an existing highly optimized library can be used in the kernel, like BLAS in our benchmark application, the right solution is to learn the usage of this library and to adapt the kernel to use it. It often allows the achievement of high performance with limited extra development efforts and a low expertise level.

5.3.3.3 Use Context Criterion

When the kernel code is used in a multi-node parallel context, the kernel time will be often compared to the communication time. When an overlapping of those two activities is possible, an efficient strategy is to reduce computation and communication times as much as possible so that they become of the same order. This maximizes the potential overlapping. However, it is shown in Section 5.4.2 that it is sometimes efficient to have communication times greater than computation times, in order to get an experimental gain close to the expected one (deduced from the potential overlapping).

Some software is designed for a long term exploitation and for intensive usage. In such cases, the development time remains smaller than the sum of the execution times ($T_{dev} << \sum(T_{exec})$) of the application during its lifetime. So, increasing the development time in order to decrease each execution time a bit may have a strong attraction. Nevertheless, if the application has a short life cycle, or if it is to be used rarely, the optimization effort may be more time consuming than the sum of the gained times in all the executions of the optimized version during its lifetime.

5.3.3.4 Complete Decision Chain

Finally, before investing into optimization effort of some computation kernels, one has to take care of:

(1) the ratio of this kernel in the total execution time of the application,

(2) the distance between the performance obtained and the theoretical performance of this kernel with the considered parallel architecture,

(3) the expertise level of the available developers or the existence of a suitable highly optimized library,

(4) the opportunity to overlap kernel computations with inter-node communications (in a multi-node architecture), and

(5) the estimated total amount of execution times of the application during its lifetime compared to the extra development time required.

5.4 Global Experiments and Analysis

The hardware and software components of the parallel system that has been used to perform the following experiments are detailed in Section 5.1.3.

5.4.1 Experimentation Strategy

The different versions of the benchmark application have been tested on different numbers of nodes of the cluster and with different computing variants (variable number of computing threads when relevant). As it is not reasonable (and not fully pertinent) to present the entire set of results obtained from those experiments in this chapter, two levels of selection have been applied. The first one concerns the application variants (overlapping scheme and computing kernel). Hence, Algorithm 2 is not selected as it has already been shown in Section 5.2.5 that this scheme is not efficient at all. The second selection level is related to the runtime configurations (number of computing threads). For each selected variant only the runtime configuration that has obtained the best performance is retained.

As mentioned in Section 5.2.5, each result is the average of five consecutive executions after an initial warm-up. Moreover, as a first analysis of the overlapping schemes

