Using the Cell

for some operations in the DBM library of UPPAAL

SSE4, Spring 2008.

Olivier Monsonego

Aalborg University
Department of Computer Science

Aalborg University
Department of Computer Science

TITLE:
Using the Cell
for some operations in the
DBM library of UPPAAL

THEME:
Distributed Systems
and Semantics

PROJECT PERIOD:
01/02/2008-03/06/2008

PROJECT GROUP:
d606b

GROUP MEMBERS:
Olivier Monsonego

SUPERVISOR:
Alexandre David

SYNOPSIS:

In this project, we implemented a solution to optimize the resolution of a short path problem in view of accelerating the use of the model checker UPPAAL. Also, we presents the result of the experiments we made, and few solution to improve our implementation. In the best condition (big matrix), our program is able to make the calculation 3,85 times faster than the regular implementation.

NUMBER OF COPIES: 1

NUMBER OF PAGES: 34

CONCLUDED: 03/06-2008

This report is a result of the work done in the SSE4 semester of the Cand.Scient study at Department of Computer Science, Aalborg University. This project was done in the research area of Distributed Systems and Semantics and constitutes the complete work towards a master thesis.

I would like to thank our supervisor Alexandre David for supervising this project and the many helpful comments made throughout the project.

This thesis has been revised on 3st of June, 2008.

Olivier Monsonego

Summary

The model checker UPPAAL, an integrated tool environment for modeling, simulating and verifying real-time systems, needs to make calculations on matrix, and some are used really often like the shortest path calculation which is one of the most used. This report presents different all pairs algorithms and several solutions to parallelize these on a cell broadband engine. We also provide some test scenarios and experimental results to show the capacities of the CBE (Cell Broadband Engine) and how much the parallelization is improving the speed.

Contents

Chapter 1

Introduction

Nowadays, the CPU technology is much more advanced than few years ago. But the evolutions are comming slower than before, and we almost reached the maximum in term of miniaturization and frequency.
The Cell Broadband Engine (CBE) is making a small revolution, and is considered by many people as a "monster" in parallelized calculation thanks to its 8 SPEs and a bunch of new improvements.
Unfortunately, most of the programs are not optimized for this architecture...

1.1 UPPAAL Overview

The model checker UPPAAL [1] is an integrated tool environment for modeling, simulating and verifying real-time systems. UPPAAL was first released in 1995 and is developed by the Department of Information Technology at Uppsala University in Sweden and by the Department of Computer Science at Aalborg University in Denmark. The name UPPAAL comes from the first letters of the universities cities.

1.2 Problem Statement

The use of UPPAAL to check or simulate complex models might take time. Some calculations are done often like the shortest path calculation. If some redundant operations can be speeded up, it is the whole UPPAAL which will be speeded up.

1.3 Goal

The main aim of this project is to accelerate the calculation of the shortest path algorithm used in UPPAAL by parallelizing it on the CBE architecture. But to achieve this goal, we also need to study the CBE by testing several

points such the bus communication speed, time to create SPE contexts, ... All those tests can be found in the Chapter 5.

1.4 The Structure of the Report

The rest of the report is organized as follows:

Chapter 2 Describes the architecture used during this project, and several existing algorithms to solve our problem.

Chapter 3 Describes the chosen way to parallelize the algorithms described in Chapter 2 and how they are implemented on the CBE architecture with technical details.

Chapter 4 Describes some testing scenarios and experiments that were carried out.

Chapter 5 Describes some possible algorithm optimizations, or improvements using new technologies.

Chapter 6 Concludes this report where are also specified some possible future work related to this project.

Chapter 2

Analysis

The aim of this chapter is to provide an overview of the architecture we are using, and to present the most known algorithms in accordance with our goal: the optimized calculation of the shortest path.

2.1 Playstation 3 and CELL Processor Architecture Details

The device we are using to perform the tests is a Sony Playstation 3. The raison of this choice is that nowadays, the Playstation 3 is the most accessible platform with a CBE.

2.1.1 Playstation 3 Architecture Details

The Playstation 3 is built with an architecture similar to a PC and has these main characteristics [2]:

- 256MB of XDR DRAM at 3,2GHz dedicated to the CPU,
- 256MB of GDDR3 VRAM at 700MHz dedicated to the GPU,
- 20GB of Hard-drive,
- Cell Broadband Engine at 3,2GHz with 512KB of L2 cache.

About this architecture, three points are interesting for us: The memory, the CPU and the operating system.
The amount of memory on the Playstation 3 can be considered as a problem: only 256Mb. On the Playstation we used for the tests, once the operating system is launched, only 210Mb are visible, and only few Mb are free... The performances might be affected by this lack of memory. About the swap, almost all of the 512Mb is free.

About the Operating system, the one installed is a Yellow Dog Linux release 5.0 (Phoenix) which is an OS specialized in the Power PC architecture.
We will talk about the CPU in the next part.
We are not interested about the other characteristics such the different interfaces or the graphical processor, so we are not enumerating those.

2.1.2 Cell Broadband Engine Details

The Cell Broadband Engine [3,4,5] is a CPU which is the result of the cooperation of three companies: IBM, Sony and Toshiba. It has been revealed in february 2005.
Traditional CPUs are able to reorganize the code before the execution. It is helping a lot the programers in their work, and a non optimized code would work in good condition. This require a big part of the processor capacities. In addition, the technology of CPU in not evolving as fast as before. We have almost reached the limits of miniaturization, and the frequency increase really slowly.
The Cell is using an original architecture. It doesn't optimize or reorganize the code before the execution, this is the job of the programers or compilers. Thanks to this, there is more place on the chip to add new execution units.
The next figure shows how is made a CBE:

