Dynamic performance prediction for chunk-wise parallelization on heterogeneous CPU/GPU systems

A. Dab[1], **Y. Slama**[1]
[1]Computer Science Department, Tunis El Manar University, Tunis, Tunisia

Abstract - *Many aspects of heterogeneity in multicores such as performance variation may affect the overall execution time and cores efficiency. An effective mapping should support this variation. A complex challenge is cores load balancing to minimize the program makespan. In this context, we propose a predictive approach using iterations chunking at runtime allowing parallel code adaptation to heterogeneous systems containing GPU and heterogeneous CPUs. Our approach is based on thread pinning and performance detection at runtime. From a parallel program, we run a chunk on the GPU device. Another chunk is running on CPU using a first mapping assuming homogeneous cores. Then, performance assessment would correct mapping by speculating the future GPU and core's state. The new mapping would be then applied to a new chunk for further evaluation and prediction. This process would stop when the program is fully executed or when judging that chunking is no longer effective.*

Keywords: Chunk; GPU; Load balancing; Multicores; Thread affinity; Prediction

1 Introduction

Nowadays, multicore architectures and general-purpose GPUs are the state of the art in the industry of processor design for desktop and high performance computing. With this design, multiple threads can run simultaneously exploiting thread level parallelism and consequently, improve overall program performance of the system. Another way to improve the program performance is the wise use of GPUs in addition to the CPU cores. When targeting multicore machines, several tools have been proposed in the literature for the parallelization of different kinds of programs such as polyhedral programs PP. Based on a mathematical model; many tools called polyhedral tools have been developed in order to parallelize PP. In Targeting homogeneous machines, approaches based on polyhedral tools have proven their effectiveness. However, when targeting heterogeneous architecture such as CPU/GPU systems or heterogeneous CPUs, 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 powers. Thus, such parallelization influences in particular the parallel computing time a.k.a *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 the core's available computing power at runtime. Therefore, thread affinity has appeared as an important technique to exploit data sharing, to accelerate program execution times and to fix thread affinity for better performance stability. 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 PP which contain loop nests where each loop bounds are affine functions of enclosing loop indices. After a study of speculation and thread pinning strategies in Section 2, we decide to inspire speculation in order to adapt to processors performance variation and to take advantage of the GPU device leading to the wise use of the CPU/GPU target system. In section 3, we propose a predictive method for parallelizing polyhedral programs on CPU/GPU systems. In section 4, our approach is validated by experiments on different configurations of heterogeneous CPU/GPU systems.

2 Related work

The trend in general-purpose chips consists of multiple cores of various types— including traditional, general-purpose compute cores (CPU cores), graphics cores (GPU cores), digital signal processing cores (DSPs), cryptographic engine, etc.—connected to each other and to a memory system. Already, general-purpose chips from major manufacturers include CPU and GPU cores, including Intel's Sandy Bridge, AMD's Fusion, and Nvidia Research's

Echelon. IBM's PowerEN chip includes CPU cores and four special-purpose cores, including accelerators for cryptographic processing and XML processing. Other aspect of heterogeneity is the cores performance or the available computing power which may affect the cores efficiency. For these reasons, thread affinity has appeared as an important technique to improve the overall program performance and for better performance stability.

The performance of software schemes depends not only on the target architecture, but it depends also highly on the application characteristics, design and implementation of the scheme. A common technique is speculation where code sections that cannot be fully analyzed by the compiler are optimistically executed in parallel. Software schemes require no changes for the hardware of existing shared-memory systems, but can suffer from significant overheads induced by the speculative execution. Simple methods such as Fixed-Size Chunking (FSC) [1] need several 'dry-runs' before an acceptable chunk size is found. MESETA [2, 3] assigns chunks with increasing 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 [4] which is a runtime scheduling mechanism that avoids performance degradation of FSC strategy. 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 runtime and can be adjusted accordingly by the runtime system in real time. In particular, speculative parallelizing techniques [5, 6, 7, 8, 9, 10] are possible since an online verification can consecutively launch recovery actions, in the 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) [11, 12]. A TLS framework allows optimistic execution of parallel code regions before all dependencies between instructions are known. Hardware or software mechanisms track registers 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 [13, 11] and CorD technique which is an execution model [12] improved later by dynamic data structures [7]. Other techniques targeting polytope model are proposed. We cite among them the polytope model adaptation for dynamic speculative parallelization [14,15] and polyhedral parallelization using compiler-generated skeletons [16]. In [17], Aravind proposes a framework Apollo (Automatic speculative POLyhedral Loop Optimizer) that goes one step beyond the current automated compilers, and applies the polyhedral model dynamically by using TLS system. The idea consists of using the polyhedral model at runtime, aided by thread level speculation.