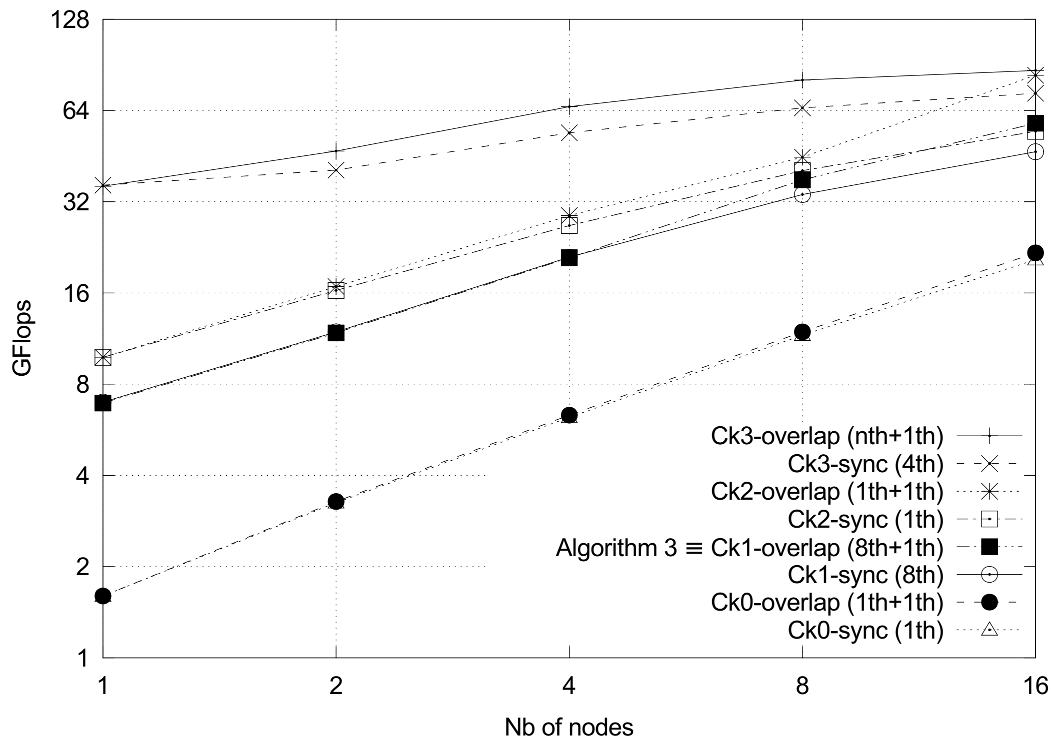

Figure 5.7: Performance curves of different CPU computing kernels with or without overlapping on a multi-core CPU cluster

already has been presented in that previous section, the current section is more dedicated to the analysis of the actual efficiency of the overlapping (according to the expected performance) as well as the efficiency of the computing kernels.

5.4.2 Experimental Results

For clarity's sake, the analysis is decomposed into two parts according to the type of computing kernel.

5.4.2.1 Performance on Multicore CPU Cluster

In Figure 5.7 are shown the performance curves achieved, with 1 to 16 nodes, by our different CPU kernels, with no overlapping or with the overlapping scheme in Algorithm 3. The two bottom curves correspond to the performance achieved with the *Ck0* CPU kernel that includes only some basic serial optimization (see Section 5.1.2 and algorithm 1) and that uses only one CPU core (no multithreading inside this kernel).

In the *Ck0* CPU kernel with no overlapping (*Ck0-sync* curve), the communication time ranges from 3.2% up to 20.4% of the entire application time respectively from 2 up to 16 nodes. Moreover, it is 3.4% up to 25.8% of the computation loop time. This computing kernel has limited performance, but when the number of nodes in-

creases it becomes interesting to overlap communications with computations to save up to 25.8% of the computation loop time (see Section 5.2.1). Then we have implemented the overlapping scheme in Algorithm 3, using an explicit OpenMP thread to achieve MPI communications, and another thread to run the computation kernel (see Section 5.2.3). Finally, the performance curve *Ck0-overlap* stays quite similar to *Ck0-sync* but shows a small improvement on 16 nodes.

We have followed a very similar approach with the *Ck1* CPU kernel, that is a multi-threaded version of the previous one. Curves *Ck1-sync* and *Ck1-overlap* respectively show the performance achieved without and with overlapping. It has to be noticed that due to technical constraints in OpenMP, the actual implementation of the *Ck1-overlap* is Algorithm 3. However, these two schemes have the same semantics that consist of having several computing threads and one communication thread. The analysis of the curves shows that running 8 threads on the 4 hyperthreaded cores of our Nehalem processors has led to a significant decrease in the computation time. The communication time remains unchanged, and from 2 to 16 nodes it ranges from 12.1% to 35.1% of the application time, and from 14.1% to 55.8% of the computation loop time. In this context, the overlapping of computations with communications is highly attractive. With 8 and 16 nodes, the performance increase is significant and justifies the development effort of the overlapping.