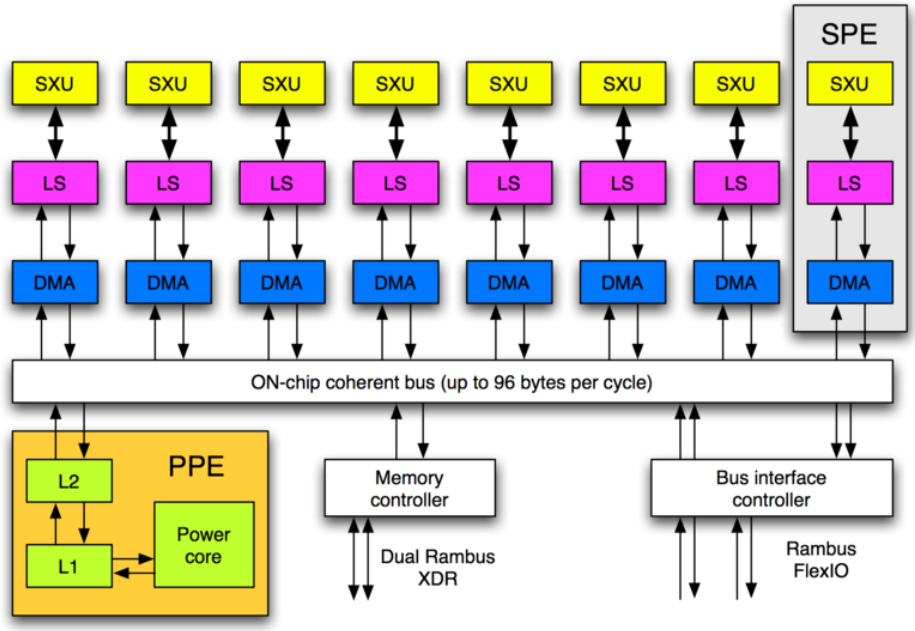


Figure 2.1: CBE Architecture

The CBE is composed of 1 main core, called PPE (PowerPC Processing Element), and 8 other cores called SPE (Synergystic Processing Elements). The PPE is similar to a classical core (64bits), and possesses two levels of cache. The role of the PPE is to manage the SPEs, and is doing the operations that cannot be done easily on the SPEs.
The SPEs are composed of 1 vectorial unit, call SPU (Streaming Processor Unit) and 1 local storage of 256KB.

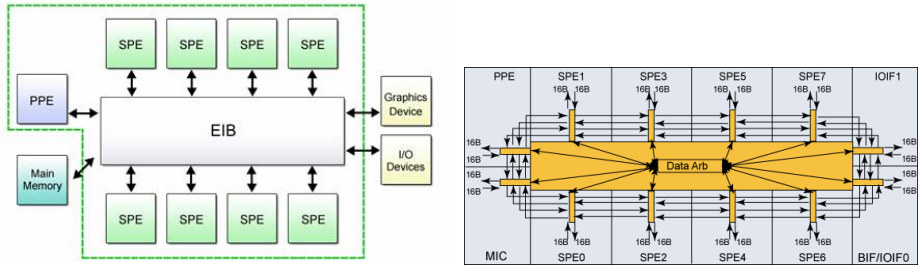


Figure 2.2: Cell Element Interconnect Bus (EIB)

One last interesting point in this architecture: the EIB (Element Interconnect Bus)
The EIB is a bus which is connecting all elements together. It is composed of 4 "loops" of 128bits each, and is able to manage several transfers at the same time.

About the performance, the CBE is able to reach a speed of 20,8GFlops in 64 bits, and 205GFlops in 32 bits with 8 SPEs (25,6GFlops / SPE).
Unfortunately, on the Playstation 3, "only" 7 SPEs are working for a maximal speed of 179,2GFlops in theory.

2.2 Shortest path algorithms overview

In graph theory, the shortest path [6,7,8] problem is the problem of finding the shortest way between two pair of vertices. The path length is determined by the sum of the weights of all the edges in this path.
A good example is to find the quickest way to go from on place to another. The edges would represent the streets and the vertices would represent the intersection of the streets.

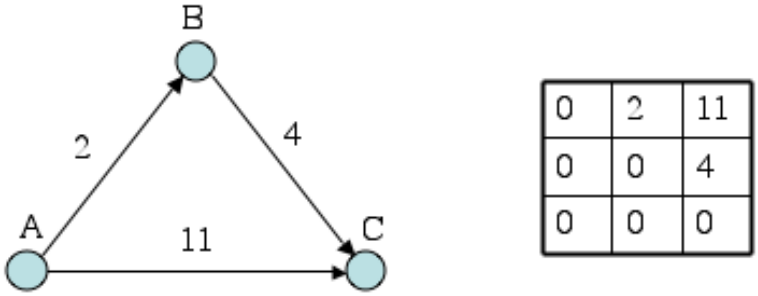
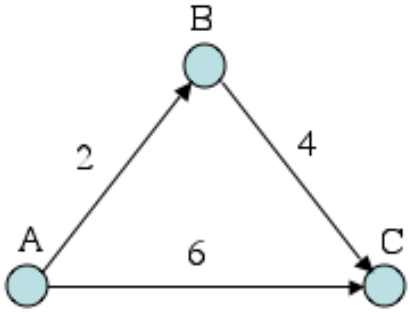


Figure 2.3: Example of Graph and its Weighted Matrix

In this figure, we will try to find the shortest path from A to C. We can see the distance between A to C is 11, but the distance from A to B is 2 and from B to C is 4. So the path from A to C when we go through B is A->B->C = 2 + 4 = 6, and is lower than 11. This is the shortest path is our case.
The new graph and its associated weighted matrix would be:



0	2	6
0	0	4
0	0	0

Figure 2.4: Example of Graph and its Weighted Matrix

2.3 Floyd

The Floyd algorithm [8,9] has been created by Robert Floyd in 1959. Among all the different algorithms, the Floyd algorithm is probably the simplest to understand.
Here is the algorithm:

---

procedure FLOYD_ALL_PAIRS_SP(A)
begin
  $D^{(0)}$ = A;
  for k := 1 to n do
    for i := 1 to n do
      for j := 1 to n do
        $d_{i,j}^{(k)} := \min\,(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)})$;
end FLOYD_ALL_PAIRS_SP

Explanation of the algorithm:
For each vertex k, we check if k is not the shortest way between each pairs of vertices in the matrix.
Lets take this example:

