Using SystemC to Model and Simulate Many-Core Architectures

Ana Rita Silva¹, Wilson M. José¹, Horácio C. Neto², Mário P. Véstias³
INESC-ID¹: INESC-ID/IST/UTL²; INESC-ID/ISEL/IPL³
anaritasilva@esda.inesc-id.pt, wilson@esda.inesc-id.pt, mvestias@deetc.isel.pt, hcn@inesc-id.pt

Abstract

Transistor density has made possible the design of massively parallel architectures with hundreds of cores on a single chip. Designing efficient architectures with such high number of cores is a very challenging task. Simulation of many-core architectures can help designers to explore the design space.

This paper addresses the applicability of SystemC to simulate many-core architectures. We demonstrate the use of SystemC to model a system of $P$ processors executing matrix multiplications. The simulation of the model allows analyzing the results regarding the number of transfers and the number of clock cycles required to complete each transaction.

1. Introduction

During the last decade massively parallel systems have been proposed as high-performance computing architectures delivering very high computing speeds which make them particularly attractive for scientific applications.

Whether in Application Specific Integrated Circuits (ASIC) or Field Programmable Gate Array (FPGA), these are very complex systems whose simulations must be done at system level since Register-Transfer Level (RTL) simulations are two or three orders of magnitude slower. Even so, only a few existing simulators are able to sustain many-core architectures with acceptable simulation times. The COTSon team at HP labs proposed a trace-driven simulator fed with thread instruction streams computed by a single-core full system simulator [1]. The approach only considers an idealized architecture with a perfect memory hierarchy, without any interconnect, caches or distribution of memory banks.

Most of the other recent approaches parallelized discrete-event simulators with varying levels of detail. SlackSim [2] is a cycle-level simulator allowing individual cores to progress at different paces in a controlled manner. In [3], the authors proposed a system level architectural modeling methodology for large designs based on SystemC. The proposed methodology uses normal function calls and SystemC method processes to implement a very fast but accurate model to evaluate a large design at system level. The platform was used to model a NoC-based SoC. The experimental results show improvement up to 98% in elaboration time and up to 90% in simulation time for small size NoCs.

SystemC is a system design language that has evolved in response to a pervasive need for improving the overall productivity for designers of electronic systems [4].

One of the primary goals of SystemC is to enable system-level modeling — that is, modeling of systems above RTL, including systems that might be implemented in software, hardware, or some combination of the two [5].

The higher level of abstraction gives the design team a fundamental understanding, early in the design process, of the behavior of the entire system and enables better system tradeoffs, better and earlier verification, and overall productivity gains through reuse of early system models as executable specifications [4].

SystemC is based on the C++ programming language, which is an extensible object oriented modeling language. It extends the C++ data types with additional types useful for modeling hardware that support all the common operations and provide special methods to access bits and bit ranges.

SystemC adds a class library to C++ to extend its capabilities, effectively adding important concepts such as concurrency, timed events and data types. This class library is not a modification of C++, but a library of functions, data types and other language constructs that are legal C++ code [6].

This work presents a SystemC design specification and model implementation of a 2D-array multiprocessor system executing matrix multiplications using $P$ processors.

The rest of the paper is organized as follows. The next section describes an overview of SystemC. Section 3 describes the parallel dense matrix multiplication algorithm proposed. Section 4 describes the system-level model of the architecture. Experimental results are discussed in Section 5. Finally, Section 6 concludes the paper.
2. Overview of SystemC

Modules are the basic building blocks for partitioning a design in SystemC. They allow designers to break complex systems into smaller, more manageable pieces, and to hide internal data representation and algorithms from other modules [5].

Fig. 1. Module structure.

A typical module contains processes that describe the functionality of the module, ports through which the module communicates with the environment, internal data and channels for maintenance of model state and communication among the module’s processes, among other modules (figure 1 sketches a module structure).

A SystemC module is simply a C++ class definition and is described with the \texttt{SC\_MODULE} macro (figure 2).