The next curves concern the usage of a BLAS library kernel (Atlas implementation), identified as the *Ck2* CPU kernel in Figure 5.7. This is a sequential kernel using only one core, but with a very high degree of optimization. With this kernel, the communication time ranges from 13.5% to 37.1% of the application time, and from 16.5% to 62.6% of the computation loop time. The performance increase on the *Ck2-overlap* curve, compared to the *Ck2-sync* curve, starts as soon as 2 nodes are used and becomes surprisingly strong with 16 nodes. In fact, when using 16 nodes with this computing kernel, communications become longer than computations (in the computation loop): $T_{comm} = 1.7 \times T_{comput}$. Then, the loop computation time appears to be very close to the communication time: the time saved by the overlap is 93% of the expected one. On 8 nodes, the communication time is less than the computation time ($T_{comm} = 0.86 \times T_{comput}$), and the time saved by the overlap is only 22% of the expected one. It seems that overlapping inter-node communications with node computations is better when the computation time is smaller than the communication time. However, in this case the performance is bounded by the communications. Usually, designers of parallel codes try to get communications shorter than computations. See Section 5.4.3.1 for a detailed analysis of this phenomenon.

The last performance curves correspond to the variant of the application with the same sequential BLAS kernel, but called in parallel from different threads in order to use the available cores of the CPU. Parallel runs of the BLAS kernel have been tested with 2, 3, on up to 12 threads. With no overlapping, the number of threads providing the best performance is constant and corresponds to 4. With the overlapping, the best number of threads varies with the number of nodes (and so with the size of the sub-problem run on each node). With 2, 4 and 8 nodes the best performance is obtained with 8 computing threads (and one communication thread), while with 16 nodes only

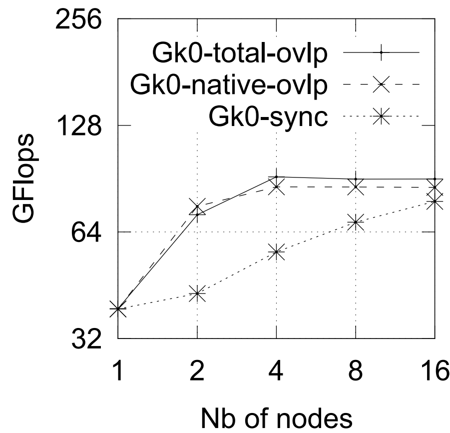

Figure 5.8: Performance of the basic Gk0 kernel on a GPU cluster

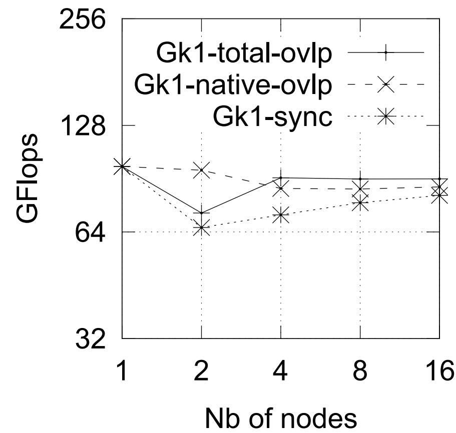

Figure 5.9: Performance of the optimized Gk1 kernel on a GPU cluster

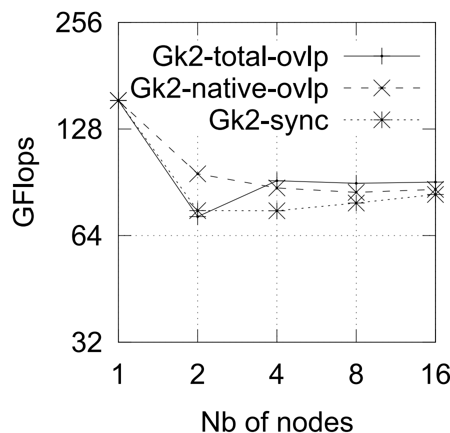

Figure 5.10: Performance of the highly optimized Gk2 kernel on a GPU cluster

4 computing nodes (and one communication thread) are necessary. These results are depicted by the *Ck3-overlap* curve. Compared to the *Ck3-sync* curve, the improvement in the overlapping is visible and significant from 2 to 16 nodes. In this context, the communication time is greater than the computation time with 4, 8 and 16 nodes. However, the overlap is close to 100% of its expected value only with 16 nodes when $T_{comm} = 5.5 \times T_{comput}$.

