On Optimization of Parallel Communication-Avoiding Codes for Multicore Architectures

Emna Hammami, Yosr Slama
University of Tunis El Manar, Faculty of Sciences of Tunis, University Campus - 2092 Manar II, Tunis, Tunisia

Abstract - Parallelizing polyhedron programs (PP) i.e. programs structured in nested loops with affine bounds is still a matter of focus for a lot of research works due to its practical interest in scientific applications. Indeed several efficient code generation tools (CGT) from PP’s have been proposed in the literature such as Pluto. Despite its performance, these CGT’s rarely take into consideration the target architecture (TA) which may be a source of performance improvement. We precisely address this aspect i.e. adapting code generation from a PP to a given TA, namely a multicore machine in order to enhance the efficiency of the generated parallel communication-avoiding codes. For this purpose, we propose a two-phase approach. The first consists, for given TA and PP, in fairly distributing the parallel tasks involved by the PP onto the TA cores. The second improves data locality through cache memory optimization. In order to evaluate the interest of our contribution, we carried out a series of experiments targeting a quadcore bi-processor machine and choosing three PP’s often encountered in scientific applications, namely matrix addition, matrix-vector product and matrix-matrix product.

Keywords: Allocation, cache memory, code optimization, multicore architecture, polytope model, Prefetching.

1 Introduction

Parallel computing may be performed on different parallel platforms. Recently, many studies have focused on multicore machines which are nowadays the most used components of supercomputers, clusters and grids. Following the recent architectures development, it becomes necessary to adapt existing parallelization methods and tools to the new architectures by taking into account the target machine (TA) characteristics, e.g. cache memory organization and size, cores’ speed, etc. It is in this general context that our work focuses especially on polyhedron programs (PP) i.e. programs structured in nested loops with affine bounds, which are the most used codes in scientific computing. Since code generation (CG) is the last phase of automatic parallelization based on the polytope model, we are interested in parallel generated codes from PP’s. In spite of the performance of the most known CG tools (CGT) such as Pluto, these latter don’t take into account all the TA intrinsic characteristics which may be a rich source of performance enhancement. Thus, we take a special interest in the generated code optimization

based on some hardware characteristics of the TA in order to improve the completion time. In particular, we aim at optimizing parallel generated communication-avoiding codes. The remainder of the paper involves four sections organized as follows. In section 2, we present a brief state-of-the art on the most known optimization tools for parallel codes incorporated in Pluto. Section 3 is devoted to the description of our general two-phase approach, where we detail an experimental study on a multicore parallel platform that allows our theoretical contribution validation and evaluation. We finally conclude and present some perspectives in section 4.

2 Brief state-of-the art

In this section, we present a summary on existing code optimization tools related to the Pluto compiler [1].

In fact, we find in the literature many code optimization methods, particularly for parallel processing. This optimization may be seen as aiming at reducing either the code size or the run time in order to increase the overall performance. In our work, we mainly address the second point i.e. improving run time. Clearly, such optimization is quite important for codes automatically generated by CGT’s. Indeed, these latter, though powerful in parallel code generation, are sometimes inefficient from an optimization point of view because of their generality and the overheads they induce.

In this paper, a special interest is attached to the Pluto compiler [1] because of the high-level optimizations it achieves by means of the code optimization tool Cloog [2] and the tiling technique. More precisely, Pluto transforms C programs from source to source for coarse-grained parallelism and data locality simultaneously for generating parallel C OpenMP codes [1]. It uses a scheduling algorithm which tries to find affine transformations allowing the most efficient tiling.

Cloog [2] is a free software which generates code for scanning Z-polyhedra. It is designed to build control automata for high-level synthesis and find the best polynomial approximation of a given function. Based on Quilleré’s method [3], Cloog helps to solve scanning Z-polyhedra matters [2]. It was originally written to solve code generation
problems for optimizing compilers based on the polytope model. It allows avoiding control overhead and producing effective codes in a reasonable amount of time when there is full control on generated code quality. Besides, it performs dependency analysis and provides scattering functions which remove redundant constraints and specify better transformations reordering. These latter transforms the original polyhedron, by applying a new lexicographic order, into an equivalent target polyhedron [4].