Within the class definition, ports are usually the first thing declared because they represent the interface to the module. Local channels and sub-module instances come after that.

Next, we place the class constructor. Each module requires a constructor block (\texttt{SC\_CTOR}), which maps designated member functions to processes and declares event sensitivities; its argument must be the name of the module being declared. The header file finishes out with the declarations of processes, helper functions and other private data.

In the example of figure 2, the definitions of the process reside in a separate file from the module declaration. The traditional template places all the instance creation and constructor definitions in header (.h) files. Implementation of processes and helper functions are deferred to the compiled (.cpp) file.

In SystemC the basic unit of functionality is called a process. A process must be contained in a module – it is defined as a member function of the module and declared to be a SystemC process in the module’s constructor [5]. The processes within a module are concurrent.

An event is an object, represented by class \texttt{sc\_event}, that determines whether and when a process’s execution should be triggered or resumed; it has no value and no duration. An event is used to represent a condition that may occur during the course of simulation and to control the triggering of processes accordingly [5]. We can perform only two actions with a \texttt{sc\_event}: wait for it or cause it to occur.

The owner of the event is responsible for reporting the change to the event object. The act of reporting the change to the event is called \texttt{notification}. The event object, in turn, is responsible for keeping a list of processes that are sensitive to it. Thus, when notified, the event object will inform the scheduler of which processes to trigger [5].

SystemC has two kinds of processes: method processes, defined with the macro \texttt{SC\_METHOD}, and thread processes, defined with the macro \texttt{SC\_THREAD}.

\texttt{SC\_THREAD} processes are started once and only once by the simulator. Once a thread starts to execute, it is in complete control of the simulation until it chooses to return control to the simulator. Variables allocated in \texttt{SC\_THREAD} processes are persistent.

SystemC offers two ways to pass control back to the simulator. One way is to simply exit (e.g., return), which has the effect of terminating the thread for the rest of the simulation. When an \texttt{SC\_THREAD} process exits, it is gone forever, therefore \texttt{SC\_THREADs} typically contain an infinite loop containing at least one \texttt{wait}. The other way to return control to the simulator is to invoke the module wait method.

---

// File example.h //
#include <iostream>
#include <systemc.h>

SC_MODULE (example)
{
    sc_in <bool> clk;
    sc_in <bool> port_in;
    sc_out <bool> port_out;

    SC_CTOR (example) {
        SC_METHOD (example_method);
        sensitive_pos << clk;
    }

    void example_method();
};

// File example.cpp //
#include "example.h"

void example:: example_method (){ 
     port_out=port_in;
}

Fig. 2. Code example.
The `wait` (dynamic sensitivity list) suspends the SC_THREAD process [4].

When `wait` executes, the state of the current thread is saved, the simulation kernel is put in control and proceeds to activate another ready process. When the suspended process is reactivated, the scheduler restores the calling context of the original thread, and the process resumes execution at the statement after the `wait` [4].

During simulation a thread process may suspend itself and designate a specific event `event_name` as the current event on which the process wishes to wait. Then, only the notification of `event_name` will cause the thread process to be resumed.

SC_METHOD processes never suspend internally. Instead, SC_METHOD processes run completely and return. The simulation engine calls them repeatedly based on the dynamic or static sensitivity list.

In terms of dynamic sensitivity list, SC_METHOD processes may not call the `wait` method, because they are prohibited from suspending internally. Instead of calling `wait()`, a method process calls `next_trigger()` to specify the event that must occur for it to be triggered next time.

Until the event occurs, the static sensitivity list is temporarily disabled. Unlike `wait()`, however, calling `next_trigger()` does not suspend the current method process. Instead, execution of the method process continues to the end, and next time the method process will be invoked only when the event specified by `next_triggered()` occurs [5].

SystemC provides another type of sensitivity for processes called static sensitivity. Static sensitivity establishes the parameters for resuming before simulation begins.

Like method processes, a thread process may have a sensitivity list describing the set of events to which it should normally react. As mentioned, when we encounter a `wait()` statement, the execution of a thread process is suspended. When any of the events in the sensitivity list occurs, the scheduler will resume the execution of the process from the point of suspension [5].

