Predictive Performance Balancing of Parallel Polyhedron Programs on Multicore Platforms

Asma DAB  
Elmanar University, Tunisia  
asmadab1@yahoo.fr

Yosr SLAMA  
Elmanar University, Tunisia  
yosr.slama@gmail.com

ABSTRACT

Nowadays multicores machines are becoming more and more common. Ideally, all the applications benefit from these advances in computer architecture. A complex challenge in parallel computing is cores load balancing to minimize the overall execution time called Makespan of the parallel program. As multicores may have different architecture, an effective mapping should support this unknown variation to avoid drawbacks on Makespan. In fact, mapping or static load balancing method may not be effective when the target state machine changes during program execution. In this context, we propose a predictive approach using iterations chunking at runtime allowing parallel code adaptation to processor’s performance. From a parallel program, we define a set of loop nest iterations, forming what is called chunk, and we run it using a first mapping assuming homogeneous cores. Then, performance assessment would correct mapping by speculating the future core’s state. The new mapping would be then applied to a new chunk for further evaluation and prediction. The process would stop when the program is fully executed or when judging that piecewise execution is no longer effective.

In this paper, a state of art multicore platforms as well as speculative techniques is presented. Then, our proposal dealing with performance prediction process is detailed. Finally, the contribution is validated with several tests and a comparison is held with existing static and dynamic approaches.

KEYWORDS

Automatic parallelization Chunk, Load balancing, Mapping, Multicores, Speculation, Polyhedron program, Prediction.

1 Introduction

Several tools have been proposed in the literature for the parallelization of such programs, some of which are placed in a mathematical formal called polyhedral model (PM). In Targeting homogeneous machines, approaches based on polyhedral model PM have proven their effectiveness. However, when targeting heterogeneous architecture, these approaches become less effective. In contrast, the parallel program generated by the polyhedral model causes a load imbalance between the processors since it is the result of a methodology assuming equality between computing power. Thus, such parallelization influences in particular the parallel computing time Makespan. Another problem is the increasing overhead of communication due to the waiting time for processors synchronization. This problem causes parallel system blocking which decreases the parallel solution effectiveness. Waiting for the slower processor sometimes rises a problem of famine between communicating processors. The heterogeneity of platforms presents a big challenge for developers because with multicore machines the programmer has no idea about the variation of core performance in execution time. In this work, we are particularly interested in
improving the parallelization of a particular class of programs called nested loops. We target especially the polyhedral programs which contain loop nests where each loop bounds are affine functions of enclosing loop indices. After a study of multicore and speculation strategy in Section 2, we decide to inspire speculation in order to adapt to processors performance change. In section 3, we propose a predictive method for parallelizing polyhedron programs on multicore processors. Our approach is validated next by experiments and a comparative study with other automatic methods.

2 STATE OF ART

2.1 Multicore platforms

The advent of multicore processors imposes new strategies for reaching good software performance and exploiting advantageously the provided hardware. The key to success is now radically related to parallelism where applications are performed simultaneously on all the available processors.

There are several possible options to accomplish this: programs can be written by explicitly describing what can be run in parallel and what cannot, the compiler can extract parallel computations from a serial code by performing advanced code analyses and then generate parallel code, or the software can be run on top of runtime system, or virtual machine, performing on-the-fly analyses and parallelization. Nevertheless, although all options have been intensely studied, they all have some inherent limitations. Emerging heterogeneous multicores contain specialized processing elements that do not possess traditional cache hardware. Instead, multiple levels of the memory hierarchy must be explicitly managed by software.

Indeed, multicore machines are now the core of supercomputers, clusters, and even grids. Thus, a parallelizing effective program must take into account the intrinsic characteristics of this new class of machines. In addition to the memory’s organizational aspects (central and cache) that differ one multicore machine from another, the core’s speed (clock) can be so different if the machine is not dedicated to the parallel execution of the already mentioned program (some cores are busier than others creating some heterogeneity). Between 2000 and 2010, CPU and GPU used to be separate, but starting from 2011, they have been integrated on a single chip which increases the heterogeneity within multicore (e.g sandy bridge intel, AMD fusion project Nvidia Denver / Maxwell). A new architecture was proposed for this purpose SIMT (Single Instruction Multiple Data) [1].