Tiling which is a loop transformation that decomposes the whole computation set into subsets of smaller computation blocks, is especially used in vectorization, coarse-grained parallelism, and also in many high-level program optimizations such as data locality. Following old tiled loop generation methods with fixed tile sizes e.g. the approaches of Goumas and al [5], Kelly and al. [6], the SUIF [7] tool and the method of Ahmed and al. [8], new tiling techniques have been developed. These latter use parameterized and multi-level tiled loop generation methods where the block size is not fixed at compile time, but remains a symbolic constant. Hence, it may be changed even at runtime. Among such methods, we can mention the PrimeTile tool of Hartono and al. [9], the techniques of Baskaran and al. [10] and those of Kim [11]. Let us notice that the tiled code generation scheme of Pluto, which initially used fixed tile sizes, was extended to use parametric tiling by incorporating the tiled code generation with parametric tile sizes within the polyhedral model addressed by Renganarayanan and al. [12] in order to make it convenient for iterative and empirical optimization.

To conclude, we notice that most of these optimization techniques, though effective in parallel code generation, don’t care of the most intrinsic characteristics of the TA. Subsequently, through this paper, we aim to present our proposal for optimizing the quality of the code generated from Pluto by taking into account some physical characteristics of the TA. For this purpose, based on a selective study of hardware architecture detection tools, we have chosen the Hwloc package [13]. Indeed, it is able to supply generic and complete vision of hardware architecture. The provided data are more detailed relatively to the other methods. It generates a portable abstraction of the hierarchical topology of modern architectures with gathering information about NUMA memory nodes, sockets, shared caches, cores and simultaneous multi-threading.

3 Proposed approach

In this section, we’re going to present the general principle of our contribution as well as the following two steps: (i) distribution of the parallel tasks onto the cores of a target architecture (TA) and (ii) cache use optimization. A series of experiments were achieved on the chosen TA in order to evaluate the interest of our contribution.

### 3.1 General principle

Our aim is to link the software topology of the parallel code generated by Pluto [1] to the target hardware topology in order to optimize completion time. Thus, we have taken into consideration some hardware features e.g. cache memory size, cache sharing, cores organization on nodes, etc. In this work, as it is shown in Fig. 1, given a parallel program (not adapted to any architecture and generated by Pluto) and a target multicore architecture, we propose the following approach: After detecting the architecture hardware characteristics using the Hwloc framework [13] and the Cpuinfo library, we had developed a software component allowing code adaptation to the TA. So, we propose two steps towards this adaptation consisting, first, in distributing the different parallel tasks on the various available cores, and second in optimizing the cache memory use.

In the following subsections, we’ll first study various ways for distributing parallel tasks on the cores and we’ll choose the best allocation through an experimental study. Secondly, we’ll propose a method to improve the quality of a parallel generated code by taking into account some cache memory characteristics of the TA. This improvement will be evaluated through a series of experiments.

Let us mention that our multicore TA is an Intel ® Xeon dual quadcore processor CPU E5420 @ 2.50GHz. Its eight cores have dedicated L1 cache with 32 KB size, and share in pairs the L2 cache level whose size is 6MB. The RAM size is 4 GB.

Our target parallel programs don’t require any core communication since the iterations (tasks) are independent. This feature may be called communication-free (or more precisely communication-avoiding). We chose three benchmarks codes: (i) matrix addition (MA), (ii) matrix-vector product (MVP) and (iii) matrix-matrix product (MMP) where N denotes the size of the processed matrices.

The two proposed improvement phases were implemented in an automatic tool which takes as input a parallel communication-avoiding code generated by Pluto (CGP) and then generates an equivalent code undergoing the optimization that will be more detailed in the remainder.
3.2 Distributing the parallel tasks on cores

Taking into account the target architecture (TA), there are several factors which have a direct impact on run-time. Particularly, changing the way to distribute threads on the different cores may reduce data exchanges and thus improve data locality. For this reason, it is important to appropriately choose the allocation of parallel tasks on the available cores. We therefore examined some parallel code allocation methods expressed by the OpenMP library. In fact, the distribution of parallel iterations (tasks) on physical cores may be automatized by specifying the chunk size (CHS) i.e. the number of iterations constituting a thread. The chunks may be computed then allocated to threads in compile time.