Interfaces, ports and channels are the way through which SystemC implements synchronization and communication at system level.

A SystemC interface is an abstract class that inherits from `sc_interface` and provides only pure virtual declarations of methods referenced by SystemC channels and ports. No implementations or data are provided in a SystemC interface.

A SystemC channel is a class that implements one or more SystemC interface classes and inherits from either `sc_channel` or `sc_prim_channel`. SystemC has two types of channels: primitive and hierarchical.

SystemC’s primitive channels are known as primitive because they contain no hierarchy, no processes, and are designed to be very fast due to their simplicity. All primitive channels inherit from the base class `sc_prim_channel` [4]. The simplest channels are `sc_signal`, `sc_mutex`, `sc_semaphor`, and `sc_fifo`.

In this work will use `sc_fifo` channels to model the communication between processors and memories. `sc_fifo` is probably the most popular channel for modeling at the architectural level. First-in first-out queues are a common data structure used to manage data flow; by default, an `sc_fifo<>` has a depth of 16. The data type of the elements also needs to be specified. An `sc_fifo` may contain any data type including large and complex structures.

Two interfaces, `sc_fifo_if<>`, and `sc_fifo_out_if<>`, are provided for `sc_fifo<>`. Together these interfaces provide all of the methods implemented by `sc_fifo<>`.

A SystemC port is a class template with and inheriting from a SystemC interface. The port base class is called `sc_port`. A port is an object through which a module, and hence its processes, can access a channel’s interface. A channel cannot be connected to a port if it doesn’t implement the port’s interface. There are three types of ports: in, out or inout. Each port has a data type, passed on between the angle brackets (<, template class).

At last, the SystemC library provides its own definition of `main()`, which in turn calls `sc_main()`. Within `sc_main()`, code executes in three distinct major phases, which are elaboration, simulation, and post-processing.

Elaboration establishes hierarchy and initializes the data structures. Elaboration consists of creating instances of clocks, design modules, and channels that interconnect designs. At the end of elaboration, `sc_start()` invokes the simulation phase. During simulation, code representing the behavior of the model executes.

Finally, after returning from `sc_start()`, the post-processing phase begins. Post-processing is mostly optional. During post-processing, code may read data created during simulation and format reports or otherwise handle the results of simulation. Post-processing finishes with the return of an exit status from `sc_main()` [4].
3. Matrix Multiplication Algorithm

In this section, we present the algorithm proposed to parallelize the matrix multiplication problem for a system with $p$ processors organized as a 2-dimensional array.

The parallel architecture is organized as a 2D mesh of execution (see figure 3). Each core unit consists mainly of a floating-point multiply and accumulate unit (FPMAC) and a dual-port memory. Access to the external memory is controlled with a direct memory access (DMA) module that can deliver burst transactions with one transfer per cycle.

To facilitate the presentation of the algorithm, we consider that the mesh array is square and define matrix $C$ to be the result of the product between two square matrices, $A$ and $B$. The results can be easily generalized to nonsquare meshes. We also consider that all matrices involved are square and have the same size $n \times n$, and that its dimensions are multiple of the partitioned blocks dimensions. Again, this consideration does not limit in any way the generality of the results.

Each of the $p = q \times q$ processors, where $q = \sqrt{p}$, is responsible for calculating one block of matrix $C$ with size $\frac{n}{q} \times \frac{n}{q}$. Each one of these blocks is partitioned, according to the memory limitations of the processor, in sub blocks $C_{ij}$ with size $y \times x$.

To generate a $C_{ij}$ block, the processor must multiply a $y \times n$ block of matrix $A$ with a $n \times x$ block of matrix $B$. This multiplication is segmented as a sequence of $k_0 = \frac{n}{z}$ block multiplications as specified in equation (1).

$$C_{ij} = \sum_{k=1}^{k_0} A_{ik} \times B_{kj}$$

Therefore, each partial block multiplication consists of the multiplication of a $y \times z$ sub block $A_{ik}$ with a $z \times x$ sub block $B_{kj}$ resulting in a partial sub block $C_{ik}$ of size $y \times x$. The final $C_{ij}$ result is obtained after accumulating the $k_0$ partial block multiplications.

