

# Hierarchical Dependency Graphs: Abstraction and Methodology for Mapping Systolic Array Designs to Multicore Processors<sup>\*</sup>

Sudhir Vinjamuri and Viktor Prasanna

3740 McClintock Avenue EEB 200, Ming Hsieh Department of Electrical Engineering  
University of Southern California, California, USA 90089-2562  
Tel.: +1-213-740-1521, +1-213-740-4483; Fax: +1-213-740-4418  
sudhir.vinjamuri@usc.edu, prasanna@usc.edu

**Abstract.** Systolic array designs and dependency graphs are some of the most important class of algorithms in several scientific computing areas. In this paper, we first propose an abstraction based on the fundamental principles behind designing systolic arrays. Then, based on the abstraction, we propose a methodology to map a dependency graph to a generic multicore processor. Then we present two case studies: Convolution and Transitive Closure, on two state of the art multicore architectures: Intel Xeon and Cell multicore processors, illustrating the ideas in the paper. We achieved scalable results and higher performance compared to standard compiler optimizations and other recent implementations in the case studies. We comment on the performance of the algorithms by taking into consideration the architectural features of the two multicore platforms.

**Keywords:** parallel programming, multicore, systolic array designs, dependency graphs, high performance computing.

## 1 Introduction and Background

Signal and image processing algorithms, matrix and linear algebra operations, graph algorithms, molecular dynamics and geo-physics are some of the core scientific computing research areas ([1]). The 70s, 80s saw the upsurge of a revolutionary high performance computing technology - systolic array processors ([3]) necessitated by increasing demands of speed and performance in these areas. A lot of research work has been done to expose the parallelism and pipelining available in several important scientific computing applications to be exploited by systolic array processors ([1], [2], [4], [5]). For many of these algorithms, the

---

<sup>\*</sup> This research was partially supported by the NSF under grant number CNS-0613376. NSF equipment grant CNS-0454407 is gratefully acknowledged. The authors acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of Cell Broadband Engine resources that have contributed to this research.

hardware needed to be flexible and robust to be able to adapt to new problems and also variations in known algorithms. Many solutions were proposed to this problem such as configurable systolic array processing platforms, FPGAs and reconfigurable computing platforms ([8]).

Today's computing revolution is driven by massive on-chip parallelism ([9]). For the foreseeable future, high performance computing machines will almost certainly be equipped with nodes featuring multicore processors where each processor contains several full featured general purpose processing cores, private and shared caches. So the primary motivation of this work is to study, how "classical" algorithms can be "recycled" now that parallel computing has a renaissance with the advent of multicore computers. Also, the configurable platforms of systolic arrays pale out in comparison with multicore processors of comparable area and cost in terms of raw compute power and peak performance achievable due to high clock rate, chip density and economies of scale of multicore processors. Hence, it is highly desirable to extract parallelism and pipelining necessary for systolic array designs from multicore processors. If done intelligently, this will result in highly optimized performance since those properties are inherent to multicore architectures.

To the best of our knowledge, there is no known prior work to map dependency graphs or systolic arrays to the current generation of multicore processors. We believe this is the first attempt for studying this problem. The main challenge of coming up with a methodology to map any systolic array designs to a multicore processor requires deep understanding of systolic array design procedures, data partitioning, scheduling operations and data flow control. On a multicore processor this poses extra challenges where synchronization of the operations and data of the cores has to be controlled by the programmer. The remainder of this paper is organized as follows: Section 2 is the crux of this paper where we present the approach for this study, the abstraction of Hierarchical Dependency Graphs and mapping methodology to multicore processors. In Section 3, we present two case studies of two algorithms on two multicore architectures. We conclude the paper with a brief summary and avenues for future work in Section 4.

## 2 Hierarchical Dependency Graphs

In this Section, first we substantiate the approach adopted in this paper. Then in Section 2.2, we discuss the properties of dependency graphs, differentiate and define data flows. In Section 2.3, we describe the abstraction of hierarchical dependency graphs and their properties. In Section 2.4, we discuss the steps to generate a mapping for a generic dependency graph to a multicore processor.

