Efficient Pipelining of Level 3 BLAS Routines

Frédéric Desprez and Stéphane Domas

LIP, ENS Lyon
46 Allée d'Italie
F-69364 Lyon cedex 07
(desprez,sdomas)@ens-lyon.fr

Abstract. This paper presents a method that handles automatically a pipeline of level 3 BLAS routines, executed on different processors. It is based on linear graph of tasks (a single BLAS 3 call), which describes the dependences between the matrices used for computations. From such a graph and from a theoretical model of level 3 BLAS execution time, we determine the best blocking for each task and the optimal blocking size for the pipeline.

1 Introduction

Optimizing parallel algorithms is a difficult task, especially if we keep in mind the portability issues. However, several techniques exists to speed up parallel applications on distributed memory machines. Asynchronous communications are a way to improve the performances and the scalability of parallel routines. Using such communications, a processor can compute while communicating, reducing the total overhead of communications. Pipelining is an optimization technique that can be used when data dependences produce too much idle time. It consists in splitting a task in a certain number of sub-tasks and to communicate the result of one sub-task as soon as it has been computed, allowing the next processor to start sooner. The way tasks are split is of course dependent of many parameters (algorithm, network bandwidth and latency, computation speed, ...). It can often be computed or approximated to obtain the best performances.

Since many numerical applications are now using the standardized level 3 BLAS calls, our aim is to provide a general mechanism for the use of pipelined overlap when the result of a level 3 BLAS task has to be communicated. Because level 3 BLAS routines are already optimized for a broad range of processors, possible optimizations can be made in a succession of BLAS calls, executed on different processors. At the moment, we target linear tasks graphs where each task is executed on a different processor and where tasks can be split. We show how tasks have to be split and when.

Many numerical applications can benefit of such optimization. In [3], the authors describe mechanisms to parallelize applications with two levels of parallelism; coarse-grain task parallelism at the outer level and data-parallelism at the inner level. Their motivating example is an algorithm for the computation of eigenvalues of a dense non-symmetric matrix [1]. Sparse matrix factorization like the one described in [5] or [7] can also make use of these mechanisms. Those algorithms compute dense BLAS operations on different blocks with dependencies between the blocks (given by the sparse structure of the matrix). The problem in this last example is that dependencies are only known at run-time. However, the optimization we describe can be used but inside the run-time system or after the symbolic factorization.

One important part of the optimization process is the blocking of the computation routine. The goal is to keep the computation speed high and to reduce its grain. Several papers discuss the optimization of level 3 BLAS codes using loop partitioning techniques and efficient use of the memory hierarchy [4, 8]. The most recent works are inside the PHiPAC [2] and ATLAS projects [9].

In this paper, we present an overview of the problem for a motivating example. Then, we describe in detail the two main steps of our methodology. We finish by experimental results and an example of combination of our approach with data-parallelism.

2 Motivating Example

Our motivating example is a linear task graph made of five matrix product tasks (level 3 BLAS `DGEMM` routine) executed on five different processors. Dependencies between the tasks are given in Figure 1. The first arrow means that after its computation, matrix C is sent to the next processor as an entry of the next matrix product. Its a flow dependence (read after write).

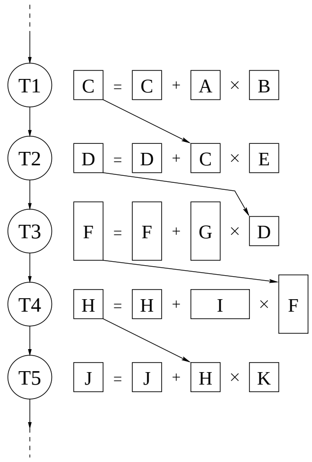

Fig. 1. Motivating Example: pipeline of 5 matrix product tasks.

This example is quite simple but it shows the main issues in the optimization of such a pipeline of level 3 BLAS routines. We have chosen the matrix product because it is used as a kernel operation of many linear algebra algorithms and there are many simple ways of blocking its loops. Moreover, other level 3 BLAS routines can be efficiently rewritten with `DGEMM` calls [8].

Because of the dependences between the tasks, the execution of such task graph can not be optimized without a split of the tasks. If different processors are chosen for the execution of each task, the overall parallel execution time is even worse than the sequential one because we add the cost of the communications between different processors.