5.4.2.2 Performance on GPU Cluster

We have experimented each of our three GPU kernels with three parallel variants: with no overlapping (Algorithm 1 with a GPU kernel), with a native overlapping mecha-

nism (Algorithm 4), and with an overlapping including the CPU/GPU data transfers (Algorithm 5). In Figure 5.8 are given the three performance curves of the *Gk0* kernel, a basic GPU kernel not using the fast *shared memories* of the GPU but using only its *global memory* (and some registers). The bottom curve illustrates the performance of the version with no overlap of the inter-node communications, GPU computations and data transfers between CPUs and GPUs. We have enforced synchronization of the GPU kernel so that these operations do not overlap. From 1 to 2 nodes the performance increase is poor, and is stronger from 2 to 16 nodes. Using 2 nodes doubles the computing power, but inserting inter-node communications is a great penalty that seriously limits the impact of a second GPU. When increasing the number of nodes from 2 to 16, the communication time increases very slightly (from $1.39s$ to $1.58s$) while the computing power increases strongly. So the performance curve increases significantly up to 16 nodes. The upper performance curves correspond to the overlapping modes. The penalty of inter-node communications is not sensitive from 1 to 2 nodes, but on 4 nodes the computation time (on the GPU) is less than the inter-node communication time. So the overlapped execution time is limited by the communication time that remains approximately constant, and the performance reaches its limit. Our gigabit Ethernet interconnection network appears insufficient to support GPU nodes: they require and produce data faster than this network can route.

The highest curve shows the performance of the version overlapping inter-node communications with both GPU computations and CPU/GPU data transfers. This code is more complex to develop (see Section 5.2.4.2 and Algorithm 5), but it achieves slightly better performances from 4 to 16 nodes than the native overlapping version. However, performances are a little bit weaker on 2 nodes. As already mention in Section 5.2.5, this phenomenon is still under investigations.

The results obtained with the *Gk1* GPU kernel are given in Figure 5.9. This kernel code is more complex and uses the fast *shared memories* of the GPU like a cache memory explicitly managed by the developer at application level. With or without any overlap mechanism, we can observe that best performance is obtained on only 1 computing node. Performance of this computing kernel on one node is approximately three times greater than performance of the *Gk0* kernel, and computation times are smaller than communication times. So the communication times of our gigabit Ethernet network always compensate the computation times won by using several nodes, and the total execution time does not decrease. This phenomenon is more visible with the third GPU kernel (*Gk2*), using the highly optimized cuBLAS library. Figure 5.10 shows very high performance on only 1 node (close to $154Gflops$), and a strong performance decrease when using more nodes, even with a total overlap of the communications with both computations and data transfers.

5.4.3 Discussion

5.4.3.1 Assessment of Overlapping Strategy on CPU Clusters

Finally, overlapping communications with computations on a multicore CPU cluster is successful with the strategy based on an explicit thread running and managing the MPI communications (see Section 5.2.3). However, it is not obvious to reach the ideal execution time: usually we observe $T_{overlap} > max(T_{comput}, T_{comm})$. Considering that in the ideal case $T_{saved}^{ideal} = (T_{comput} + T_{comm}) - max(T_{comput}, T_{comm}) = min(T_{comput}, T_{comm})$, detailed experiments on our cluster have shown that:

- when $0 < \frac{T_{comm}}{T_{comput}} < 0.95$, we get: $T_{saved} < 0.30 \times T_{saved}^{ideal}$,
- when $1.0 < \frac{T_{comm}}{T_{comput}}$, we can achieve: $T_{saved} = T_{saved}^{ideal}$.