Our current paper inspires by speculation through chunk-by-chunk execution to predict the appropriate chunk sizes for CPUs and GPU. On CPU, our approach applies also chunk-by-chunk execution using thread-to-core pinning to adapt parallel program to performance variation at runtime on multicore machines.

3 Approach view

3.1 Motivation

The high performance of GPU accelerators encourages programmers to use it in parallel scientific computations. Hence, another aspect of GPUs known as general purpose GPU GPGPU appeared. Many projects are oriented toward heterogeneous processors like the AMD Fusion project, or the Larra bee project Intel. The general idea is to integrate specialized cores to certain treatments, rather than adding additional functionality on general core. The ultimate goal is to decrease the overall consumption of the processor in equal functionality. Always in a power saving purpose, manufacturers also imagine heterogeneous cores at the operating frequency within the same processor. In order to take advantage from existing and future multicore processor architectures, it is essential to develop parallel applications and to adapt existing applications accordingly. To take advantage of multicore architecture, most of thread affinity studies on multicore architectures focus on data locality and cache sharing in parallel applications. A hierarchy of memory caches allows exploiting data sharing between threads running on such platforms. Of course, to exploit that, a multi-threaded application has to meet two conditions. First, threads have to access common data. Second, the reuse distance has to be short enough to effectively exploit these shared data across multiple threads. Mazouz [18] proposes an approach that allows changing thread affinity dynamically (thread migrations) between parallel regions at runtime to account for these distinct inter-thread data sharing patterns. An automatic tool for thread pinning is autopin – Automated Optimization of Thread-to-Core Pinning on Multicore Systems: this framework is designed to automatically detect the best binding between threads of a running parallel application and processor cores in an SMP system, using hardware performance counters [13]. In our study case, we find it very important to adapt parallel codes to new architectures using our approach based on performance prediction and thread pinning.

3.2 Parallel Execution Model

3.2.1 General CPU/GPU approach

In our work, we apply a technique called performance prediction by considering a parallel program generated from the classic polyhedral model. Our approach takes as input the parallel program obtained by applying the Polytope model and generated by an automatic parallelizer (such as Pluto). Our goals are: 1) Adapt the input parallel program to computing power of the heterogeneous system (heterogeneous multicore machine and accelerator), 2) Speculate polyhedral program at runtime, 3) improve the *makespan* through maximum resources exploration. This approach is an extension of our previous work that was limited to heterogeneous CPU case[1]. Hence, two chunks are executed by CPU and GPU. On CPU, the parallel execution of the chunk assumes that it is equally divided among cores. Then, we compute the parallel execution times of cores and the execution time of GPU. To predict the next chunk sizes, we compare the average time of CPU cores to the execution time of the accelerator. After that, CPU and GPU will execute the assigned iterations. In the case of CPU, the CPU approach is adopted (for more details see the section 3.3).

3.2.2 CPU approach

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 variations in cores performance at runtime. It is also very important for programmer to be independent from the architectures and the characteristics of the target machines. After the execution of a chunk with a first mapping resulting from previous chunk execution on CPU, performance evaluation would correct mapping predicting the future cores state. Thus, the new mapping would be applied to a new chunk for further evaluation and prediction. When threads finish assigned tasks, only one thread computes all parallel execution times and then executes an algorithm of load balancing. Hence, it recalculates the number of assigned iterations to each core according to its parallel time. After the execution of the assigned iterations, parallel times will be the input of a second load balancing step. Then, we apply a load balancing method to compute the load balancing metric denoted ε(p) (which will be detailed in the section 4.4.). After a test of this metric; if it is equal to a fixed threshold k, we stop prediction and the rest of the program is executed without chunking. Otherwise, we continue the prediction process with a new chunk (cf. figure 3).