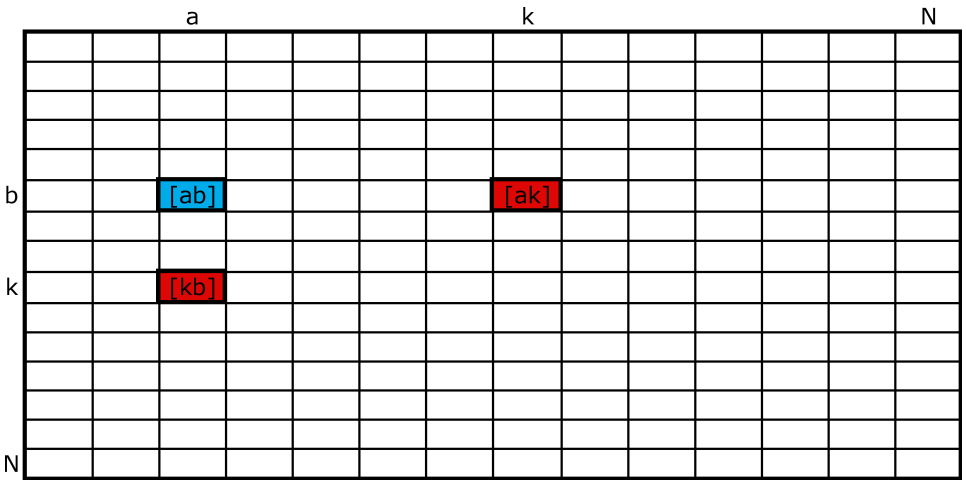


Figure 2.5: Example of Use of Floyd Algorithm

We are checking if the distance between *ab* is not longer than the sum of the distance between *ak* and *kb*. If it's longer, then the new distance is the sum of the distance between *ak* and *kb*.
We are doing the same thing for all the pairs *ab* of the matrix, and once done, we are taking the next value of *k*.
The algorithm is finished when we tried all the values of *k* with all the pairs *ab* of the matrix.

2.4 Dijkstra

The Dijkstra algorithm [8,10] has been created by Edsger Dijkstra in 1959. Here is the algorithm:

```
procedure DIJKSTRA_SINGLE_SOURCE_SP(V, E, w, s)
begin
    V_T := {s};
    for all v ∈ (V - V_T) do
        if (s, v) exists set l[v] := w(s, v)
        else set l[v] := ∞;
    while V_T ≠ V do
    begin
        find a vertex u such that l[u] := min{l[v]|v ∈ (V - V_T)};
        V_T:= V_T∪ u;
        for all v ∈ (V - V_T) do
            l[v] := min{l[v],l[u] + w[u, v]};
```

```
    endwhile
end DIJKSTRA_SINGLE_SOURCE_SP
```

We encountered some difficulties with the Dijkstra algorithm. For this reason, we are not giving details about it.

Chapter 3

Design and Implementation

This chapter describes the details of the way of parallelizing the different algorithms discussed in the previous chapter. Here we will also describe several points of the implementation.

3.1 Floyd

In this section, we showing how we can parallelize the Floyd algorithm presented in the previous chapter.

Parallel Formulation

The Floyd algorithm is a really simple algorithm. Because of that, there are not many way to parallelize it. To do so, we decided to cut the matrix in p "blocks" and to attribute these blocks to the p SPEs available[8].
Here is the way we used to cut into blocks:

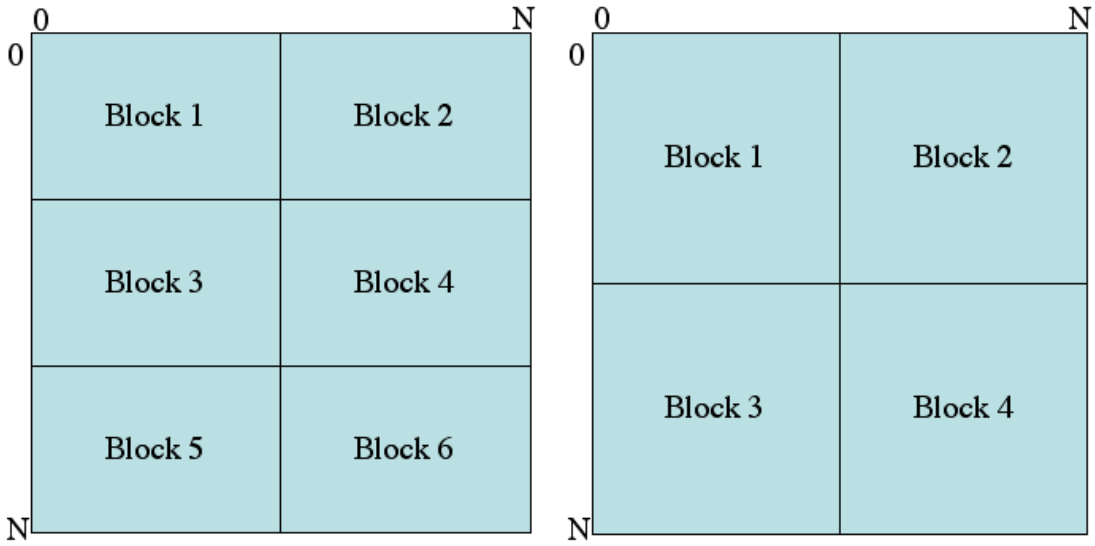


Figure 3.1: Two Examples of Matrix Partitioning for Parallelization

In this figure, we can see the matrix cut into six blocks and each block will be computed by a SPE. That way, the global calculation of the floyd algorithm should be six times faster. Unfortunately, this estimation doesn't take into consideration the "preparation" of those data, the time needed to create the SPE contexts and the communication between the PPE and the SPEs.
Here is a diagram which shows what is needed to communicate between blocks.