In this case, the OpenMP task allocation may be performed either statically or dynamically. A static allocation means that iteration blocks divided into chunks are statically mapped to the execution threads in a round-robin manner. However, for the dynamic allocation, once a thread is available, it requests a chunk of iterations to execute.

Another manner in task distributing on cores is the OpenMP sections directive. Each section is assigned to one thread and each thread (TH) may be explicitly executed by one core. To illustrate this distribution manner, let us consider the following example consisting of a single parallel loop (the loop body may be a perfect nest not only a single iteration):

\[
\text{FORALL } (i = 1, N) \ \{ \text{S}(i) \} /*\text{loop body}*/
\]

\[
\text{ENDFORALL}
\]

In our case, we run as many sections as the cores number (i.e. 8) as follows:

- **Section 1**: \{ TH 0 : (i = 1, N/8 ) \} Allocate (TH 0; Core0) 
- **Section 2**: \{ TH 1 : (i = (N/8)+1, N/4) \} Allocate (TH 1; Core1) 
- **Section 3**: \{ TH 2 : (i = (N/4)+1, 3N/8) \} Allocate (TH 2; Core2) 
- **Section 4**: \{ TH 3 : (i = (3N/8)+1, N/2) \} Allocate (TH 3; Core3) 
- **Section 5**: \{ TH 4 : (i = (N/2)+1, 5N/8) \} Allocate (TH 4; Core4) 
- **Section 6**: \{ TH 5 : (i = (5N/8)+1, 6N/8) \} Allocate (TH 5; Core5) 
- **Section 7**: \{ TH 6 : (i = (6N/8)+1, 7N/8) \} Allocate (TH 6; Core6) 
- **Section 8**: \{ TH 7 : (i = (7N/8)+1, N) \} Allocate (TH 7; Core7) 

An experimental study covering on the one hand static and dynamic automatic allocation, and on the other hand manual allocation (sections) was carried out in order to compare them and choose the most efficient.

Three chunk types in the automatic case were studied by varying the sizes i.e.:
- X1 = cache line size
- X2 = cache size
- X3 = cache size / 2
- X4 = problem size (iterations number) / the cores number

X is considered ST (resp. DY) if the automatic allocation is static (resp. dynamic).

We detail experimental comparisons carried out on our TA and illustrated through some excerpts of our performed tests in Tables I, II and III, and Fig. 2 for the three benchmarks MA, MPV and MMP. We mention that the following notations are adopted in these tables in order to indicate the execution time of the studied codes:

- PLUTO : code generated by Pluto (CGP)
- STi : CGP with a CHS equal to Xi (i=1…4, see above) when the allocation is static
- DYi : CGP with a CHS equal to Xi (i=1…4) when the allocation is dynamic
- TILE : CGP undergoing the tiling technique adopted by Pluto
- Section : CGP with manual allocation (fair sections)
- Execution times : T1 for MA (ms), T2 for MVP (ms), T3 for MMP (s)
- Reducing time ratio r (%): rj(Version)=(Tj(PLUTO)-Tj(Version)) / Tj(PLUTO)

The various algorithm versions were coded in C under Linux. For the parallel experiments, we used the shared memory OpenMP environment. We point out that for each benchmark; we chose 10 values of N in the range [500, 5000] with a step equal to 500. For each N, the execution time is the mean of five runs. We therefore achieved 330 tests in total (110 for each benchmark). Excerpts of the results we obtained are depicted below.