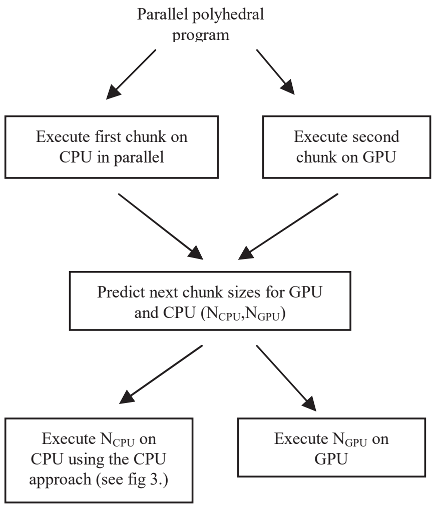

Parallel polyhedral program
Execute first chunk on CPU in parallel
Execute second chunk on GPU
Predict next chunk sizes for GPU and CPU (N_{CPU},N_{GPU})
Execute N_{CPU} on CPU using the CPU approach (see fig 3.)
Execute N_{GPU} on GPU

Figure 1. General CPU/GPU approach

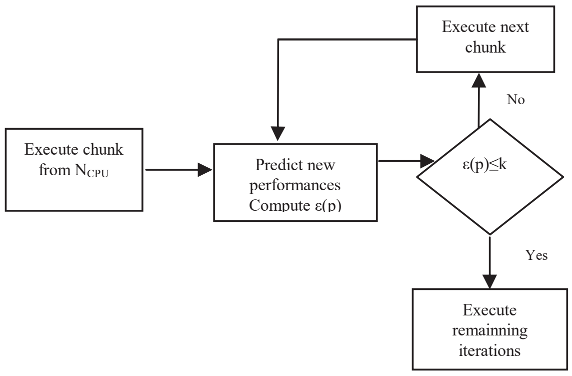

Execute next chunk
No
Execute chunk from N_{CPU}
Predict new performances Compute ε(p)
ε(p)≤k
Yes
Execute remainning iterations

Figure 2. General CPU approach [1]

3.3 CPU/GPU Load balancing algorithm

After two chunks running on CPU and GPU, one thread Th1 updates the CPU performances and computes the average execution time of cores parallel execution times. Hence, it computes the ratio beta as follows: beta= min(T_{GPU}, T_{moy})/max(T_{GPU},T_{moy}) (where T_{moy} is the average of the CPU cores parallel execution times and T_{GPU} is the GPU execution time). Basing on beta values, this sequential thread determines N_{CPU} (the next CPU chunk size) and N_{GPU} (the next GPU chunk size). The N_{CPU} is then executed chunk by chunk on CPU cores; the first chunk is divided among processors based on their previous parallel execution times. After each chunk execution we check if processors are well balanced (i.e there is perfect load balancing intra-chunk which will be detailed in section 4.4.) or not based on their parallel execution times. In the case they are not well balanced, a new chunk is divided among processors using new mapping. After this chunk

[1] This approach was detailed in a paper accepted in the HPCS conference (to appear in the 2017 proceeding)

execution, we check again the load balancing, update mapping, execute next chunk and repeat the same process until the whole program is executed. In the other case, where processors are well balanced, the chunking process is stopped and the remaining iterations are divided among processors using previous mapping. We give in the following the CPU/GPU load balancing algorithm whose inputs are: the problem size is: the program size N, the chunk_size and the number of processors p.

CPU/GPU Load Balancing Algorithm

```
Inputs : N, chunk_size,p
 Initializations :
        coef1=coef2=coef3= …=coefp=1/p

    Execute a chunk on GPU and compute $T_{GPU}$

    Compute the assigned iterations to each processor pj
    Execute a chunk on CPU: execute Nj and compute parallel times Tj
    Compute Tmoy (Tmoy= $\Sigma_{j=1..p} T_j/p$)
    Compute beta value: beta=min($T_{GPU}$, $T_{moy}$ )/max($T_{GPU}$,$T_{moy}$)
      if($T_{moy}$>$T_{GPU}$) then
        $N_{GPU}$= (1- beta) *$N^*$, where $N^*$=N-2*chunk_size // the number of
remaining iterations after two chunks execution
      Else
  $N_{GPU}$= beta*$N^*$
End if
   $N_{CPU}$= $N^*$- $N_{GPU}$
   Execute $N_{GPU}$ on GPU
   Compute coefj for each processor pj (coefj =Tj/ $\Sigma_k T_k$ )
   Execute $N_{CPU}$ using the CPU load balancing algorithm
```