Figure 4 illustrates how the system performs the product between the matrices $A$ and $B$ considering four processors and 16 sub-blocks (4 per processor).

The processor $P1$ is responsible for calculating blocks $C_{11}$, $C_{12}$, $C_{21}$ and $C_{22}$, processor $P2$ for blocks $C_{13}$, $C_{14}$, $C_{32}$ and $C_{33}$, processor $P3$ for calculating blocks $C_{15}$, $C_{16}$, $C_{51}$ and $C_{52}$, and processor $P4$ for blocks $C_{35}$, $C_{36}$, $C_{43}$ and $C_{44}$. The order in which the blocks of matrix $C$ are calculated corresponds to the equations defined in figure 4 (for the first “iteration”).

To calculate the four blocks, $C_{11}$, $C_{13}$, $C_{15}$, $C_{33}$, it is only necessary to transfer the data from two rows of blocks $A$ and two columns of blocks $B$. In fact, all the processors in the same row require the same data from matrix $A$, while all the processors in the same column require the same data from matrix $B$. Therefore, each sub block fetched from memory is broadcast to a row (column) of $\sqrt{p}$ processors.

The total number of communications is given by equation (2).

$$N_{comm} = \frac{n^3}{\sqrt{p}} \left( \frac{1}{x} + \frac{1}{y} \right) + n^2$$

The number of communications does not depend on the dimension $z$ of the sub blocks from matrix $A$ and matrix $B$, thus $z$ can be simply made equal to 1 in order to minimize the local memory required.
The partial block multiplication is implemented such that, each processor receives the sub block $A_{ik}$ with $y$ elements and stores it. Then it receives the elements of the sub block $B_{kj}$ which are multiplied the corresponding locally stored elements of $A_{ik}$ resulting in a partial sub block $C_{ij}$. This process repeats $n$ times, after which the final sub block $C_{ij}$ is obtained.

The total number of computation cycles, assuming a processor throughput of one accumulation/cycle, is given by

$$N_{comp\text{cycles}} = \frac{n^3}{p}$$  \hspace{1cm} (3)

where $n^3$ is the total number of multiply-add operations to be performed by the $p$ processors.

The minimal execution time is achieved when all the communications, except the initial and final overhead, can be totally overlapped with the computations, that is, when the number of communication cycles required is lower than the number of computation cycles, $N_{comp} > N_{comp\text{cycle}}$.

If there is full overlap, the total execution time is given by

$$N_{exec\text{cycles}} = \frac{n^3}{p} + Ovhd_{INI} + Ovhd_{END} = \frac{n^3}{p}$$  \hspace{1cm} (4)

The initial overhead, $Ovhd_{INI}$, corresponds to the number of cycles it takes until all data becomes available for the last processor(s) to initiate the computations (that is, when the first element of the last required block of $B$ arrives). The final overhead, $Ovhd_{END}$, corresponds to the additional number of cycles needed to write back the very last blocks of $C$.

These initial and final communication overheads are negligible (for large matrices) and, therefore, are not detailed herein.

### 4. Architecture model in SystemC

This section describes the implementation of the system model using SystemC. For ease of explanation, we describe the approach considering only four processors.

The model consists of three modules, where modules $Matrix$ and $Result$ represent the DMA and module $Processors$ represents the 2D-Array (figure 5).

The communication between modules is performed by FIFOs of type $sc\_fifo$. Each FIFO has only one direction, so it is necessary one FIFO to communicate with module $Result$ and, for each processor, one FIFO to communicate with module $Matrix$.