So, although obtaining $T_{saved} = T_{saved}^{ideal}$ seems attractive, usually it provides *inefficient* results as the communication times are longer than the computation ones. Even if the overlapping strategy can lead to achieving 100% of the expected gain, this gain is strongly limited by the small overlapping potential. Moreover, it may require a lot of resources for a limited extra-speedup. For example, the *Ck3-overlap* performance curve increases up to 16 nodes, but the minor increase from 8 to 16 nodes does not justify doubling the cluster size. It is important to know how to track and achieve overlapping to improve the performance, but not to track it at all costs.

5.4.3.2 Assessment of Overlapping Strategy on GPU Clusters

Our experiments show that an overlapping mechanism can lead to significant performance increase on a GPU cluster. We can evaluate the efficiency of the two overlapping strategies that have been implemented. The synchronous implementation of the computation loop can be modeled with:

$$T_{loop}^{sync} = T_{comp} + T_{trans} + T_{comm}$$

the execution time of the native overlap strategy is:

$$T_{loop}^{native-ovlp} = T_{trans} + max(T_{comp}, T_{comm})$$

and the execution time of the more complex and total overlap strategy is given by:

$$T_{loop}^{total-ovlp} = max(T_{comp}, T_{trans} + T_{comm})$$

As on the CPU cluster, we measured T_{comp}, T_{trans} and T_{comm} with the (strongly) synchronous version (no overlapping). Then, the ideal saved times that are expected for each overlapping strategy are computed and compared to the actual saved times obtained in the experiments. Finally, the ratios T_{comm}/T_{comp} (or $T_{comm}/(T_{comp} + T_{trans}))$ and $T_{saved}/T_{saved}^{ideal}$ are deduced. We obtained very different results from the ones with the CPU kernels. With the *Gk0* kernel:

- with the native overlapping strategy we obtain:
  $0.75 < T_{comm}/T_{comp} < 8.4$, and $0.84 < T_{saved}/T_{saved}^{ideal} < 1.02$

- with the total overlapping strategy we obtain:
  $0.75 < T_{comm}/(T_{comp} + T_{trans}) < 7.8$, and $0.92 < T_{saved}/T_{saved}^{ideal} < 1.12$

Compared to the CPU kernel results, we have a greater ratio T_{comm}/T_{comp} due to the very high speed of the GPUs. Therefore communication times quickly exceed computation times (even when adding the transfer times). Moreover, the overlapping being very efficient in those contexts, the major part of the expected gain of the overlapping is actually achieved: at least 84% with the basic and native overlapping strategy. When using the total overlapping strategy, we have measured a gain of time greater than the expected one (110% of the expected time). Obviously, such results are not coherent with the theory. However, they can be explained (at least partially) by the additional GPU synchronization that had to be used in the basic version with no overlapping in order to force the CPU to wait for the GPU kernel termination. We remind the reader that by default a GPU kernel call is non-blocking. Such synchronization induces additional costs that may lead to measuring longer computation times than the real ones.

Anyway, independently of the problem of the exact correspondence of the monitored activities between two application variants, it appears that the native overlapping strategy on a GPU cluster is very easy to deploy, the total overlapping strategy is not so complex (see Section 5.2.4), and both are successful. We save almost 90% of the expected time with a T_{comm}/T_{comp} ratio around 1, and almost 100% with a higher ratio. However, as we claimed for CPU clusters, it is not interesting to run parallel programs with strongly dominant communication times. In Figure 5.8 a strong improvement when using overlapping mechanisms can be observed, but there is no global improvement when using more than 4 nodes with these mechanisms (using more resources is *useless*).

Analyses of the T_{comm}/T_{comp} and $T_{saved}/T_{saved}^{ideal}$ ratio for *Gk1* and *Gk2* kernels would lead to similar results to the *Gk0* ones. However, they have a limited interest. As explained previously, our interconnection network is not fast enough to use these kernels. A network upgrade would be required (towards Infiniband for example), or the use of benchmark requiring many more computations.