3 Pipelining the motivating example

There are several ways to block the computations to pipeline the `DGEMM` tasks. Is is equivalent to split the matrices. In Figure 2, we give three different possibilities to block the matrices used in a multiplication. Every loop can be blocked but it may not lead to an efficient communication pipeline.

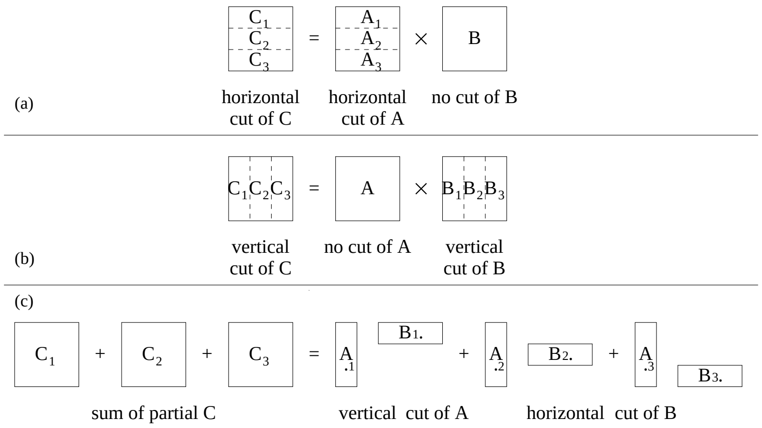

Fig. 2. Motivating Example: three possible blockings for the matrix product algorithm.

In the first two cases, we only block one matrix and compute vertical or horizontal blocks of C. This resulting algorithm is perfect for a pipeline since a task receives, computes and sends rectangular blocks. The third case is problematic since we compute square blocks which are intermediate results. This situation is best avoided but in our example, the dependencies between matrices have been chosen to bring such a case. We will show how to solve this problem .

3.1 Choosing a Blocking for the Computations

Taking into account the dependencies, we must find an efficient way to block the computations for each task in the graph. Instead of vertical or horizontal blocks, we will use more obvious notations. According to the BLAS reference guide [6], a `DGEMM` uses three matrices: $A_{(M\times K)}$, $B_{(K\times N)}$ and $C_{(M\times N)}$. In the

the first blocking in Figure 2, A is blocked along its M dimension. In the second solution, B is blocked along its N dimension. In the last solution, A and B are blocked along their K dimension. Thus, we will use the notations M-block, N-block and K-block for the three possibilities to block a `DGEMM` and more generally, each BLAS 3.

We will also use the notation S-block when the matrices are the sum of partial results like C in the third case. Meanwhile, an S-block `DGEMM` requires more than $O(MNK)$ floating operations to compute and thus, must be avoided.

3.2 Choosing a Blocking for the Communications

In our example, we always send a result matrix. Thus, the "orientation" (vertical or horizontal) of the blocks that are sent is determined by the blocking used for the computation. For example, if the first task is M-block, the result matrix C' is blocked horizontally and horizontal blocks must be sent. If it is not a result matrix that is sent, we have the choice for the orientation.

Since there is no preferred dimension for the communications, we use the notation H-block or V-block depending on the "orientation" of the blocks that are sent. We also use the notation S-block when a matrix is sent in partial results and $\oslash$-block when the matrix is sent in a whole.

3.3 Some Solutions to Block the Matrices

In Table 1, we present four ways of blocking computations and communications for the motivating example. For each solution, the left column gives the blocking used for the computations and the right column, the blocking for the communications.

	Task 1	**Task 2**	**Task 3**	**Task 4**	**Task 5**
Solution 1	O/O	O/O	O/O	O/O	O/-
Solution 2	M/H	M/H	K/S	S/S	S/-
Solution 3	M/H	M/H	K/O	M/H	M/-
Solution 4	M/H	M/O	N/V	N/V	K/-

Table 1. Four different blocking for our motivating example.

- **First solution:** no pipeline. Sequential execution.
- **Second solution:** Task 1 and 2 use M-block for the computations. Task 3 receives horizontal blocks in D. It implies that Task 3 is K-block and the result matrix F is S-block. Further tasks in the pipeline are also S-block.
- **Third solution:** it begins like the second solution but Task 3 sends the whole F matrix ($\oslash$-block communication) instead of an S-block communication. Then, Task 4 begins a "new" pipeline with M-block computations.
- **Fourth solution:** Task 1 and 2 are M-block for computations but Task 2 sends the whole D matrix. Then, Task 3 begins a "new" pipeline with N-block computations.