<table>
<thead>
<tr>
<th>N</th>
<th>PLUTO</th>
<th>ST1</th>
<th>ST2</th>
<th>ST3</th>
<th>ST4</th>
<th>DY1</th>
<th>DY2</th>
<th>DY3</th>
<th>DY4</th>
<th>TILE</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>116.783</td>
<td>118.135</td>
<td>14.743</td>
<td>15.660</td>
<td>48.610</td>
<td>113.249</td>
<td>15.325</td>
<td>15.783</td>
<td>111.800</td>
<td>8.349</td>
<td>7.779</td>
</tr>
<tr>
<td>3000</td>
<td>983.732</td>
<td>954.530</td>
<td>108.772</td>
<td>108.650</td>
<td>480.052</td>
<td>1049.271</td>
<td>109.783</td>
<td>97.027</td>
<td>75.970</td>
<td>45.061</td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>1799.962</td>
<td>1750.715</td>
<td>194.027</td>
<td>192.424</td>
<td>982.679</td>
<td>1809.196</td>
<td>192.318</td>
<td>193.677</td>
<td>1730.625</td>
<td>142.906</td>
<td>74.757</td>
</tr>
<tr>
<td>5000</td>
<td>2832.404</td>
<td>2704.928</td>
<td>304.036</td>
<td>302.762</td>
<td>2825.001</td>
<td>116.783</td>
<td>303.613</td>
<td>305.762</td>
<td>2682.211</td>
<td>209.901</td>
<td>138.349</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>PLUTO</th>
<th>ST1</th>
<th>ST2</th>
<th>ST3</th>
<th>ST4</th>
<th>DY1</th>
<th>DY2</th>
<th>DY3</th>
<th>DY4</th>
<th>TILE</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>501.556</td>
<td>256.033</td>
<td>42.869</td>
<td>42.878</td>
<td>484.484</td>
<td>247.971</td>
<td>43.953</td>
<td>42.898</td>
<td>498.401</td>
<td>29.642</td>
<td>6.571</td>
</tr>
<tr>
<td>3000</td>
<td>1159.214</td>
<td>565.293</td>
<td>100.598</td>
<td>96.345</td>
<td>1058.303</td>
<td>540.607</td>
<td>96.287</td>
<td>96.318</td>
<td>1097.031</td>
<td>66.422</td>
<td>22.584</td>
</tr>
<tr>
<td>5000</td>
<td>3097.488</td>
<td>1375.272</td>
<td>269.459</td>
<td>272.383</td>
<td>2979.059</td>
<td>1592.616</td>
<td>268.522</td>
<td>268.374</td>
<td>3009.494</td>
<td>183.488</td>
<td>50.349</td>
</tr>
</tbody>
</table>

TABLE I. EXECUTION TIME (MS) OF MA FOR DIFFERENT ALLOCATIONS

TABLE II. EXECUTION TIME (MS) OF MVP FOR DIFFERENT ALLOCATIONS
TABLE III. EXECUTION TIME (S) OF MMP FOR THE DIFFERENT ALLOCATIONS

<table>
<thead>
<tr>
<th>N</th>
<th>PLUTO</th>
<th>ST1</th>
<th>ST2</th>
<th>ST3</th>
<th>ST4</th>
<th>DY1</th>
<th>DY2</th>
<th>DY3</th>
<th>DY4</th>
<th>TILE</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td>2262.674</td>
<td>1962.621</td>
<td>386.629</td>
<td>386.618</td>
<td>2054.293</td>
<td>2035.842</td>
<td>386.889</td>
<td>386.897</td>
<td>2054.293</td>
<td>386.889</td>
<td>2035.842</td>
</tr>
<tr>
<td>4000</td>
<td>2262.674</td>
<td>1962.621</td>
<td>386.629</td>
<td>386.618</td>
<td>2054.293</td>
<td>2035.842</td>
<td>386.889</td>
<td>386.897</td>
<td>2054.293</td>
<td>386.889</td>
<td>2035.842</td>
</tr>
<tr>
<td>5000</td>
<td>10629.747</td>
<td>9198.934</td>
<td>1798.107</td>
<td>1800.011</td>
<td>9320.983</td>
<td>9353.020</td>
<td>1798.316</td>
<td>1798.423</td>
<td>9320.983</td>
<td>9353.020</td>
<td>1798.316</td>
</tr>
</tbody>
</table>

Fig. 2. Experimental comparison of the different allocations of: (a) MA, (b) MVP, (c) MMP

From Tables I, II and III, we noticed that automatic allocation of OpenMP tasks performed either statically or dynamically is less efficient than the manual one. Indeed, it’s due to the generation of some overheads. Fig. 2 shows that the best inherent allocation is expressed in “Section” version for all the studied benchmarks. This improvement is more clarified in Table IV where we notice that the reducing time ratios (r) are high and vary in the range 93.33 (N=1000) - 95.84 % (N=4000) for MA, 96.84 (N=1000) - 98.68 % (N=2000) for MVP, and 95.48 (N=3000) - 97.90 % (N=1000) for MMP. Remark that the variations of r are not regular in terms of N. We expect that the improvement would continue for larger N. Let us add that with ST2, ST3, DY2, DY3 and TILE, we also obtain an improvement but less than with Section. Notice finally that a negative r corresponds to an increase in execution time. According to the comparative study below (see Fig. 2 and Table IV), we hence choose the manual allocation with fair sections. Subsequently, we decided in the tool that we have implemented to produce in the generated code as many sections as available cores (manual allocation).