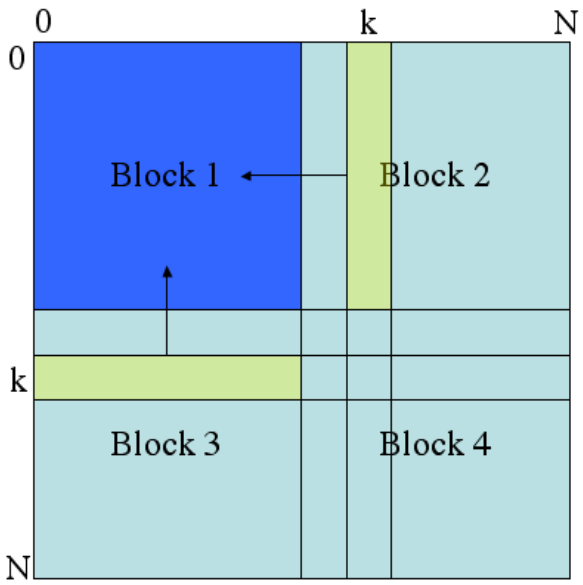


Figure 3.2: Example of Communication between blocks

As we can see on this diagram, each block will receive the line and the column of another block. It's corresponding to the line and column of k, which will be required to process the calculation of the next iteration of k. The two next diagrams summarize how are working the programs(PPE and SPEs):

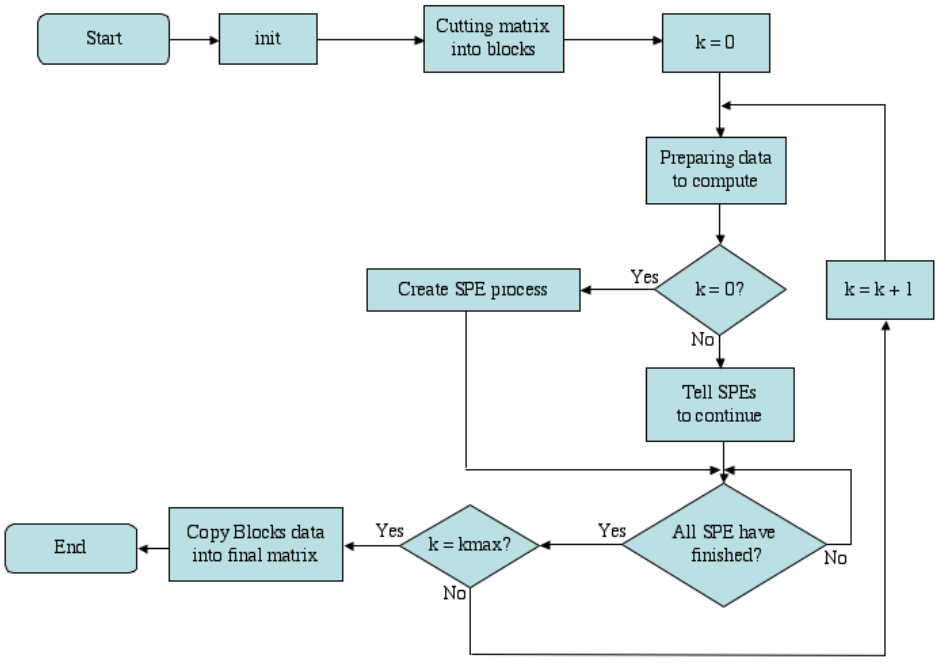


Figure 3.3: Floyd PPE Parallel Algorithm

Init First, we are initializing the program. During this step, the program is creating all the variables needed, allocating the aligned memory for structures and tables we will send to the SPEs, and we are generating a pseudo-random matrix to test the program.

Cutting matrix into blocks The second step is one of the most important: the preparation of blocks. Graphically, it seems to be really easy and fast to cut a block, but unfortunately, for a computer, it is a little more complicated. For each block, we need to determine the limits, and to copy the data. Copying the data takes time. To copy all the data, we are using a N^2 algorithm.

k = 0 Initializing k for the first iteration.

Preparing data to compute This is the step which extract the line k and the column k from the appropriate block. Currently, the algorithm used, for 6 SPEs working, is a $5N$ algorithm ($4N$ when we use only 4 SPEs), but it is launched for each iteration of k. So the total cost of this part for the program is $5N^2$ ($4N^2$ when we use only 4 SPEs). This can be easily optimized to a N^2 algorithm.

k = 0? It is just a test to know if it is the first iteration or not. If it is the first iteration, the program will have to create the SPE contexts. Otherwise, it will just send a message to the SPEs which are waiting to continue.

Create SPE process Here, we are creating a structure which contains all the required informations for the SPE. Then, we are creating the process and giving this structure as parameter. This structure contains the limits of the block, the size, and the address of the data. The SPE will access those data using the DMA (Direct Memory Access).

Tell SPE's to continue The SPE were waiting for the next step. Once all the SPEs finished their job and the PPE finished to prepare the data, a message is sent to all SPEs to continue their work for the next iteration.

All SPE have finished? To continue to the next step, all the SPE have to be done with the calculation.

k = kmax? Did we finish the calculation?

k = k + 1 Next iteration of k if we didn't finish.

Copy blocks data into final matrix The calculation is done, now the program copy the content of all blocks into a result matrix. The algorithm used is an N^2 algorithm.

End During this step, the PPE destroys all the SPEs contexts and the allocated memory. For the tests, we also use the "regular" Floyd algorithm during this test, and we compare the results with the matrix we obtained with the parallelized Floyd. This way, we are checking if the algorithm is giving the good results.