Assuming that each task needs 3 seconds to complete and we divide them in 3 (.i.e blocking size = 3). We also assume that each sub-task takes 1 second to complete, (which is not experimentally true). The Figure 3 shows a Gantt-like diagram for each solution. Rectangles are the tasks or the sub-tasks and the lines, the communications.

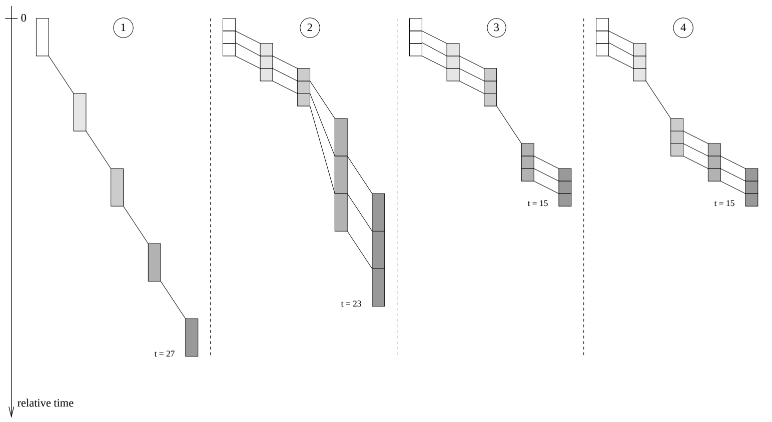

Fig. 3. Motivating example: Gantt chart for the four solutions.

- In the first column, all tasks are completed before sending the result to the next task in the pipeline. It is the no-pipeline, no-overlap solution and it takes 27 seconds to complete.
- The second solution takes a little less time than the no-pipeline solution since there are some S-block computations. Task 4 computes $F.I + H \to H$. Blocking F in 3, Task 4 mathematically computes $(F_1 + F_2 + F_3).I + H \to H$ which requires $3n^2 + n^3$ operations. But the F_i matrices are received sequentially. Thus, Task 4 computes $F_1.I + F_2.I + F_3.I + H \to H$ which requires $3n^2 + 3n^3$ operations. The same occurs for the Task 5 and it leads to an execution time in 23 seconds. It is better than the first solution but largely more than a sequential execution on a single processor (15 seconds).
- For the third and fourth solutions, a gain close to 2 can be reached despite of the chain of BLAS is divided in two pipelines.

4 Computing the optimal parameters for the pipeline

The motivating example points out that the pipeline efficiency greatly depends on the blocking for each matrix. Solutions 3 and 4 prove that there is a set of best blockings. Another parameter is the blocking size. A blocking size of 3 is surely not the optimal value for experimental tests, especially for solutions 3 and 4 where the two pipelines may have different blocking sizes. Choosing the blocking size too small leads to an inefficient pipeline. Choosing too big multiplies the communication latencies and reduces the BLAS 3 efficiency since the blocks used in computations are smaller

In order to fix these two parameters, our method consists in two steps :

- Finding the set of best blocking solutions.
- Finding the optimal blocking size(s) for each solution.

With the optimal blocking size(s), we can compute the total execution time of each pipeline and determine the best solution.

4.1 Finding the set of Best Blocking Solutions

The blocking of each task is very often determined by a previous one in the pipeline but sometimes, it can be chosen. In this case, we obtain several possible chains of blocking. A tree with all possibilities can be constructed but with an assumption and a condition, only the “best” solutions in the tree can be picked. Best means that these solutions lead to an efficient pipeline. The motivating example introduces the assumption and the condition.

In the second solution of the Figure 3, we see that S-block computations are propagated and reduce the efficiency of the pipeline. In the solution 3, we exchange the first S-block communication by an $\oslash$-block communication. Then, the pipeline is more efficient.

In the fourth solution, there are no K-block computations (except the last but it has no influence). Mathematically, a K-block computation always implies a S-block communication, which must be avoided. In this solution, each task sends a block that does not produce a K-block computation in the next task. If it is not possible, the task sends a $\oslash$-block matrix.