TABLE IV. REDUCING TIME RATIOS R(%)
Remark: A negative ratio corresponds to an increasing time (e.g. 
\( r(\text{ST1}) = -1.15 \) for MA when \( N=1000 \)).

### 3.3 Optimizing the cache memory use

Typically, when the core-processor needs to read or to write a data, it fetches it first in the cache memory which avoids access to the main memory. If the cache memory has a copy of the data required by the processor, this is called a cache hit, otherwise this is called a cache miss. Cache memory is in fact categorized in levels that describe its closeness and accessibility to the processor. Level 1 (L1), which is extremely fast but of relatively small size, is located close to the processor and used for temporary instructions and data storage. Level 2 (L2) cache, located half-way between the processor and the system bus, is fairly fast and medium-sized. Level 3 (L3) cache is relatively large and close to the RAM.

Some cache memory levels can be shared between cores. This sharing allows a better cache coherence as well as a considerable decreasing of cache misses. Cores communication becomes faster since it depends on the shared cache memory speed which is faster than the main memory.

There are two basic types of reference locality. Temporal locality refers to the re-use of specific data, and/or resources, within relatively small time duration. Spatial locality refers to the use of data elements within relatively close storage locations.

To take advantage of spatial locality, we suggest loading in advance data that we shall probably need in the near future in order to enable the processor to find the requested data it usually seeks in the cache memory and to avoid the overhead due to main memory access. This phenomenon is called the Prefetching, and may be performed either by the programmer or directly by the processor which is called the hardware prefetcher. This latter can operate transparently, without any programmer intervention, to fetch data and instructions from memory into the unified second-level cache. The hardware prefetcher can be trusted to prefetch highly regular accesses, while software Prefetching can be used for irregular accesses that the hardware one can’t handle.

In this work, we supposed that our TA doesn’t rely on a hardware prefetcher. It’s in this context that the second phase focuses on the software Prefetching technique (PT) to take advantage of data locality in order to optimize cache memory use.

Let us adopt the following notations:
- \( CL \): Size of cache memory line
- \( DS \): Data size
- \( CQ = CL/DS \): Cache Line-data ratio

The Prefetching technique can be explained through the following MMP code generated by Pluto (see Nest 1.).

```c
lbp=0;
ubp=N-1;
#pragma omp parallel for
for (t2=lbp;t2<=ubp;t2++) {
    for (t3=0;t3<=N-1;t3++) {
        for (t4=0;t4<=N-1;t4++) {
            C[t2][t4]=C[t2][t4]+A[t2][t3]*B[t3][t4];
        }
    }
}
```

- Starting from the innermost for loop (t4), we detect the cache memory size line (assuming that \( CL = 64B \)) by calling a function from the Cpuinfo library.
- Since we know the data size to prefetch (e.g. \( DS(C[i,j]) = \text{sizeof(int)} = 4 \ B \)), we compute the cache line-data ratio corresponding to the data to prefetch (in this case \( CQ = CL/DS = 64/4 = 16 \)). Since all transfers between the main memory and the cache memory are done cache line-wise, when loading \( C[t2,t4] \), we’ll load \( CQ \) data in the same line starting by \( C[t2,t4] \) until \( C[t2,t4+15] \).
- We detect the access type for each data structure, then we call the necessary Prefetching instructions. In our case, \( C[t2,t4] \) would be prefetched when writing and \( B[t3,t4] \) when reading.
- We apply the Prefetching mechanism on the innermost loop by duplicating the instruction \( CQ \) times and by Prefetching at each iteration, the requested data.