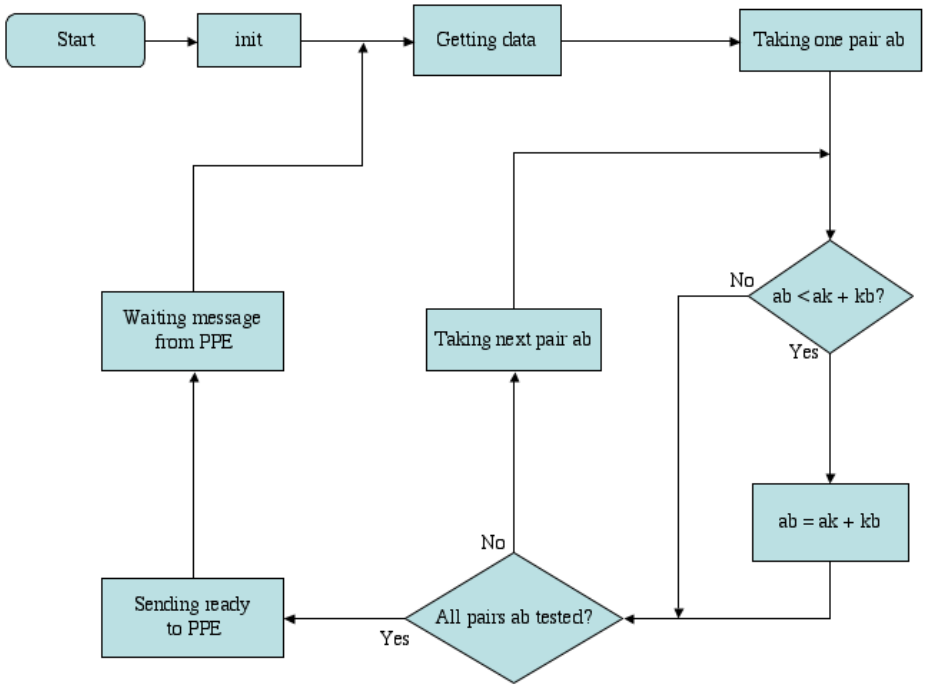


Figure 3.4: Floyd SPEs Parallel Algorithm

The program of the SPEs is much easier to understand.

Init First, we are initializing the program. During this step, the program is creating all the variables needed, allocating the memory for structures and tables we will receive from the PPE.

Getting data Then, we are getting the data in the memory by DMA.

Taking one pair ab The program is picking a pair *ab* to make the tests.

ab < ak + kb? Is the way *ab* shorter than the sum of the ways *ak* and *kb*?

ab = ak + kb If the new way is shorter, we store it.

All pairs tested? Did we finish?

Taking next pair ab If we didn't finish, we are picking the next pair *ab*

Sending ready to PPE If we finish, the SPE send a message to the PPE to notify we are ready for the next step.

Waiting message from PPE The SPE is waiting the notification from the PPE to continue the work.

3.2 Dijkstra

In this section, we showing how we can parallelize the Dijkstra algorithm presented in the previous chapter.

Parallel Formulation

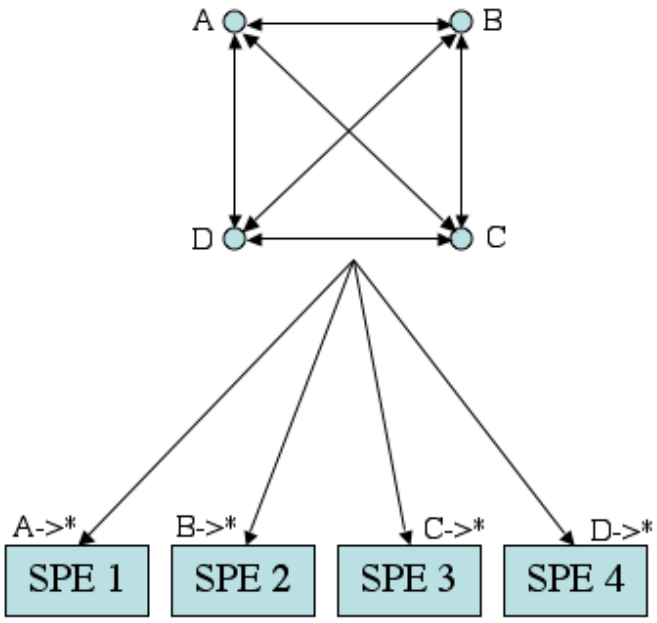


Figure 3.5: Parallel Formulation of Djikstra Algorithm

The Dijkstra algorithm is able to give the shortest path between on vertex and the rest of the matrix. One solution to parallelize the Dijkstra algorithm is to launch the algorithm on the SPEs and to calculate the shortest path between a different vertex on each SPE.

3.3 Implementation

For the implementation of the algorithms for the CBE, we need to synchronize the PPE and the SPEs, and to communicate between these. Due to the architecture specificities, we had to find answer to these questions:

- How to transmit the data?

- How should be the memory for DMA?
- How to synchronize the PPE and the SPEs?

Transmission of Data

There are several way to communicate between SPEs and PPE. In our implementation, we are communicating with three different ways:

Using the parameters when we create the SPE contexts When the PPE is creating SPE contexts, we have the possibility to give a structure as argument. There are some constrains to respect with this structure:

- The size of the structure should be 1, 2, 4, 8, 16 or any multiple of 16 bytes
- The maximum size of the structure is 16 Kbytes.

Accessing directly the memory by using DMA This way of accessing memory [11->16] is really useful to communicate between SPEs and PPE, or just to have more than the 256 Kbytes available for each SPE. In the structure given in argument during the creation of the SPE context, it is possible to put a pointer and to access it later. To do so, we are using the function *SPE_mfcdma32*. This function allow to read of write in memory using DMA. But to use it, there is one main condition to respect: the allocated memory we want to access from the SPE should be 16 bytes aligned.

Using the mailboxes The mailboxes are used only for the synchronization and will be explained further in this report.

Allocation of memory

As we said in the previous part, the memory accessed by the SPEs needs to be 16 bytes aligned. To align the memory, we are processing this way:

```
void * malloc_align(size_t size, int n, void * origPointer)
{
   void * abe;
   int i;
   for(i = 1; i < size; i* = 2);
   origPointer = malloc(i * sizeof(int) + 8 * sizeof(int));
   abe = origPointer;
   while((int)abe&0x7f){
      ++abe;
   }
```

```
    return abe;
}
```

As you can see, instead of allocating the size of the memory, we are allocating the 2^n higher than what was required. We are doing this to avoid crashes when the SPE is accessing the memory. Because of this, in some cases, we are loosing memory and the time for allocation might be a little longer. This point is one of the parts we might be able to improve.