CPU load balancing Algorithm

```
Inputs : $N_{CPU}$, chunk_size,p,{coefi,i=1..p}
 Initializations :
      chunk_number= ┌ $N_{CPU}$/chunk_size ┐


Compute the assigned iterations to each processor pj
(Nj=chunk_size*coefj)

Repeat{
   Execute Nj on each processor pj
   Compute parallel times Tj
   Well_balanced=check_cores_balancing
    Nj <-  Predict_next_chunk
       }

Until(j<chunk_number && well_balanced=false)

  Execute the remaining iterations in one chunk
```

Predict_next_chunk algorithm

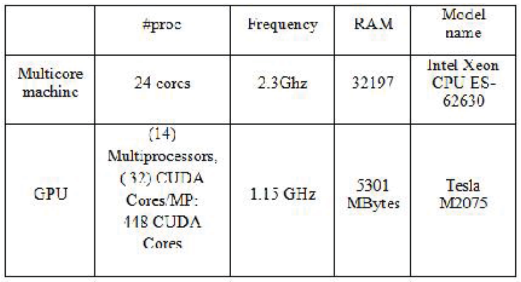
```
  Inputs :{Tj,j=1..p}, p,$N_{CPU}$,chunk_size,well_balanced, i
 Outputs: {coefj,j=1..p}, {Nj,j=1..p}

   Compute coefj for each processor pj   (coefj= Tj/ Σk Tk)
  if(well_balanced=false) then
           Compute Nj (Nj =chunk_size*coefj)


Else
           Compute Nj (Nj=($N_{CPU}$-i*chunk_size)*coefj)
  End if
.
```

4 Experimental study

4.1 Platform description

This section presents a performance evaluation and analysis about the effectiveness of our approach. We create heterogeneity between cores cpu_burn strategy. This strategy consists in creating heterogeneity between cores by masking a percentage of core's frequency. This is made by affecting to the given core random tasks which have realtime priority prt-rt=99 (high priority), so these tasks may not be migrated to other cores. Our experimentations are made on GRID'5000 [18] at the "Lyon site" using a node and an accelerator GPU from "orion cluster" forming hence a CPU/GPU system. The characteristics of the target multicore machine and the accelerator used in our experimentations are presented in the table 1.

Table 1. Characteristics of the target platform



	#proc	Frequency	RAM	Model name
Multicore machine	24 cores	2.3Ghz	32197	Intel Xeon CPU ES-62630
GPU	(14) Multiprocessors, (32) CUDA Cores/MP: 448 CUDA Cores	1.15 GHz	5301 MBytes	Tesla M2075

To evaluate our approach, tests are done using different multicore configurations. Let p be the number of used cores (4 or 8 cores), they are denoted Pi (i=0..(p-1)). Configurations are created using cpu_burn strategy. They are denoted (x0, x1, x2, x(p-1)), where xi is the available frequency percentage of Pi. Table 2 resumes the used configurations based on Cpu_burn on each CPU/GPU system (***d-h*** stands for heterogeneity degree which is the power of the fastest processor divided by the power of the slowest one),

Table 2. Used configurations based on cpu_burn

	CPU/GPU system	***Cpu_burn***	***d-h***
C1	8cores+1GPU	(50,100,100,100,100,100,100,100)	2
C2	8cores+1GPU	(34,100,100,100,100,100,100,100)	2,94
C3	8cores+1GPU	(34,19,100,100,100,100,100,100)	5,26
C4	4cores+1GPU	(34,19,100,100)	5,26
C5	8cores+1GPU	(27,19,40,100,100,100,100,100)	5,26
C6	4cores+1GPU	(27,19,40,100)	5,26
C7	4cores+1GPU	(50,100,100,100)	2
C8	4cores+1GPU	(34,100,100,100)	2,94

A major problem when affecting threads to cores is thread migration. This mechanism is integrated in the operating system to create load balancing. To handle this problem, we use a proper strategy which consists of pinning threads to cores at real-time (dynamic pinning at runtime). Threads are pinned to cores according to their execution time in the previous chunk. To validate our approach, we applied it to a matrix product using arrays of size= N*N for best memory access. The outer parallel loop is transformed hence onto a loop nest. The new outer loop is executed in parallel with a lower bound 1 and an upper bound p where p is the number of processors. The added inner loop is sequential; it represents the processor load. The chunk size will be a set of iterations from the outer loop. To pin threads to cores, we used Posix Pthread API. To parallelize our code, we use the cuda library (nvcc compiler to compile the parallel code) and the openmp library (compilation is done using the flag –Xcompiler /–fopenmp). In the next sections, we present obtained results for different matrix sizes, different chunk sizes and different configurations. We conclude by evaluating our approach performance using metrics detailed in next sections.