To summarize, we have:

- **NSP-assumption** (“No S-block Please !”): each time an S-block communication occurs, replace it by the communication of the whole matrix, i.e. delay the communication outside the loops.
- **NCB-condition** (“Next Computation Blocking”): sending horizontal or vertical blocks produces a K-block computation in the next task.

According to Table 1 and Figure 3, when the NCB-condition occurs, we have the choice to send horizontal or vertical blocks as in solution 3, or to send the whole matrix as in solution 4. Indeed, the third and the fourth solution are equally efficient thus we cannot decide when it is interesting to send the whole matrix and when it is not. Consequently, two solutions must be generated.

The algorithm that constructs the set is given in Figure 4. The computation and communication blocking of the i^{th} task are noted $BL_i = [comp_i; comm_i]$ (for example, $[M; H]$). Each solution is a set of BL_i, constructed by iterating on the tasks, taking account of the dependencies, the type of BLAS 3 and testing if NSP-assumption or NCB-condition occurs.

```
nb_sol = 1    /* number of solutions */
Sol = ∅    /* set of solution is empty */
For i = 1 to nb_task − 1 do
  For j = 1 to nb_sol do
    Find BL_i from description of task_i and BL_{i−1}.
    If (comm'_i = S) || (comm'_i = ⊘) then
      Use NSP-assumption→ comm'_i = ⊘.
      Concat BL_i with Sol_j.
    Else
      Concat BL_i with Sol_j.
      If (NCB-condition applies) then
        /* create a new solution */
        cut'_i = BL_i
        comm'_i = ⊘
        Sol_{nb_sol+1} = Sol_j.
        Concat cut'_i with Sol_{nb_sol+1}.
        nb_sol = nb_sol + 1.
      Endif
    Endif
  Endfor
Endfor
```

Fig. 4. Algorithm to construct the set of the best solutions.

4.2 Computation of the Optimal Blocking Size

We have seen in the motivating example that it is mandatory to know the execution time of each task to determine the best blocking solution. We have also supposed that the tasks were split in 3 sub-tasks. In fact, the blocking size very influences the efficiency of the pipeline. A matrix multiplication split in L partial computations is longer than the the computation performed in one time. But, the execution time of the pipeline can only be determined if the blocking size is fixed. Our goal is to compute automatically the optimal blocking size L for each solution.

Theoretical Level 3 BLAS Execution Time All level 3 BLAS subroutines are matrix-matrix operations and thus compute $O(n^3)$ floating point operations for $O(n^2)$ memory accesses. More generally, the execution time of a level 3 BLAS depends on the size of the matrices which are involved in the computation. Let $A_{(M \times K)}$, $B_{(K \times N)}$, and $C_{(M \times N)}$ be the three matrices used by a level 3 BLAS. We assume that the execution time of a level 3 BLAS is given by the expression:

$$T_{BLAS3}(M,N,K) = aMNK + bMN + cMK + dNK + eM + fN + gK + h \quad (1)$$

where $a \ldots h$, parameters that depend on the algorithm used in the routine and on the target machine used. These parameters can be found by an interpolation on experimental tests. Unfortunately, we have noticed that they vary as the protocol of the tests vary (what increasing step for the matrix size, what dimension increases...). In order to have consistent parameters, it is necessary to have specific expressions of level 3 BLAS execution time for our problem.

According to the `BLAS` syntax, a `DGEMM` ($\alpha A.B+\beta C \rightarrow C$) can be M-block, N-block or K-block. The expression 1 does not take care of the blocking dimension. Consequently, we take a different expression for each case. For example, the expression for an M-block computation of `DGEMM` is :

$$T^{M}_{GEMM}(M,N,K) = (a_M M + b_M)NK + (c_M M + d_M)\sqrt{NK} + (e_M M + f_M)$$

Furthermore, the matrices can be transposed. It implies that the parameters $a_M, b_M, \ldots$ are also different for each possibility of transposition. For `DGEMM`, there are four possibilities thus, a total of 12 set of parameters.

Execution Time of a Level 3 BLAS Cut in *L* Blocks In the pipeline, a BLAS 3 is computed in several times. What is its execution time when it is cut in L sub-computations along the D dimension ? We assume that the blocks must almost have the same width in order to have a regular pipeline. Choosing a single size may penalize the pipeline efficiency because the last block may be to large or to small. We prefer to choose two sizes that differ only by one.