Synchronization

In order to manage the SPEs correctly, the PPE needs to be synchronized with these[15,16]. To do it, we are using the mailboxes available in the CBE architecture.
The mailboxes allow the SPE to send or receive messages of 1 byte (FIFO).

Chapter 4

Testing and Benchmarks

This chapter presents the results of the tests that were carried out to measure the performance of the implementation. Of course, all the programs which generated these results are validated and give the best solution for the application of the shortest path algorithm.

4.1 Test setup

This section explains how the performance are measured, and give details about the validation.

Validation

The first step to test an algorithm is to test if this algorithm is giving the good result. To do so, we implemented the Floyd algorithm at the end of the programs we wanted to validate to check if the results are the same. We choose to use Floyd because of its simpleness.

Measurements

The second step is to measure the performances[17,18]. To measure the time required, we are using the function *gettimeofday* which is able to give the actual time with a precision of $\sim 1\mu$s. There is also another possibility if we needed a precision of more than 1ns: to use the time base register. The time base register is giving the actual CPU time. If the CPU frequency is 1GHz, then the precision would be 1/1GHz = 1ns. (in our case with a CPU at 3,2 GHz, the precision would be ∼0,3215ns in theory).
To measure the results, we are first changing some parameters (like the matrix size, number of SPEs, ...) in a file which is include in the program, we are recompiling it, and we are launching the program 100 times. The results are stored in a file, and an average is calculated from this. The parameters we are changing are the number of SPE we are using, and the size of

N. To do this, we made a shell script which is doing everything automatically:

```
♯!/bin/bash

maxrand = 1000
y = 2

rm result.txt
echo"x; y; N; maxrand; result" > result.txt
for((x = 2; x < 4; x + +))
do
   for ((n = 4;  n < 129;  n + +))
   do
      rm define.h
      echo "♯define MAX_BLOC_Y $y" >> define.h
      echo "♯define MAX_BLOC_X $x" >> define.h
      echo "♯define N $n" >> define.h
      echo "♯define MAX_RAND $maxrand" >> define.h
      make clean
      make
      for ((i = 0; i < 100; i + +))
      do
         echo "$x; $y; $n; $maxrand; `./cell_prog|greptotal`" >> result.txt
      done
   done
done
```

To execute this script with the parallelized Floyd, we need 20 minutes and we get a 1,2MB result file.

4.2 CBE Testing

To understand the capacities of the CBE, some programs were created to measure these points:

- Time required to create and launch a SPE context
- Speed for SPE to read in memory with DMA
- Speed for SPE to write in memory with DMA

Time required to create and launch a SPE context

In one of the first version of the parallelization of the Floyd algorithm of this project, the program was creating the SPE contexts for each iteration of k and destroying it once the calculation was done. In the end, we created *6*N* contexts (6 SPEs, N iterations of k). When we compared the results of the parallelized program and the regular implementation of the Floyd algorithm, we realized that the parallelized one was 15 times slower. Then, we decided to understand why.
For this test, we created and destroyed 1000 contexts with only the reading the arguments given to SPEs, and we calculated the average time. In the end, to create a context and to destroy it, we need 1,27ms. It's easy to understand why the initial Floyd parallelized program was much slower than the regular one. To compare, when 1,27ms is the time required to create and destroy a context which is not doing anything, 1,27ms is also the time required to apply the not parallelized Floyd algorithm on a matrix with *N=28*.

Speed for SPE to read or write in memory with DMA

In our quest to optimize the execution to its maximum, we tried to measure the speed of reading or writing using a DMA. Unfortunately, we don't have relevant results. To measure it, we created a program which was allocating memory, and once done, we checked how much time was required for the SPE to read all this memory, or to write in it. The results didn't have coherence. We measured that for reading 32Mbytes, only 25ms are required (more than 1,2 GBytes/s), but this delay can change without any apparent reason and is far from the theory(25,6 Gbytes/s). We can say for sure that the speed of the communication is fast enough for our case.

4.3 Floyd

To measure the improvement thanks to the parallelization, we need first a reference. So we made experiments on the "regular" Floyd algorithm and we measured the performances:

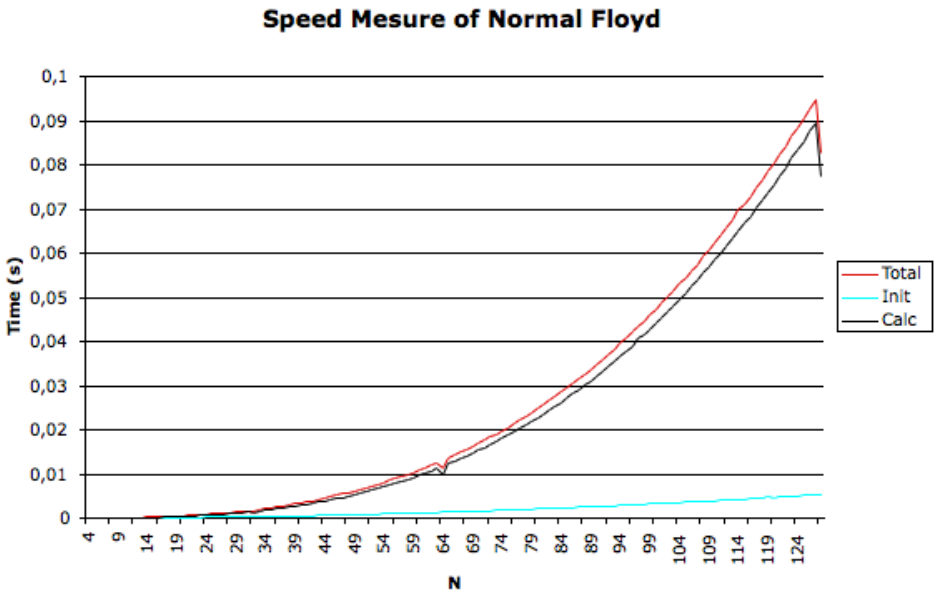


Figure 4.1: Results

On this graph, we can see the tree measurements we made on the "regular" Flyod program.
The first one is the initialization, which correspond to the generation of the pseudo random matrix used for the test.
The second one is the calculation of the Floyd algorithm.
The third one is the sum of the two others.
As we can see on this graph, the floyd algorithm is really fast when the matrix is small, but is increasing really quickly. It was expected because the Floyd algorithm is a N^3 algorithm.
For N=4,8,16,32,64,128, we can see on the graph a decrease of the calculation time. We can guess it is probably due to an hidden optimization made by the compiler.