4.2 CPU/GPU load balancing

To compute the CPU/GPU load balancing, we used a metric called beta wich is: beta= min(TGPU, Tmoy)/max(TGPU,Tmoy) (where Tmoy is the average of the CPU cores parallel execution times and TGPU is the GPU execution time). The values of beta on different configurations are depicted in the figure 3. As it is shown in the figure 3, more the configuration is heterogeneous more the beta value decrease (the chunk assigned to GPU is higher than that assigned to CPU). This is, in fact, because the performance decreases which affects Tmoy. When comparing beta values basing on chunk size, it is clear in all configurations that the lower the chunk size is, the better the beta value is. We also note that the beta value increases if the number of processor increases (by comparing the results on C3 and C4 or those on C5 and C6).

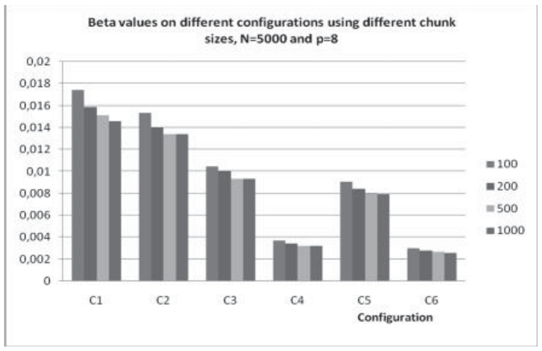


Figure 3. Beta values on different configurations using different chunk sizes, N=5000

The table 3 presents some values of beta on C2 for different matrix and chunk sizes. As it is shown, the beta value decreases when the matrix or the chunk size increases. In fact, this can be explained by the increase of Tmoy. This is noticed for all configurations not only on C2.

Table 3.Beta values on C2 for different matrix and chunk sizes

N	Chunk size	Beta value
1000	100	0,038
	200	0,031
	500	0,028
3000	100	0,020
	200	0,018
	500	0,017
	1000	0,017
5000	100	0,015
	200	0,014
	200	0,014
	500	0,013
	1000	0,013
10000	100	0,014
	200	0,013
	500	0,012
	1000	0,012

4.3 Chunk size impact on execution time

On used configurations, several matrix sizes are tested. For each size, we have varied chunk size and computed the total execution time *makespan* as well as the load balancing overhead. On all configurations, many chunk sizes are tested then we chose these values: 100, 200, 500 and 1000 to be presented in this paper. By comparing the program makespan for various chunk sizes with different matrix sizes, we note that more the chunk size increases more the *makespan* rises (cf. figure 4). We note also that the overhead is negligible comparing with the *makespan* in all cases.

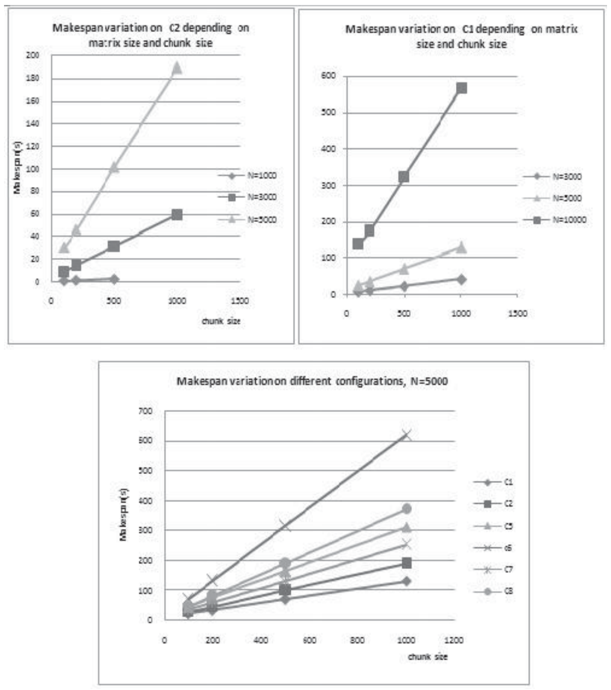


Figure 4. Makespan variation on different configurations depending on chunk and matrix sizes