Let $r = D \text{mod} L$. There will be $L-r$ blocks of width $\left\lfloor \frac{D}{L} \right\rfloor$ and r blocks of width $\left\lceil \frac{D}{L} \right\rceil$. Assuming that $D = M$, the execution time of a D-block level 3 BLAS cut in L blocks becomes:

$$\begin{aligned} T^{M}_{BLAS3}(M,N,K,L) = \\ &(L-r)\left[(a_M \left\lfloor \tfrac{M}{L} \right\rfloor + b_M)NK + (c_M \left\lfloor \tfrac{M}{L} \right\rfloor + d_M)\sqrt{NK} + (e_M \left\lfloor \tfrac{M}{L} \right\rfloor + f_M)\right] + \\ &r\left[(a_M \left\lceil \tfrac{M}{L} \right\rceil + b_M)NK + (c_M \left\lceil \tfrac{M}{L} \right\rceil + d_M)\sqrt{NK} + (e_M \left\lceil \tfrac{M}{L} \right\rceil + f_M)\right]\end{aligned}$$

It gives after simplification:

$$T^{M}_{BLAS3}(M,N,K,L) = (a_M M + b_M L)NK + (c_M M + d_M L)\sqrt{NK} + (e_M M + f_M L)$$

$$T^{M}_{BLAS3}(M,N,K,L) = L \times T^{M}_{BLAS3}(\tfrac{M}{L}, N, K)$$

Thus, the theoretical execution time of a level 3 BLAS split in L blocks is a linear function of L. Meanwhile, experimental results show that it is partially true. In fact, the $a_{M_{nn}}$, $b_{M_{nn}}$,... parameters are defined by intervals. Figure 5 gives the experimental execution time of a M_{nn} `DGEMM`, as a function of M. We can see that there are at least two intervals ($[2,11], [12,128]$) in which the parameters are different. For an accurate simulation, we define a lot of intervals for small matrix sizes because a small number of floating operations does not allow a good utilization of the arithmetic unit pipeline. But it very depends on how the level 3 BLAS is implemented and which processor is used.

In order to compute parameters on a large number of intervals, we have implemented a routine that produces sets of experimental curves for each BLAS

3, computes automatically the intervals and the parameters defined on, and finally, outputs a C code containing array initializations.

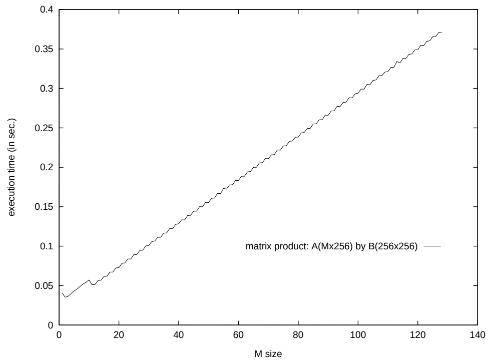

Fig. 5. Execution time of a `DGEMM` as a function of dimension M.

Execution Time of a Pipeline The next step is to fix the optimal blocking size L to determine the total execution time of each solution. In fact, there may be several pipelines for a single graph, like solutions 3 and 4 of the motivating example. Thus, several optimal blocking sizes must be computed to obtain the total execution time of the graph.

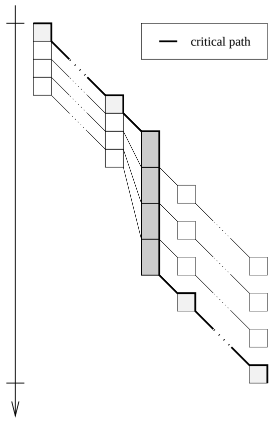

Fig. 6. Critical path.

For each pipeline, we define a critical task and a critical path. If there are N tasks in the graph, the critical path is an execution path in the Gantt diagram which crosses $N-1$ sub-tasks, the critical task and $N-1$ communications. A general example is given in Figure 6. The bold line is the critical path. All the dark gray blocks form the critical task which is the longest task in the pipeline and is totally crossed by the critical path. We also suppose that this task has enough space for communication buffers, not to wait to send a message between its sub-task. For other tasks, only one of their sub-tasks is crossed by the critical path. The length of the critical path gives the total execution time of the pipeline.