2.2 Speculation strategy

With speculative parallelization, code sections that cannot be fully analyzed by the compiler are optimistically executed in parallel. Hardware schemes are fast but expensive and require modifications to the processors and/or memory system. Software schemes require no changes for the hardware of existing shared-memory systems, but can suffer from significant overheads involved with the speculative execution. In fact, the performance of software schemes highly depends on application characteristics, design and implementation of the scheme, and system configuration and size. Simple methods such as Fixed-Size Chunking (FSC) [2] need several ‘dry-runs’ before an acceptable chunk size is found. MESETA [3, 4] assigns chunks with sizes as the execution proceeds, at a certain point, chunk size growth stops and it behaves like FSC. Another technique is Just-In-Time Scheduling for loop-based speculative parallelization JIT [5] which is a runtime scheduling mechanism which avoids performance degradation of FSC strategy. Other traditional scheduling methods were originally designed for loops with no iteration.
dependencies, so they are primarily focused on the problem of load balancing. In general, all these methods perform poorly when used for speculative parallelization, where loops may present unexpected dependencies that adversely affect performance. A solution is to run the targeted program in the frame of a runtime system whose role is to use advantageously the available dynamic information and automatically parallelize on-the-fly some code parts. One main advantage is that the effectiveness of a code transformation can immediately be evaluated at execution time and can be adjusted accordingly by the runtime system in real time. In particular, speculative parallelizing techniques [6, 7, 8, 9, 10, 11] are possible since an online verification can consecutively launch recovery actions, in case where previously speculated information is invalid. Speculative parallelization is an essential strategy to handle the parallelization of general-purpose codes. A well-researched direction in speculative parallelization is thread-level speculation (TLS) [12, 13]. A TLS framework allows optimistic execution of parallel code regions before all dependencies between instructions are known. Hardware or software mechanisms track register and memory accesses to determine if any dependence violation occurs. TLS was implemented using different techniques such as value prediction, state separation, multiple value prediction [11, 12] and CorD technique which is an execution model [13] improved later by dynamic data structures [8]. Other techniques targeting polytope model such as adapting the polytope model for dynamic and speculative parallelization [14,15] and a dynamic speculative approach for polyhedral parallelization using compiler-generated skeletons [16].

3 OVERVIEW OF OUR APPROACH

Our work idea is to propose a predictive parallelization in order to adapt the parallel execution to multicore’s variable performance. Using piecewise execution, code parallelization may improve the Makespan. A motivation example is given in the next section, then we introduce our idea by presenting a speculative execution model and finally we present the load balancing algorithm.

3.1 Motivating example

Given a set of two processors P1 and P2 having respectively available computing power A1 and A2, where A1 = A2 * 2. Suppose that A1 = 1 (iteration per second), to run 1000 iterations, two scenarios are possible: the parallelization with and without speculation. A simple approach is to assign to each processor the half of iterations; in this case, P1 ends computation before P2 but the Makespan will be the parallel execution time of P2. In contrast, if the iterations are divided into pieces of 100 iterations, we will assign iterations as follows: first, we use fair division between P1 and P2; 50 iterations for each. Then, after comparing the parallel times of processors, we take account of processors heterogeneity. So, for the following 100 iterations, we assign 67 iterations to P1 and 33 to P2. And so on until finishing all iterations. We distinguish two possible cases:

1. The frequency changes when executing the first 100 iterations: in this case we define the out-speculation as the modification of frequency at execution time, so to avoid this out-speculation, we pursue the current iterations execution and the following 100 iterations will be executed using new detected performances.