In figure 4, we fixed matrix size to N=5000 and compute the execution time on the six configurations using different chunk sizes. In the configuration C3, which has the highest degree of heterogeneity, we notice that using large chunks will increase the overhead due to the time spent on waiting the slowest processor when executing the first chunk. Chunks should not be too small neither too large. From figures 4, we notice that it is recommended to use a small chunk size to achieve the best *makspan*. Also, by comparing the *makespan* on C7 (or C4) and C5 for all chunk sizes, we find that applying our approach on 4 cores and 1 GPU gave better results than using 8 cores (where 3 cores are used in low frequencies) and 1 GPU. In our experiments, we find that 100 is the best chunk size to be used. This is also found in the case N=10000 (cf. figure 4) and all other cases. To validate our approach, we compute the speed-up of the proposed parallel program with reference to the original parallel program where iterations are fairly distributed among CPU and GPU (N/2 iterations for each); N/(2*p) for each CPU core. The speed-up is computed as follows: Speed-up = O*riginal version execution time / Proposed version execution time ;* where the proposed version execution time is the sum of all the chunks execution times . We notice from figure 5 that the speed-up achieves high values on the configuration representing the highest heterogeneity degree (such as C8). In the next section, we present another optimization for our approach according to the intra-chunk load balancing.

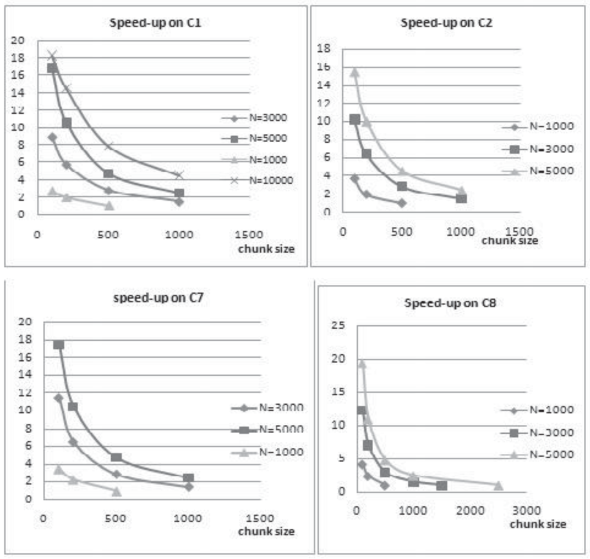


Figure 5. Speed-up variation on different configurations depending on chunk and matrix sizes

4.4 Intra-chunk load balancing

To measure the intra-chunk balancing, let us remember one again that we introduced a new metric called ε(p) which determines the load balancing level between the p processors after a chunk execution. This metric is computed as follows: $\varepsilon(p)= \Sigma_{i=1..p}$ (T_max-Ti) / (p*T_max) where T_max is the maximum of cores parallel execution time. The perfect load balance would be in the case $T_i = T_j =$ T_max. Hence, the lower the epsilon is, the better the load balancing is. In the figure 6, after a first chunk execution on CPU, the amount of assigned iterations to CPU (N_{CPU}=155) is executed chunk by chunk on CPU. Using chunk size 100 in the case N=10000, our approach adapts to cores heterogeneity very well. This is clear, in the figure 6, at the second and the third chunk. The epsilon value decreased from one chunk to another (cf. table 4 and figure 6).

Table 4. Epsilon values on C2, N=10000, chunk_size=100

Chunk	Epsilon
1	25,11
2	2,55
3	1,62

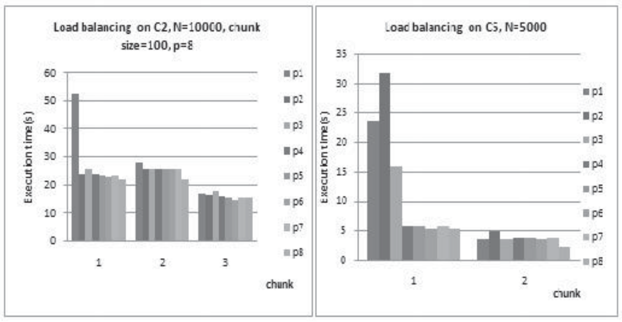


Figure 6. Intra-chunk load balancing on C2 for N=10000 and N=5000, chunk size=100

In all tested cases, our approach adapts the parallel code to the target platform after the execution of the first chunk. The load balancing depends on the number of iterations assigned to the CPU (N_{CPU}). In fact, in most cases, N_{CPU} is very small. That's why the CPU approach is executed using one or two chunks (cf. figures 6).