From equations given in Section 4.2, we obtain the execution time of the critical task and each sub-task crossed by the critical path. The sum of all gives the execution time of the pipeline as a polynomial function of the blocking size L. With a derivative, we find the optimal blocking size. Another way is to make an exhaustive research of the optimal blocking size, testing all possible values of L. The second method is longer but leads to better results because all parameters are taken in the execution time computation. With a derivative, some parameters are removed.

5 Library Interface

With the optimal blocking size of each pipeline of each solution, we determine the best solution and then, have all keys to implement an efficient pipeline. Meanwhile, this implementation may be hard for someone who is not familiar with the BLAS 3 syntax and communications libraries such as PVM, MPI or BLACS. For a convenient use of our method, we have provided an interface which handles at runtime and transparently the pipeline(s) of any linear task graph of level 3 BLAS. It consists in 5 main routines that a user has to call. The Table 2 gives the C code that implements a pipeline for motivating example.

```
NewPipe(5);
InitTask(0,DGEMM,P_nn,i,i,i,1.2,A,LDA,B,LDB,1.3,C,LDC,NO,MCC,NO,NO,0,1);
InitTask(1,DGEMM,P_nn,i,i,i,1.2,A,LDA,B,LDB,1.3,C,LDC,MA,MCC,0,0,0,2);
InitTask(2,DGEMM,P_nn,2*i,i,i,1.2,A,LDA,B,LDB,1.3,C,LDC,MB,MCC,0,1,0,3);
InitTask(3,DGEMM,P_nn,i,i,2*i,1.2,A,LDA,B,LDB,1.3,C,LDC,MB,MCC,0,2,0,4);
InitTask(4,DGEMM,P_nn,i,i,i,1.2,A,LDA,B,LDB,1.3,C,LDC,MA,NO,0,3,NO,NO);
InitPipe();
RunPipe();
FreePipe();
```

Table 2. Motivating example : C code to handle the pipeline at run-time.

The description of each task is done by `InitTask(...)` calls. Parameters are the name of the BLAS 3, the matrices and their sizes, which matrices are sent, ... `InitPipe()` computes the best solution and the optimal blocking size. `RunPipe()` execute the pipeline.

This interface has three advantages. First, it is very simple. The user just have to describe the task graph. He does not need special knowledge on what is a pipeline and how to implement it. The execution is also transparent and no calls to communication routines is necessary. Secondly, it is portable. The communications are based on the BLACS and the computations on the BLAS. In third, all is handle at runtime. Thus, the initialization of the task graph can dynamically parameterized.

6 Experimental results

Our library gives solution 4 as the best solution for the motivating example. This result is logical since tasks 3 and 4 work on larger matrices. It is obvious that pipelining the biggest tasks together (solution 4) is more accurate than pipelining them separately (solution 3).

In order to validate our method, we have also tested solution 3. The Figure 7 presents the execution time of solutions 1 (no pipeline),3 and 4, on our cluster of Pentium, using the BLACS over MPI-LAM for communications. The time is given for the optimal blocking size. The Figure 8 shows the gain for solutions 3 and 4. The gain is the execution time of these solutions divided by the execution time of solution 1. It clearly shows that solution 4 is better than 3. Furthermore, a gain greater than 2 can be reached despite the graph is broken into two pipelines.

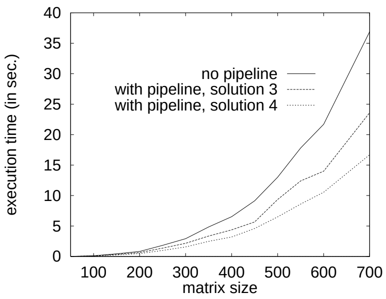

Fig. 7. Motivating example: execution time on a cluster of Pentium.

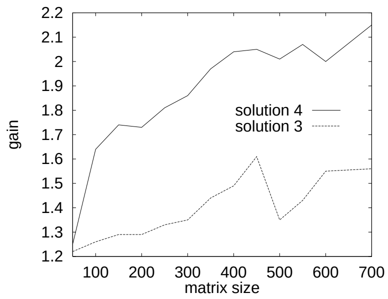

Fig. 8. Motivating example: gain on a cluster of Pentium.

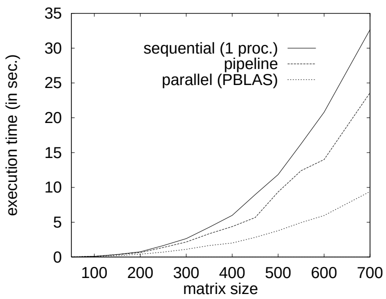

Fig. 9. Motivating example: comparison between sequential, parallel and pipeline execution time on a cluster of Pentium (POPC).