5.4.3.3 Looking for the Most Interesting Solution

Due to the limited capacity of our gigabit Ethernet interconnection network, the number of nodes providing the best performance of our benchmark problem decreases when the speed of the computing kernel increases. Table 5.6 summarizes the configurations most suited to the different kernels. With all the CPU kernels, it is interesting to use the 16 nodes of the cluster and to implement a multithreaded overlapping of computations with communications. However, when running the fastest CPU kernel, the performance increase from 8 to 16 nodes is small.

Kernel	Most suited parallel scheme	Nodes	Gflops
Ck0	Overlapping (1 comp. thread and 1 comm. thread)	16	22
Ck1	Overlapping (8 comp. thread and 1 comm. thread)	16	58
Ck2	Overlapping (1 comp. thread and 1 comm. thread)	16	84
Ck3	Overlapping (*n* comp. thread and 1 comm. thread)	16	87
Gk0	Total overlapping (comm. *vs* comp. + trans.)	4	92
Gk1	Mono-node exec. (no comm.)	1	98
Gk2	Mono-node exec. (no comm.)	1	154

Table 5.6: Configurations providing the best performance for each computing kernel

When running GPU kernels, the most interesting number of nodes decreases. With the basic *Gk0* GPU kernel (easy to design) it is better to use only 4 nodes, and to implement a total overlapping of communications with both computations and CPU/GPU data transfers. With faster GPU kernels, it is better to use only 1 node to run this benchmark problem. Then, no inter-node communication is required and it is not necessary to implement any overlapping mechanism.

Obviously, it could be interesting to run a larger benchmark, with bigger matrices. But problems to solve are not always *infinitely scalable*. Sometimes we have to solve a large but finite problem. Then, we can look for the fastest solution, with limited development time and with limited computing resources. If some adapted high performance libraries are available (like BLAS and cuBLAS in our example) it is highly recommended to (test and) use such libraries. When some problem-specific high performance kernels have to be developed, a pertinent trade-off between the development time (and cost) and the gain in execution time must be found. For example, we can track to:

- decrease the execution time under a fixed threshold, no more, no less,
- minimize the sum: $T_{dev} + \sum(T_{exec})$

When spending a lot of time (and money) to develop a very fast computing kernel, it is possible to exceed the capacities of the available interconnection network. Then, the application performance will be limited to the speed achieved with only a few nodes, or even only one node (if the problem size fits the memory size on one node). Then, it may be more interesting to spend less time developing the computing kernel, and to spend some time overlapping computations with communications in the parallel program. Another possibility is to decrease development time and cost, and to buy a better interconnection network. The most suited strategy depends on each use-case.

5.5 Conclusion

Due to the high hardware complexity of current processors, code optimization is mandatory for high performance computing codes. But an optimization process is,

in some sense, an endless process that may lead to important extra development costs for only small performance improvements. A methodology is required to avoid this type of pitfall.

In this chapter we have investigated the optimization process of a toy application (a dense matrix product) on a multicore CPU/GPU cluster. Throughout this process, some methodologies have been proposed to develop optimized computing kernels and efficient overlapping of communications with computations, and to identify the most interesting configurations and deployments on CPU or GPU clusters. Following these methodologies, different possible degrees of optimization have been presented and several series of criteria have been proposed to help developers decide up to which degree of optimization the development effort has to be led.

Our study shows that even on a basic test application, a significant increase in the code complexity (especially in GPU kernels) can be observed with the increase of the optimization degree, requiring more expertise to develop and maintain and leading to longer development times.

The variants of the application obtained have been fully benchmarked with different runtime parameters (when pertinent) and different configurations of the test platform. Those benchmarks have pointed out that the highest optimization degrees may sometimes be useless as they bring no visible gain. Moreover, the experiments have also shown that a strong limitation that quickly appears with optimized codes comes from the interconnection network. In fact, current classical networks such as Gigabit Ethernet are not suited to the interconnection of powerful nodes running optimized computing kernels.

In the near future, we plan to achieve complementary benchmarks with different problem sizes and with different clusters, in order to confirm the generality of our analysis and methodology. Moreover, we aim at studying other overlapping schemes as well as CPU kernels using vector units (like SSE or AVX ones).