Conclusion

In this paper, we propose an approach which is an extension of our previous CPU approach to the CPU/GPU systems. The aim of this approach is to adapt parallel programs to heterogeneous multicores that are connected to a GPU device. Using thread affinity, threads are explicitly mapped to cores. Tested on various configurations of multicore machines and GPU, our approach showed good adaptation to variations in the processors performances. 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. Our approach, compared to the original version, which means a good resistance to heterogeneity. High speed-up has been also detected especially when using small chunk sizes. Even in the configurations having high degree of heterogeneity our approach outperforms the original version. As future work, we intend to extend our experimental studies in order to target more heterogeneous platforms. This approach will be proved later by comparisons with other existing dynamic approaches. We intend also to integrate our approach in an automatic parallelizer.

Acknowledgement: Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

5 References

[1] C.P. Kruskal and A. Weiss. 1990. Allocating independent sub-tasks on parallel processors. IEEE Transactions on Software Engineering. (11/10):1001–1016.

[2] D. R. Llanos, D. Orden and B. Palop. Meseta: A new scheduling strategy for speculative parallelization of randomized incremental algorithms. In Proc. 2005 ICPP Work-shops (HPSEC-05). 121–128. IEEE Press.

[3] D. R. Llanos, D.Orden and B. Palop. New scheduling strategies for randomized incremental algorithms in the con-text of speculative parallelization. IEEE Transactions on Computers. 839–852. 2007.

[4] Llanos, D. R., Orden, D., and Palop, B. Just-In-Time scheduling for loop-based speculative parallelization Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing PDP '08. 334-342. 2008.

[5] C. Tian, M. Feng, V. Nagarajan and R. Gupta. Speculative Parallelization of Sequential Loops on Multicores. International Journal of Parallel Programming, Springer US. 508-535. 2009.

[6] L. Gao, L. Li, J. Xue and T. Ngai. Exploiting Speculative TLP in Recursive Programs by Dynamic Thread Prediction. Compiler Construction. 78—93. 2009.

[7] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Speculative Parallelization of Sequential Loops on Multicores. International Journal of Parallel Programming, 37(5). 508-535. 2009.

[8] M. Cintra, and D.R. Llanos. Design space exploration of a software speculative parallelization scheme. Parallel and Distributed Systems, IEEE Transactions on. (16/6). 562-576. 2005.

[9] K. Hanjun, P. J. Nick, W. L. Jae, A. M. Scott and I. David. Automatic speculative doall for clusters. In Proceedings of the Tenth International Symposium on Code Generation and Optimization, CGO '12. 94–103. 2012.

[10] T. Chen, F. Min, and R. Gupta. Speculative Parallelization Using State Separation and Multiple Value Prediction. Proceedings of the International Symposium on Memory Management. 63-72. 2010.

[11] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy Or Discard Execution Model For Speculative Parallelization On Multicores. MICRO. IEEE Computer Society. 330-341. 2008.

[12] T. Chen. Speculative Parallelization on Multicore Processors. Doctoral thesis. University of California. 2010.

[13] T. Klug, M. Ott, J. Weidendorfer and C. Trinitis. Autopin–automated optimization of thread-to-core pinning on multicore systems. In Transactions on high-performance embedded architectures and compilers III (pp. 219-235). (2011). Springer Berlin Heidelberg.

[14] P. Clauss and A. Jimborean. Does dynamic and speculative parallelization enable advanced parallelizing and optimizing code transformations. DCE - 1st International Workshop on Dynamic compilation from SoC to Web Browser via HPC, in conjonction with HiPEAC. 2012.

[15] A. Jimborean. Adapting the polytope Model for Dynamic and speculative parallelization. Doctoral Thesis. Strasbourg University. 2012.

[16] A. Jimborean, P. Clauss, J. F. Dollinger, V. Loechner and J. M. Martinez Caamaño. Dynamic and Speculative Polyhedral Parallelization Using Compiler-Generated Skeletons. International Journal of Parallel Programming, Springer US. 1-17. 2013.

[17] S. Aravind. Beyond the Realm of the Polyhedral Model: Combining Speculative Program Parallelization with Polyhedral Compilation. Doctoral Thesis. Strasbourg University. 2015.

[18] Grid5000 site: https://www.grid5000.fr