In Figure 9, we give a comparison between three different execution times on the cluster of Pentium. The first one is the purely sequential time : only one processor computes the five DGEMM. The second one is the pipelined version time. The last one is the purely parallel time : no pipeline but all processors computes each BLAS 3 with a PBLAS routines.

We notice that the pipelined version is less efficient than the PBLAS version. It clearly shows that breaking the task graph into several pipelines decreases performances. Meanwhile, the PBLAS approach has two drawbacks. First, it is not so easy to use and requires some advanced knowledge in message passing implementation style. Secondly, it is more complicated to distribute the matrices. Thus, our method is a good compromise between efficiency and complexity of use.

We have also tested our method on other examples and machines (Paragon, cluster of PowerPC,...). For example, with 5 identical `DGEMM`, we reach a gain of 3.85, on the cluster of Pentium. More generally, we have notice that such a graph (identical BLAS with a single pipeline) with n tasks always leads to a gain near form $n-1$.

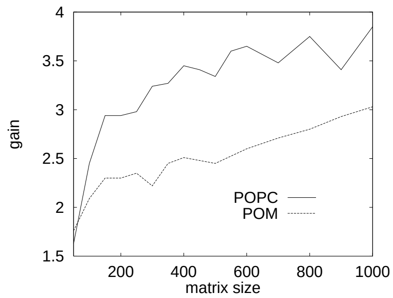

Fig. 10. 5 identical `DGEMM`: gain on clusters of Pentium (POPC) and PowerPC (POM).

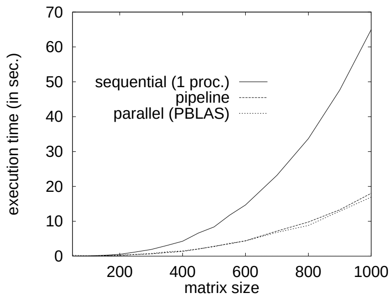

Fig. 11. 5 identical `DGEMM`: comparison between sequential, parallel and pipeline execution time on a cluster of Pentium (POPC).

In figure 11, we give a comparison between sequential, pipeline and parallel execution times (as in Figure 9). This time, the pipeline version is as efficient as the parallel version and much more than a sequential execution. This result is identical for other tests with a single pipeline even if there are different BLAS 3 in the graph. It proves that our method is as efficient as PBLAS/ScaLAPACK in such cases.

7 Mixing our approach with data-parallelism

We have seen in Figures 9 and 11 that pipelining is always better than doing computations sequentially on a single processor. Meanwhile, the parallelism (implicitly, the number of processors) strictly depends on the number of computations in the graph. It seems to be a limitation if we consider that the problem size may increase beyond the resource of the processors. This is not a problem for a PBLAS computation since we can use a variable number of processors.

In fact, the dynamism of our method allows to mix it with data-parallelism. Especially in block algorithms, there are dependencies between sequential computations on different processors. In some case, these dependencies can be expressed as a set of linear graph of task and each graph uses a separate set of processor. For example, we can construct a block algorithm with such dependencies for the multiplication of two matrices, distributed by block (cyclically or not) on a grid of processor. In all these cases, our method can be used whatever the number of processors is since our interface allows to define the graph dynamically.

We have implemented the matrix multiplication algorithm and compared it to that proposed in the PBLAS: `PDGEMM`. It uses a simple full-block distribution vs. block scattered for `PDGEMM`). The algorithm is nearly similar to the `PDGEMM` routine except that we replace the broadcast at the beginning of each step by a pipeline for each processor on the same row. Thus, all processors are working on the same amount of data and it leads to a perfect load balance. If the blocking size is well chosen, the idle time due to contentions is minimal.

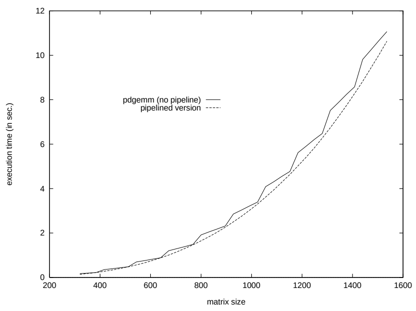

Fig. 12. Execution time of `PDGEMM` and the pipelined matrix multiplication on 16 processors of the Paragon.