This second graph is showing the difference of speed between the "regular" Floyd algorithm, and two parallelized algorithm: the first one using 4 SPEs (2x2), and the second one using 6 SPEs (2x3).

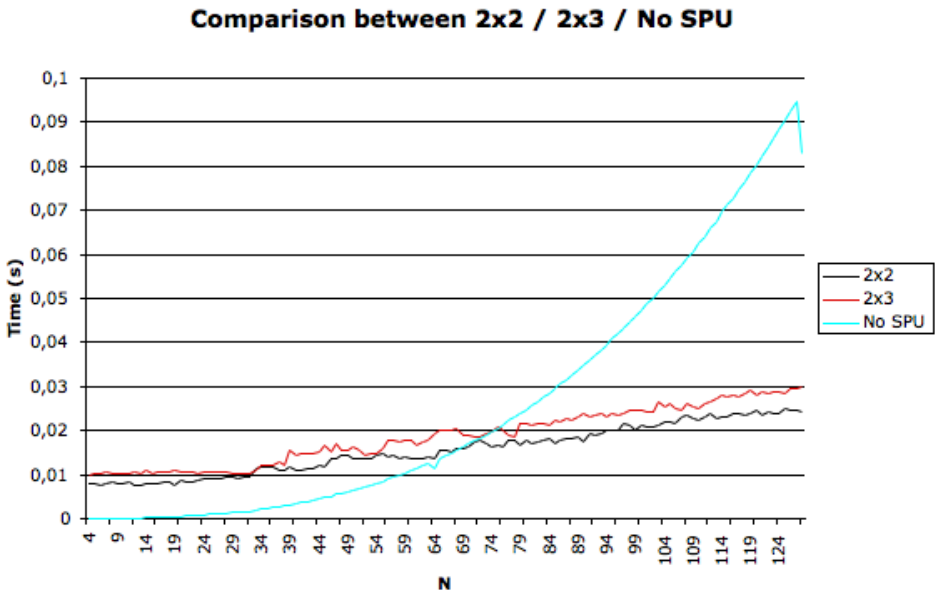


Figure 4.2: Results

On the first part of the graph ($N <\sim 70$), we can see the "regular" Floyd is faster than the parallelized one. The parallelized program needs to create contexts for the SPEs, and each creation takes 1,27ms. For the 2x2 program, we are "loosing" 5,08ms to create those , and 7,62ms for the 2x3 program. In addition, the program is also using several algorithms which are slowing down the program with a small matrix size.
On the second part, of the graph, the launching time of "regular" Floyd is increasing faster and faster due to its N^3 algorithm. But when we are looking at the parallelized ones, the time needed for the calculation is not increasing quickly. Because the SPEs are exclusively used by the program when the PPE is shared with the operating system and the different services launched, we can suppose the calculation is faster on SPEs than on PPE.
There is also something interesting on this graph: the program using only 4 SPEs is faster than the program using 6 SPEs. This fact might change with some of the improvements in the Chapter 5.

Chapter 5

Improvements

The aim of this chapter is to give an overview of what could be done to improve the actual solution.

Cutting blocks differently

When we are cutting the blocks in the way we implemented, we are loosing time by calculating the index of the element we are ready or writing in a matrix. We can find many occurrence of this case in the Floyd program, and we might gain some acceleration by cutting the matrix differently to calculate the indexes more easily.
Here is the solution to cut the blocks which might improve the most this problem:

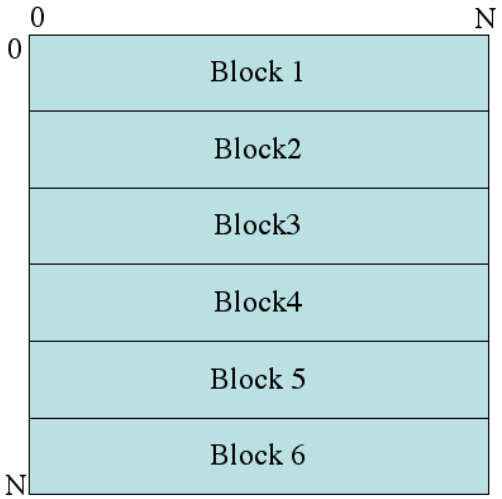


Figure 5.1: Example of Alternative Matrix Partitioning for Parallelization

For example, with a matrix of size N, and with p SPEs, the index would be:

- Block 1: $0 \leq index \leq \frac{N^2}{6} - 1$
- Block 2 : $\frac{N^2}{6} \leq index \leq \frac{N^2}{3} - 1$
- Block 3 : $\frac{N^2}{3} \leq index \leq \frac{N^2}{2} - 1$
- Block 4 : $\frac{N^2}{2} \leq index \leq \frac{2N^2}{3} - 1$
- Block 5 : $\frac{2N^2}{3} \leq index \leq \frac{5N^2}{6} - 1$
- Block 6 : $\frac{5N^2}{6} \leq index \leq \frac{N^2}{6} - 1$