2. The frequency remains unchangeable; in this case we have no out-speculation case, so we continue the execution of the following 100 iterations with the same mapping. The out-speculation number will be used later when speaking about history in the next section.
The following figure shows the two approaches.

![Motivating example 1](image1)

3.2 Execution Model

When speculation is used to parallelize a sequential program, multiple processes or threads should be created. The key idea of this technique is to make the assumption that there is no dependency between the two sequential regions of code (loop iterations) and execute them in parallel. This is based on the observation that in many programs the dependencies that prevent parallelization may not frequently occur at runtime.

In our work, we apply speculation technique by considering a parallel program obtained using the classic polyhedral model. First, the approach defines a small number of iterations, thus forming what is called chunk. After the execution of a chunk with a first mapping by considering that cores are homogeneous, performance evaluation would correct mapping by speculating the future cores state. The new mapping would be applied to a new chunk for further evaluation and speculation (cf. fig 2). The process will stop either when the program is fully implemented, or when performances maintain constant.

Given a parallel computation, all threads execute the first chunk; we assume first that all cores have the same computation power so we divide the first chunk equally among cores. When threads finish assigned tasks, only one thread execute an algorithm of load balancing after computation of all parallel execution time. Hence, it recalculates the number of assigned iterations for each core according to its parallel time. After the execution of the assigned iterations, parallel times will be the input to a second load balancing step called in figure 3 “control” and so on.

![Parallel execution model](image2)

Our approach aim to:

1. Adapt the input parallel program to computing power of multicore machine
2. Speculate polyhedral program at runtime
3. Maximum exploration of target equipment to improve the *Makespan*. 
Sometimes multicore machines are not dedicated to the parallel execution or they are partially loaded. Our goal is to modify a parallel code to adapt it to changes in cores performance at execution time. Therefore, our approach is based on the determination of the appropriate chunk size after testing some random sizes; this can be done after a series of tests. The most important goal we want to achieve is to reduce the execution time $\text{Makespan}$ by using efficiently material resources. It is also very important for programmer to be independent from architectures and characteristics of the target machines. Hence, this approach detects the variation of performance based on core’s parallel time.

Our approach takes as an input the initial parallel polyhedron program. Before starting the parallelization, we use an indicator called hist (number of successive chunks during which there is no change in cores performance) which is initialized to 0, incremented when having no out-speculation and reinitialized to 0 when using new mapping, reflecting a history of performance changes. After execution of the first chunk assuming cores are identical (having the same available computing power), we apply a load balancing method which will be detailed in the next section. After a test of the variable hist, if it is equal to a fixed threshold $k$ we stop speculation and the rest of the program is executed without chunking. Otherwise, we continue the speculation process with a new chunk (cf. figure 4).

![Figure 4. Proposed approach](image)

### 3.3 Load balancing using execution time prediction

Given a parallel program obtained by applying the polytope model, after each chunk execution, one thread executes the sequential part of updating performances. Indeed, the sequential thread will affects iterations based on parallel execution time of the previous chunk. After running a first chunk which is usually of random size, we try in the following speculations to determine the best chunk size to minimize cores idle at execution time. To do this, we tested random sizes in order to determine the ratio between the size of the chunk and the parallel time obtained. The determination of chunk size allows best resource exploitation and more adaptation to core performance changes. After the execution of a chunk, the compiler must check the history of executed chunks, if performance measures remain unchangeable $k$ times in succession, the remaining program is executed without chunking. Otherwise, the new measures will be used for the execution of the next chunk.