![Fig. 5. SystemC model of the many-core architecture.](image)

Module $Matrix$ stores the values of the matrices $A$ and $B$ and consists of two thread processes ($SC\_THREAD$) that are responsible for sending, in the correct order, the data to the processors through FIFOs (figure 6).

```cpp
(...)
SC_MODULE(matrix) {
    sc_fifo out <int> out_matrix1;
    sc_fifo out <int> out_matrix2;
    sc_fifo out <int> out_matrix3;
    sc_fifo out <int> out_matrix4;
    int* A, *B;
    sc_event MA_event, MB_event;
    void matrixA_thread();
    void matrixB_thread();
    SC_CTOR(matrix) :
        out_matrix1("out_matrix1")
        out_matrix2("out_matrix2")
        out_matrix3("out_matrix3")
        out_matrix4("out_matrix4")
    {
        SC_THREAD(matrixA_thread);
        SC_THREAD(matrixB_thread);
        if ((A = (int*) malloc((total*total) * sizeof(int))) == NULL) {
            printf("Out of memory\n");
        }
        if ((B = (int*) malloc((total*total) * sizeof(int))) == NULL) {
            printf("Out of memory\n");
        }
    }
};
```

Fig. 6. File Matrix.h.

The first process is responsible for sending data from matrix $A$ and the second for sending data from matrix $B$. The modules send the data words one at a time, starting by sending one block $A$ followed by one block $B$ and so on.

The module $Matrix$ stops and waits, whenever the module $Processors$ wants to send data to module $Result$. The synchronization of the two processes is maintained by two events ($sc\_event$) triggered by the respective threads.

The module $Processors$ consists of $p$ thread processes ($SC\_THREAD$), being $p$ the number of processors (figure 7). Each thread process receives and stores the two blocks and performs the operations. The operations are initiated at the moment the first value of the second block is available.
The product between the two blocks corresponds to partial results of block C, which are stored in the processor. After obtaining the final results, they are sent to the Result module through a FIFO. The processors send the data to the FIFO one word at a time.

![Fig. 7. File Processor.h.](image)

<table>
<thead>
<tr>
<th>Block A</th>
<th>Block B</th>
<th>Transfers</th>
<th>Cycles</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4x1</td>
<td>1x2</td>
<td>802.816</td>
<td>802.818</td>
<td>55</td>
</tr>
<tr>
<td>4x1</td>
<td>1x4</td>
<td>540.672</td>
<td>540.678</td>
<td>45</td>
</tr>
<tr>
<td>4x1</td>
<td>1x8</td>
<td>409.600</td>
<td>528.486</td>
<td>32</td>
</tr>
<tr>
<td>4x1</td>
<td>1x16</td>
<td>344.064</td>
<td>528.582</td>
<td>32</td>
</tr>
<tr>
<td>8x1</td>
<td>1x2</td>
<td>671.744</td>
<td>671.750</td>
<td>50</td>
</tr>
<tr>
<td>8x1</td>
<td>1x4</td>
<td>409.600</td>
<td>528.490</td>
<td>31</td>
</tr>
<tr>
<td>8x1</td>
<td>1x8</td>
<td>278.528</td>
<td>528.586</td>
<td>31</td>
</tr>
<tr>
<td>8x1</td>
<td>1x16</td>
<td>212.992</td>
<td>528.778</td>
<td>28</td>
</tr>
<tr>
<td>16x1</td>
<td>1x4</td>
<td>344.064</td>
<td>528.594</td>
<td>31</td>
</tr>
<tr>
<td>16x1</td>
<td>1x8</td>
<td>212.992</td>
<td>528.786</td>
<td>28</td>
</tr>
<tr>
<td>16x1</td>
<td>1x16</td>
<td>147.456</td>
<td>529.170</td>
<td>26</td>
</tr>
</tbody>
</table>