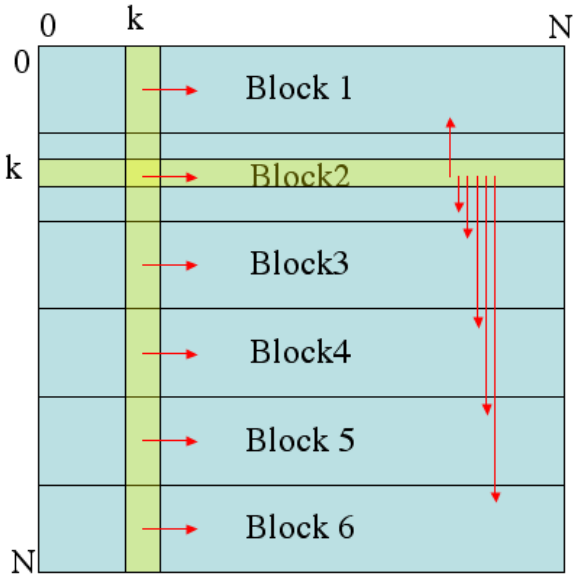


Figure 5.2: Example of Communication in Alternative Matrix Partitioning for Parallelization

Calculating the line and col during the waiting of SPUs of previous iteration

In our actual implementation, we are preparing the line and column just before sending it. But if we do that, the SPEs will wait. It is possible to prepare it for the next iteration when the PPEs are still working.
The program would work this way: (PPE):

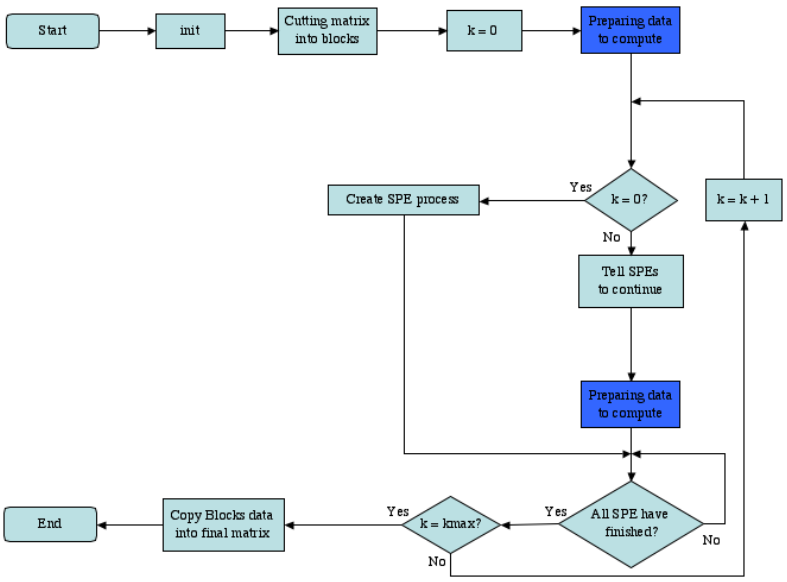


Figure 5.3: Example of calculating instead of waiting

Cutting line and col only once for all the SPUs

With our actual implementation, we are copying the line and the column required for each block. For example, when we use 6 SPEs (2x3), each block needs 1 line and 1 column (total size : $\frac{N}{2} + \frac{N}{3}$) for each iteration. In the end, this copy required at least $(\frac{N}{2} + \frac{N}{3}) * 6 * N = 5N^2$ loops.
But several blocks need the same data. If we optimize this part, the cost will decrease to N^2 loops.

Chapter 6

Conclusion and Future Work

Since the problem statement exposed in the first chapter, we are able to understand much better the Cell Broadband Engine architecture, and the parallelization of the algorithms. The project revealed some unexpected results, like the better performances of the Floyd algorithm with 4 SPEs than with 6 SPEs. We also expected to have the Floyd parallelized algorithm faster than the regular one with a N much lower than we saw on the results. The regular Floyd is at least twice faster until N=50, and the parallelized ones start to be faster for N=65.
The Chapter 5 could be a good start to improve the results, but even with these modification, the implementation of the Floyd algorithm might not be interesting enough for an application in UPPAAL.

Chapter 7

References

[1] www.uppaal.com
Kim G. Larsen (Professor, AAL), Wang Yi (Ph.D., Professor, UPP), Gerd Behrmann (Associate Professor, AAL), Paul Pettersson (Associate Professor, UPP), Alexandre David (Post Doc, AAL), Brian Nielsen (Associate Professor, AAL), Arne Skou (Associate Professor, AAL), John HŒkansson (Ph.D. Student, UPP), Jacob Illum Rasmussen (Ph.D. Student, AAL), Pavel Krc‡l (Ph.D. Student, UPP), Ulrik Larsen (Ph.D. Student, AAL), Marius Mikucionis (Ph.D. Student, AAL), and Leonid Mokrushin (Ph.D. Student, UPP).

[2] http://playstation.about.com/od/ps3/a/PS3SpecsDetails_3.htm
Roger Altizer (PlayStation Games Guide)

[3] http://www.presence-pc.com/tests/Le-processeur-Cell-366
(Dead link)

[4] http://fr.wikipedia.org/wiki/Cell_(processeur)
Multiple contributors

[5] http://en.wikipedia.org/wiki/Cell_(microprocessor)
Multiple contributors

[6] http://en.wikipedia.org/wiki/Shortest_path_problem
Multiple contributors

[8] Introduction to Parallel Computing
Ananth Grama (Associate Professor, Purdue), Anshul Gupta (Research staff IBM T.J Watson Research Center), George Karypis (Associate Professor, Minnesota), Vipin Kumar (Professor, Minnesota).

[9] http://en.wikipedia.org/wiki/Floyd-Warshall_algorithm
Multiple contributors

[10] http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
Multiple contributors

[11] http://www.power.org/resources/devcorner/cellcorner/codesample2

[12] http://www.renderstate.de/?p=6

[13] http://www.cc.gatech.edu/ bader/CellProgramming.html
David A. Bader (Professor, Atlanta)

[14] http://www.ibm.com/developerworks/library/pa-tacklecell1/
Peter Seebach (IBM)

[15] http://www.cag.csail.mit.edu/ps3/cellminiref.shtml
Massachusetts Institute of Technology

[16] http://crca.ucsd.edu/cellworkshop/Slides/
IBM Corporation

[17] http://www.ibm.com/developerworks/linux/library/pa-timebase/index.html
Carlos Cavanna (Software Developer)

[18] http://www.hlrn.de/doc/performance/programming.html♯MPI_timers