This finally leads to the illustrated code in Nest 2. An experimental study targeting the same TA was achieved on the three benchmarks MA, MVP and MMP with the different allocations previously tested after undergoing the Prefetching technique. This could lead to interesting improvements. Tables V, VI and VII show a comparison emphasizing that the best result is provided by the program working with the manual allocation using equal sized sections and undergoing the Prefetching technique (PT).

We detail in Tables V, VI and VII respectively for MA, MPV and MMP, excerpts of experimental comparisons carried out on our TA.
We mention that the following notations are adopted in order to indicate the execution time of the studied codes:

- **ST1_Pre**: code generated by Pluto (CGP) with a chunk size (CHS) equal to Xi (i=1..4, see section 3.2) when the allocation is static, and undergoing the PT.
- **DY1_Pre**: CGP with a CHS equal to Xi (i=1..4) when the allocation is dynamic, and undergoing the PT.
- **Section_Pre**: CGP with manual allocation (fair sections) after undergoing the PT.

### TABLE V. EXECUTION TIME (MS) OF MA FOR DIFFERENT ALLOCATIONS WITH PREFETCHING

<table>
<thead>
<tr>
<th>N</th>
<th>ST1_Pre</th>
<th>ST2_Pre</th>
<th>ST3_PRE</th>
<th>ST4_PRE</th>
<th>DY1_PRE</th>
<th>DY2_PRE</th>
<th>DY3_PRE</th>
<th>DY4_PRE</th>
<th>Section_Pre</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>113.844</td>
<td>40.82</td>
<td>40.923</td>
<td>94.616</td>
<td>104.572</td>
<td>40.116</td>
<td>40.483</td>
<td>99.527</td>
<td>17.697</td>
</tr>
<tr>
<td>3000</td>
<td>248.827</td>
<td>91.688</td>
<td>92.648</td>
<td>224.099</td>
<td>240.715</td>
<td>90.356</td>
<td>89.922</td>
<td>223.656</td>
<td>42.593</td>
</tr>
<tr>
<td>4000</td>
<td>411.637</td>
<td>161.35</td>
<td>161.921</td>
<td>399.57</td>
<td>410.759</td>
<td>160.156</td>
<td>160.054</td>
<td>379.512</td>
<td>76.386</td>
</tr>
<tr>
<td>5000</td>
<td>643.714</td>
<td>257.076</td>
<td>255.802</td>
<td>594.37</td>
<td>634.819</td>
<td>251.689</td>
<td>253.035</td>
<td>592.206</td>
<td>112.569</td>
</tr>
</tbody>
</table>

### TABLE VI. EXECUTION TIME (MS) OF MVP FOR DIFFERENT ALLOCATIONS WITH PREFETCHING

<table>
<thead>
<tr>
<th>N</th>
<th>ST1_Pre</th>
<th>ST2_Pre</th>
<th>ST3_PRE</th>
<th>ST4_PRE</th>
<th>DY1_PRE</th>
<th>DY2_PRE</th>
<th>DY3_PRE</th>
<th>DY4_PRE</th>
<th>Section_Pre</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>30.424</td>
<td>15.067</td>
<td>11.1</td>
<td>27.859</td>
<td>33.475</td>
<td>15.188</td>
<td>9.536</td>
<td>29.702</td>
<td>9.077</td>
</tr>
<tr>
<td>2000</td>
<td>127.351</td>
<td>37.267</td>
<td>33.586</td>
<td>109.82</td>
<td>130.924</td>
<td>34.127</td>
<td>34.106</td>
<td>106.012</td>
<td>14.698</td>
</tr>
<tr>
<td>3000</td>
<td>272.951</td>
<td>77.719</td>
<td>76.621</td>
<td>245.621</td>
<td>293.985</td>
<td>78.194</td>
<td>78.116</td>
<td>244.15</td>
<td>24.048</td>
</tr>
<tr>
<td>4000</td>
<td>486.267</td>
<td>133.826</td>
<td>133.928</td>
<td>412.417</td>
<td>527.677</td>
<td>135.708</td>
<td>135.798</td>
<td>439.297</td>
<td>32.441</td>
</tr>
<tr>
<td>5000</td>
<td>743.987</td>
<td>215.561</td>
<td>213.861</td>
<td>613.375</td>
<td>808.491</td>
<td>218.966</td>
<td>218.684</td>
<td>667.755</td>
<td>33.017</td>
</tr>
</tbody>
</table>