**Table 1.** Matrix with size 128x128 and \( z = 1 \).

<table>
<thead>
<tr>
<th>Block A</th>
<th>Block B</th>
<th>Transfers</th>
<th>Cycles</th>
<th>Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4x1</td>
<td>1x2</td>
<td>6.356.992</td>
<td>6.356.994</td>
<td>9</td>
</tr>
<tr>
<td>4x1</td>
<td>1x4</td>
<td>4.259.840</td>
<td>4.259.846</td>
<td>6</td>
</tr>
<tr>
<td>4x1</td>
<td>1x8</td>
<td>3.211.264</td>
<td>4.210.790</td>
<td>4</td>
</tr>
<tr>
<td>4x1</td>
<td>1x16</td>
<td>2.686.976</td>
<td>4.210.886</td>
<td>4</td>
</tr>
<tr>
<td>8x1</td>
<td>1x2</td>
<td>5.308.416</td>
<td>5.308.422</td>
<td>7</td>
</tr>
<tr>
<td>8x1</td>
<td>1x4</td>
<td>3.211.264</td>
<td>4.210.794</td>
<td>4</td>
</tr>
<tr>
<td>8x1</td>
<td>1x8</td>
<td>2.162.688</td>
<td>4.210.890</td>
<td>4</td>
</tr>
<tr>
<td>16x1</td>
<td>1x4</td>
<td>2.686.976</td>
<td>4.210.898</td>
<td>4</td>
</tr>
<tr>
<td>16x1</td>
<td>1x16</td>
<td>1.638.400</td>
<td>4.211.090</td>
<td>4</td>
</tr>
<tr>
<td>16x1</td>
<td>1x16</td>
<td>1.114.112</td>
<td>4.211.474</td>
<td>3</td>
</tr>
</tbody>
</table>

**Table 2.** Matrix with size 256x256 and \( z = 1 \).

<table>
<thead>
<tr>
<th>Block A</th>
<th>Block B</th>
<th>Transfers</th>
<th>Cycles</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>4x1</td>
<td>1x4</td>
<td>269.484. 032</td>
<td>269.484. 038</td>
<td>Hours</td>
</tr>
<tr>
<td>4x1</td>
<td>1x8</td>
<td>202.375. 168</td>
<td>268.697. 702</td>
<td>Hours</td>
</tr>
<tr>
<td>8x1</td>
<td>1x4</td>
<td>202.375. 168</td>
<td>268.697. 706</td>
<td>Hours</td>
</tr>
<tr>
<td>8x1</td>
<td>1x8</td>
<td>135.266. 304</td>
<td>268.697. 802</td>
<td>Hours</td>
</tr>
</tbody>
</table>

**Table 3.** Matrix with size 1024x1024 and \( z = 1 \).

The number of cycles is obtained by considering that one cycle is required to write to the FIFO and that the multiplication-addition unit has a throughput of one result per cycle. The clock cycles are described in the processes and expressed by the function \( \text{wait()} \).

Observing tables 1, 2 and 3, we can conclude that the number of transfers and the number of clock cycles are consistent with the equations (2) and (3), respectively, as the difference between the simulated values and the theoretical approximation is less than 1%.

As expected, the number of transfer cycles required decreases with the size of the C blocks.

**5. Simulation and Results**

The presented model was implemented and simulated for matrices of different sizes and different blocks, considering four processors.

Tables 1, 2 and 3 show the results obtained with regard to the number of data transfers, the total number of clock cycles and the execution time, considering \( z = 1 \).
The minimal number of cycles is achieved when all the communications, except the initial overhead, can be totally overlapped with the computations. These are shown as best results and marked in bold.

The (to be selected) size of the C block should be sufficient for the computation cycles to dominate, that is, for the transfers to be able to fully overlap with the computations, but not larger. As long as there is full overlap, there is no need to further increase the size of the C blocks (and therefore the local memory of the processors). In fact and as shown, above the ideal values the total number of clock cycles slightly increases with \( x \) and \( y \). This is because the initial and final overheads are proportional to the sizes of the matrix blocks.