We give in the following the load balancing algorithm.
4 EXPERIMENTAL STRATEGY

4.1 Experimental strategy

Our approach has been implemented and experimented using the MMC (Matrices Multiplication Code).

```c
/*Matrices Multiplication*/
forall (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (m=0; m<N; m++)
            C[i][j] = A[i][m] * B[m][j];
```

Figure 5. Matrices Multiplication code

The experimental study consists in:

- Parallel program execution on the target platform by loading cores with arbitrarily tasks. During the execution, the machine is under-use and cores load change randomly so that we have no idea about the load on each core at runtime; this configuration is called C1
- Parallel threads are pinned to specific cores using a function called cpu_bind which uses the library sched_affinity
- Parallel program execution on the target platform using different configurations of processors mask. This is done by using a code called cpuburning allowing masking a percentage of frequency of a given processor. Before starting the execution of the product code of matrices product, the frequencies monitor is fixed on performance mode so processor speeds are setting to max values (2.5 MHz). We used different configurations; for each configuration we change cores load:
  - C2: we mask 99% of one core frequency so we have 3 cores at full speed (100%) and one core with only 1% of its speed (using cpuburning)
  - C3: we mask 50% of two cores frequency (using cpuburning) and let the rest of cores with 100%
- Several parameters have been changed such as:
  - N: Matrix size
  - Chunk_size: used chunk size
  - Core mask: percentage of available performance
  - C: target Configuration
- Implementation of 3 approaches which have been then compared:
  - Our approach
  - Static approach implemented in the OpenMp library
After executing various codes on several configurations, we present next some results of different input matrices size.

In the first section, we study the intra-chunk load balancing of our approach using different matrix sizes and different configurations of target platform. In the second section, we compare our approach with two other approaches using automatic mapping. Our approach is tested on multicore machine Intel core i5 containing 4 cores each of 2.5Mhz.

4.2 Intra-chunk Balancing

For each matrix size N, we consider different chunk sizes chunk_size, we compare the cores parallel times at every speculation as shown in the following diagram (cf. figure 6). We note for (N =1000; chunk_size = 300) on C1 that our approach is adapted to the difference between cores speed. On C1, our approach reduce the difference between cores parallel time from the fifth speculation, we note that for some intervals the Makespan increases this can be explained by the asynchronous cores load when executing the chunk itself.

For the configuration C2 (N=2000; chunk_size=300), the processor P4 is initially the slowest one (spec = 1, cf. Figure 7). After three speculations our approach reduces the difference between the cores parallel time.

We present the case in figure 8 (N=5000, chunk_size=300) on C3, where the speed of two processors is loaded to 50%. Strating from spec=9, the Makespan remains constant until the execution end and, in all chunks, cores load become well balanced.
In Figure 8, Intra-chunk speculation’s load balancing on C3: N=5000; chunk_size=300

When the chunk is N / 2 and in the case where the size of the chunk is very close to N, the number of speculation decreases which reduces the chance of convergence to well balancing loads but the intra-chunk load balance of the second speculation still better than that of the first speculation (cf. table 1).

Table 1. Intra_chunk load Balancing for both cases chunk_size = N / 2 and chunk_size very close to N

<table>
<thead>
<tr>
<th>Chunk Size</th>
<th>First Speculation</th>
<th>Second Speculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3: N=5000, chunk_size=300</td>
<td>150,42, 6</td>
<td>37,556</td>
</tr>
<tr>
<td>C3: N=5000, chunk_size=2500</td>
<td>150,968</td>
<td>37,704</td>
</tr>
<tr>
<td>C2: N=5000, chunk_size=2500</td>
<td>212,74</td>
<td>42,502</td>
</tr>
<tr>
<td>C2: N=5000, chunk_size=2500</td>
<td>214,205</td>
<td>42,798</td>
</tr>
</tbody>
</table>

To sum up:

- Our approach decreases the including differences between cores parallel time
- Even for small matrix sizes, our approach adapts to speed variation
- There is always a convergence towards balancing
- Starting with a small chunk is better to converge faster to load balancing
- When the balancing is reached during successive chunks, it will be better to use large chunk for the remaining speculation steps

4.3 Chunk size Impact on Makespan

After testing different configurations, three main parts are obvious at figure 9:

1. For small chunk sizes, theMakespanreaches the maximum due to the number of synchronizations between processors and the computation’s overhead
2. From a certain size (generally 200), theMakespandecreases until reaching a minimum which differ from chunk size to another
3. In a third phase theMakespanincreases until reaching a maximum for chunk_size=N