### TABLE VII. EXECUTION TIME (S) OF MMP FOR DIFFERENT ALLOCATIONS WITH PREFETCHING

<table>
<thead>
<tr>
<th>N</th>
<th>ST1_Pre</th>
<th>ST2_Pre</th>
<th>ST3_PRE</th>
<th>ST4_PRE</th>
<th>DY1_PRE</th>
<th>DY2_PRE</th>
<th>DY3_PRE</th>
<th>DY4_PRE</th>
<th>Section_Pre</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td>762.466</td>
<td>327.575</td>
<td>327.865</td>
<td>727.719</td>
<td>588.535</td>
<td>328.164</td>
<td>328.430</td>
<td>569.553</td>
<td>66.423</td>
</tr>
<tr>
<td>4000</td>
<td>1816.399</td>
<td>775.579</td>
<td>775.503</td>
<td>1688.525</td>
<td>1407.161</td>
<td>776.375</td>
<td>776.273</td>
<td>1348.133</td>
<td>184.934</td>
</tr>
<tr>
<td>5000</td>
<td>3541.314</td>
<td>1543.684</td>
<td>1543.122</td>
<td>3326.861</td>
<td>2667.724</td>
<td>1545.763</td>
<td>1545.772</td>
<td>2619.664</td>
<td>363.515</td>
</tr>
</tbody>
</table>

- Execution times: T1 for MA (ms), T2 for MVP (ms), T3 for MMP (s).
- Reducing time ratio r’ (%): 
  
r’(Version)=(T’j(PLUTO)-T’j(Version))/T’j(PLUTO) 
  
where j= for MA, 2 for MVP and 3 for MMP and Version ∈ { STi_Pre, DYi_Pre, Section_Pre } 

For these parallel experiments, we used the “GNU_SOURCE” package. We remind that, for each benchmark algorithm, we chose 10 values of N in the range [500, 5000]. For each N, the execution time is the mean of five runs. So, we achieved 90 tests for each benchmark. Excerpts of the results we obtained are depicted below.

The experimental study of the MA benchmark (respectively MVP and MMP) was carried out with both static and dynamic automatic allocation by varying the CHS as indicated in section 3.2. It was then processed with manual allocation. All these experimental versions have undergone the PT as explained in section 3.3. Table VIII shows our excerpts of experimental comparisons of ST1, ST4, DY1 and Section versions performed on our TA. Several improvements are obtained through the reduction of the execution time.

Similar remarks as done in section 3.2 may be detailed here. The best version is still the manual allocation expressed in “Section_Pre” version for the three studied benchmarks. The reducing time ratios (r’) are also high and vary in the range 93.38 (N=1000) - 96.13 % (N=2000) for MA, 92.53 (N=1000) - 98.93 % (N=5000) for MVP, and 96.58 (N=5000) - 98.16 % (N=1000) for MMP. Furthermore, the variations of r’ are irregular in terms of N. We have to note that with the others versions, we also obtain an improvement relative to the code generated by Pluto (CGP) but less than with Section undergoing the Prefetching technique (PT).
4 Conclusions

Addressing polyhedron program automatic parallelization, we first studied the most known code optimization tools related to Pluto. Then, we proposed a two-phase approach leading to parallel code optimization. Targeting a multicore machine, our proposal takes into consideration the number of cores and the cache line size, and aims to especially optimize the cache memory use. In fact, starting with a program generated by Pluto, we proposed to automatically generate a parallel avoiding-communication code with explicit task allocation on the available cores. This code then automatically underwent the Prefetching technique. It therefore took advantage of the spatial locality which minimized the main memory access. Ultimately, our proposal was translated by a software component that was integrated with Pluto. A series of experimentations could validate our contribution and specify its practical interest. However, several interesting points remain to be seen, particularly: (i) extension of the experimental study to other benchmark matrix algorithms, (ii) optimization of parallel programs with core communications by taking into consideration the cache memory sharing, (iii) adaptation of the code generation to other parallel architectures e.g. GPUs.

Acknowledgment

We’d like to thank Pr. Zaher Mahjoub for his valuable help.

5 References