<table>
<thead>
<tr>
<th>Block A</th>
<th>Block B</th>
<th>Transfers</th>
<th>Cycles</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4x2</td>
<td>2x4</td>
<td>540.672</td>
<td>540.682</td>
<td>48</td>
</tr>
<tr>
<td>4x2</td>
<td>2x8</td>
<td>409.600</td>
<td>528.490</td>
<td>30</td>
</tr>
<tr>
<td>4x2</td>
<td>2x16</td>
<td>344.064</td>
<td>528.586</td>
<td>32</td>
</tr>
<tr>
<td>8x2</td>
<td>2x4</td>
<td>409.600</td>
<td>528.498</td>
<td>32</td>
</tr>
<tr>
<td>16x2</td>
<td>2x4</td>
<td>344.064</td>
<td>528.610</td>
<td>30</td>
</tr>
<tr>
<td>4x8</td>
<td>8x8</td>
<td>409.600</td>
<td>528.546</td>
<td>33</td>
</tr>
<tr>
<td>4x16</td>
<td>16x8</td>
<td>409.600</td>
<td>528.673</td>
<td>34</td>
</tr>
<tr>
<td>4x64</td>
<td>64x8</td>
<td>409.600</td>
<td>529.441</td>
<td>37</td>
</tr>
</tbody>
</table>

Table 4. Matrix with size 128x128.

<table>
<thead>
<tr>
<th>Block A</th>
<th>Block B</th>
<th>Transfers</th>
<th>Cycles</th>
<th>Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4x2</td>
<td>2x4</td>
<td>4,259,840</td>
<td>4,259,850</td>
<td>6</td>
</tr>
<tr>
<td>4x2</td>
<td>2x8</td>
<td>3,211,264</td>
<td>4,210,794</td>
<td>4</td>
</tr>
<tr>
<td>4x2</td>
<td>2x16</td>
<td>2,686,976</td>
<td>4,210,890</td>
<td>4</td>
</tr>
<tr>
<td>8x2</td>
<td>2x4</td>
<td>3,211,264</td>
<td>4,210,802</td>
<td>4</td>
</tr>
<tr>
<td>4x32</td>
<td>32x8</td>
<td>3,211,264</td>
<td>4,211,233</td>
<td>4</td>
</tr>
<tr>
<td>4x64</td>
<td>64x8</td>
<td>3,211,264</td>
<td>4,211,745</td>
<td>4</td>
</tr>
<tr>
<td>4x128</td>
<td>128x8</td>
<td>3,211,264</td>
<td>4,212,769</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 5. Matrix with size 256x256.

<table>
<thead>
<tr>
<th>Block A</th>
<th>Block B</th>
<th>Transfers</th>
<th>Cycles</th>
<th>Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4x16</td>
<td>16x8</td>
<td>202,375</td>
<td>268.69</td>
<td>Hours</td>
</tr>
<tr>
<td>4x128</td>
<td>128x8</td>
<td>202,375</td>
<td>268.69</td>
<td>Hours</td>
</tr>
<tr>
<td>4x256</td>
<td>256x8</td>
<td>202,375</td>
<td>268.70</td>
<td>Hours</td>
</tr>
</tbody>
</table>

Table 6. Matrix with size 1024x1024.

Tables 4, 5 and 6 show the results with regard to the number of data transfers and cycles and the execution time, considering different values of \( z \). These results confirm the theoretical conclusion, and equation (2), that the number of transfers does not change when the value \( z \) is increased. Again and as expected, the number of clock cycles slightly increases with \( z \), because the overheads depend on the size of the blocks used.

6. Conclusions

In this paper we have described an approach to simulate many-core architectures using SystemC. We started with a brief overview of the language, describing some of the important concepts needed to create a model of a system.

A parallel matrix multiplication algorithm to execute on a 2-dimensional multiprocessor array was presented and analyzed theoretically.

An architecture was developed and implemented in SystemC in order to model the multiprocessor system design.

We simulated the model to evaluate number of transfers and number of clock cycles required for the complete algorithm execution. The simulated results fully confirmed the theoretical analysis (the differences are less than 1%).

The proposed SystemC model is now being generalized to be able to simulate any number of processors, so that massively parallel architectures may be evaluated.

Acknowledgment

This work was supported by national funds through FCT, Fundação para a Ciência e Tecnologia, under projects PEst-OE/EEI/LA0021/2011 and PTDC/EEA-ELE/122098/2010.

References