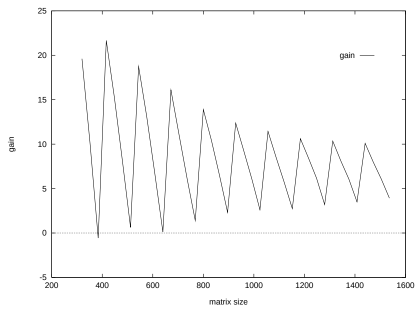

Fig. 13. Gain of the pipelined matrix multiplication over `PDGEMM`.

First, we notice in Figure 12 that the execution time of `PDGEMM` is not perfectly cubic. This comes from a slight load imbalance for some matrix sizes. Our version does not have this problem and thus, gives largely better execution time in these cases.

Secondly, our routine has always better results for large matrices. For such matrices, the average gain of our routine, given in Figure 13, is 7%. In fact, there are a lot of broadcast and global sum communications in `PDGEMM` and the processors are often idle. In our routine, there are only four steps of computations (4×4 processors), thus only four global sum and three pipeline phases. During these phases, the processors are communicating while they are computing: there are less idle than in `PDGEMM`.

8 Conclusion

In all cases, the experimental results have confirmed the usefulness and efficiency of the pipeline in tasks graph over a sequential execution. With asynchronous communications, the total execution time of the graph can be reduced by two or even more, depending on the size of the graph and the dependencies between the tasks. Moreover, experimental results have validated our methods to compute the set of best blockings and the optimal blocking size. When the dependencies leads to a single pipeline, our method gives a code as efficient as a message passing approach but much easier to implement. Ten lines of code are sufficient, against hundred for a PBLAS/ScaLAPACK code.

Finally, some real applications can be optimized by our method. We have already give some of them in the introduction but it also works each time the dependencies between sequential computations produce a set of linear graph of tasks, with each graph working on different processors.

References

1. Z. Bai and J. Demmel. Design of a Parallel Nonsymmetric Eigenroutine Toolbox - Part I. In SIAM, editor, *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, 1993. [Long version available as UC Berkeley Computer Science report all.ps.Z via anonymous ftp from tr-ftp.cs.berkeley.edu, directory pub/tech-reports/csd/csd-92-718.].
2. J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *Proceedings of the International Conference on Supercomputing*, Vienna, Austria, July 1997. ACM SIGARC. Also LAPACK Working Note 111, UTK, `http://www.netlib.org/lapack/lawns`.
3. S. Chakrabarti, J. Demmel, and K. Yelick. Models and Scheduling Algorithms for Mixed Data and Task Parallel Programs. *Journal of Parallel and Distributed Computing*, 47:168–184, 1997.
4. M.J. Dayde, I.S. Duff, and A. Petitet. A Parallel Block Implementation of Level 3 BLAS for MIMD Vector Processors. Technical Report RT/APO/93/1, ENSEEIHT - Département Informatique N7 -I.R.I.T. Gpe Algorithmes Parallèles, 1992.
5. J.W. Demmel, J.R. Gilbert, and X.S. Li. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination, February 1997. [Available via ftp in `ftp://parcftp.xerox.com/pub/gilbert/parlu.ps.Z`].
6. J.J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transaction on Mathematical Software*, 16(1):1–17, 1990.
7. A. Gupta, F. Gustavson, M. Joshi, G. Karypis, and V. Kumar. Design and Implementation of a Scalable Parallel Direct Solver for Sparse Symmetric Positive Definite Systems. In *Proceedings of the Eighth SIAM Conference on Parallel Processing*, March 1997. [Also available in `ftp://ftp.cs.umn.edu/dept/users/kumar/WEB/papers.html`].
8. B. Kågström, P. Ling, and C. Van Loan. GEMM-Based Level 3 BLAS: High Performance Model Implementations and Performance Evaluation Benchmark. Technical Report UMINF-95.18, Umeå University, October 1995.
9. R. Whaley and J. Dongarra. LAPACK Working Note 131: Automatically Tuned Linear Algebra Software. Technical Report CS-97-366, The Universtity of Tennessee - Knoxville, 1997. `http://www.netlib.org/atlas/`.