References

[1] "Open Source High Performance Computing". `http://www.open-mpi.org`

[2] "OpenMP multi-threaded programming API". `http://www.openmp.org`

[3] F. Baude, D. Caromel, N. Furmento, D. Sagnol, "Overlapping communication with computation in distributed object systems", in P. Sloot, M. Bubak, A. Hoekstra, B. Hertzberger, (Editors), " High-Performance Computing and Networking", Lecture Notes in Computer Science, 1593, 744–753, Springer Berlin / Heidelberg, 1999. doi:10.1007/BFb0100635

[4] E.M. Daoudi, A. Lakhouaja, H. Outada, "Overlapping Computation/Communication in the Parallel One-Sided Jacobi Method", in H. Kosch, L. Böszörményi, H. Hellwagner, (Editors), "Euro-Par 2003 Conference", Lecture Notes in Computer Science, 2790, 844–849, Springer, 2003. doi:10.1007/978-3-540-45209-6_115

[5] L.D. de Cerio, M. Valero-García, A. González, "Overlapping Communication and Computation in Hypercubes", in L. Bougé, P. Fraigniaud, A. Mignotte, Y. Robert, (Editors), "Euro-Par '96 Conference, Vol. I", Lecture Notes in Computer Science, 1123, 253–257. Springer, 1996. doi:10.1007/3-540-61626-8_33

[6] G.I. Goumas, N. Anastopoulos, N. Koziris, N. Ioannou, "Overlapping computation and communication in SMT clusters with commodity interconnects", CLUSTER, 1–10, IEEE, 2009. doi:10.1109/CLUSTR.2009.5289174

[7] R.L. Graham, S.W. Poole, P. Shamis, G. Bloch, N. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, G. Shainer, "Overlapping computation and communication: Barrier algorithms and ConnectX-2 CORE-Direct capabilities", IPDPS Workshops, 1–8, IEEE, 2010. doi:10.1109/IPDPSW.2010.5470854

[8] T. Hoefler, A. Lumsdaine, "Message Progression in Parallel Computing - To Thread or not to Thread?", CLUSTER, 213–222, IEEE, 2008. doi:10.1109/CLUSTR.2008.4663774

[9] T. Hoefler, A. Lumsdaine, "Overlapping Communication and Computation with High Level Communication Routines", IEEE International Symposium on Cluster Computing and the Grid, 572–577, 2008.

[10] J.B. White, J.J. Dongarra, "Overlapping Computation and Communication for Advection on Hybrid Parallel Computers", IPDPS, 59–67, IEEE, 2011. doi:10.1109/IPDPS.2011.16

[11] T.H. Kaiser, S.B. Baden, "Overlapping communication and computation with OpenMP and MPI", Sci. Program., 9(2,3), 73–81, Aug. 2001. `http://dl.acm.org/citation.cfm?id=1239928.1239932`

[12] W. Kirschenmann, "Towards sustainable intensive computing kernels - Vers des noyaux de calcul intensif perennes", PhD thesis, Lorraine University, 2012. (in French)

[13] W. Kirschenmann, L. Plagne, S. Vialle, "Parallel SP_N on Multi-Core CPUs and Many-Core GPUs", Transport Theory and Statistical Physics, 39(2), 255–281, 2010.

[14] V. Marjanovic, J. Labarta, E. Ayguadé, M. Valero, "Overlapping communication and computation by using a hybrid MPI/SMPSs approach", in T. Boku, H. Nakashima, A. Mendelson, (Editors), "ICS", 5–16, ACM, 2010. doi:10.1145/1810085.1810091

[15] NVIDIA, "NVIDIA CUDA C Programming Guide 4.0", 2011. `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`

[16] A.K. Somani, A.M. Sansano, A.K. Somani, A.M. Sansano, "Minimizing overhead in parallel algorithms through overlapping communication/computation", Technical report, Institute for Computer Applications in Science and Engineering, 1997.