Between these three phases, sometimes, there are variations. But basically we notice the presence of the three phases in all tests. More size is decreased the greater the number of chunk synchronization increases. Although the small size of the chunk allows good adaptation to the change in performance, it appears from Figure 9 that the small size of the chunk also induces an increase in the parallel time. Thus, to choose the best chunk, we must find a compromise between the size of the chunk and theMakespan.
For large chunk sizes, it should be noted that the *Makespan* increases until reaching its maximum for a chunk size equal to N. The number of speculations decreases when increasing chunk size, so the chance for adaptation decreases when chunk_size > N / 2. From a certain chunk size (e.g. chunk_size = 1000 in Figure 9), *Makespan* increases in proportion to the chunk size.

We note, in comparing curves of figure 9 that more we decrease chunk size more our approach adapts to speed changes.

![Figure 9. Chunk size effect on Makespan for N=5000](image)

**4.4 Effectiveness of speculation vs. conventional methods**

In all configurations, our approach outperforms dynamic and static approaches (cf. Fig 10). By increasing the size of the chunk, our approach converges later to load balancing.

Even on C3 configuration (cf. Fig 12), parallelization by speculation is much better than the dynamic method and the static approach. This can be explained by the OpenMP routines used for parallelization. Automatic mapping has many drawbacks such as increasing the overhead, also, when target machines are not dedicated to parallel execution, automatic mapping cannot adapt with speed variation. Indeed, manual mapping design distributes the load in accordance with core powers computation. Instead, methods which are included in the OpenMP library use automatic load distribution between cores without having any idea about the computational power of processors. Also on C2 (cf. Fig 11), our approach outperforms the two automatic approaches. Having two processors with the half frequency of the two others does not influence our approach.

![Figure 10 Comparaison of three approches on C1 chunk_size=300](image)
Figure 11. Comparison of three approaches on C2 chunk_size=300.

Figure 12. Comparison of three approaches on C3 chunk_size=500.

Our approach outperforms the GSS (guided scheduling algorithm that allows chunks to be progressively reduced in size) on all configurations such as on C1 configuration with a chunk_size=500 (cf. figure 13).

We also tested if the boundary or the chunk size is equal to the size of the input matrices (cf. table 2.).

Table 2. Comparison of three approaches in case chunk_size = N

<table>
<thead>
<tr>
<th>Processor</th>
<th>Parallel process Time</th>
<th>T(sec)</th>
<th>T Overhead</th>
<th>T(sec) Static Approach</th>
<th>T(sec) Dynamic Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1041.49</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>1044.13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>1472.9</td>
<td>1474.87</td>
<td>0.00033</td>
<td>2182.1</td>
<td>1828.19</td>
</tr>
<tr>
<td>P4</td>
<td>1474.85</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It is clear from the above table that the manual mapping used by our approach reduces the Makespan compared with that of two other approaches.

Conclusion

Speculation was a good way to parallelize code by piece and re-execute chunks in case of error detection. By studying the state of the art of parallelization methods targeting multicore
machines, we have noticed that most of the approaches which are based on speculation use this principle to speculate data computation or memory spaces or scheduling of polyhedral programs. Multicore processors machines do not always have the same computing power; the scheduler used by the operating system may cause load imbalance between processors. To adapt to the change in cores speed, we have proposed an approach based on the principle of speculation. Our approach executes a parallel code chunk by chunk. After each chunk, our approach redistributes loads among processors based on their parallel times. Tested on various configurations of multicore machine, our approach showed good adaptation to changes in the processors frequencies. This adaptation is observed via a good intra-chunk load balancing for all cases of input matrices sizes and in particular when using small chunk sizes. We also compare our approach which uses a manual scheduling with other built-in OpenMP automatic approaches namely static and dynamic methods. In all tests, and even if the chunk size is equal to the size of the input matrix, our approach outperform the three approaches.

REFERENCES