### 2.1 Approach for This Study

Systolic array algorithms may or may not have a specific design methodology/steps ([6]). The methodology for designing systolic arrays is a description

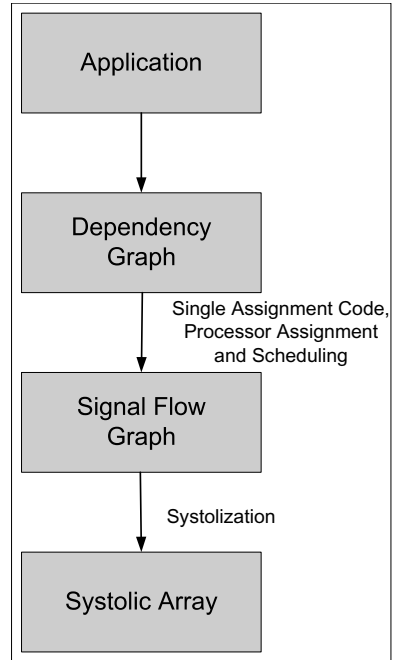
of a sequence of steps at best and not a concrete algorithm that takes an application and designs a systolic array for it. Figure 1 shows one of the widely used set of steps to design systolic arrays ([1]).

Dependency graphs are converted into systolic arrays by passing them through a sequence of steps one of which is single assignment code. The motivation of single assignment code is to avoid broadcasting in the VLSI design technology because it brings down the clock rate. But, in the case of today's multicore processors, while write conflict between cores to a memory location is a problem, broadcasting is not a problem currently because all that means is a value being read by all processors which, as will become evident from the case studies, is not a problem. Hence we consider the designs at the level of dependency graphs. But one ambiguity arises in the cases where the systolic array design is very similar or the same as the dependency graph. To resolve this issue, we consider systolic array designs at the level of dependency graphs itself in this paper.

## 2.2 Dependency Graphs

The theory of dependency graphs has been discussed in [10]. These were later used ([1]) in the design of systolic arrays. In this paper, to make our approach intuitive, instead of getting into intricate details of the definitions of dependency graphs, we use an example of a dependency graph to describe our ideas. As the properties used in the example are generic to dependency graphs and also through the case studies, it will be clear that our approach can be applied to any dependency graph. We chose the systolic array design for transitive closure([2], [4]) as the example. So we briefly describe the problem and its systolic array design below. Additional details can be found in [2].

Transitive closure is a fundamental problem in a wide variety of fields, most notably network routing and distributed computing. Suppose we have a directed graph  $G$  with  $N$  vertices and  $E$  edges. Transitive closure of the graph involves in computing for each vertex of the graph, the subset of vertices to which it is connected and the shortest distance between them (The 0-1 version of transitive closure only shows if the vertices are connected. We use the generic version which gives the shortest distances also). Given the graph, the adjacency matrix  $W$  is a 2 dimensional matrix with elements representing edge weights (eqn. 1).



**Fig. 1.** Steps for designing a systolic array

$$w_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \text{weight of edge connecting vertex } i \text{ to } j & \text{if } i \neq j \text{ and } (i, j) \in E \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E \end{cases} \quad (1)$$

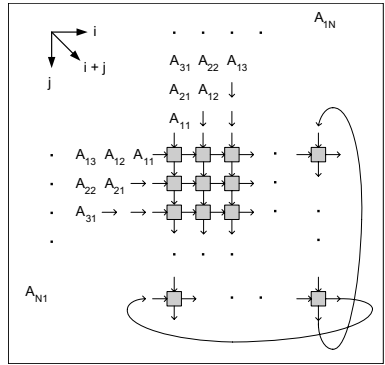
The dependency graph design for transitive closure is as follows:

1. Given a graph with  $N$  vertices in the adjacency matrix representation ( $A$ ), feed the matrix into an  $N \times N$  systolic array of processing elements (PEs) both row-wise from top and column-wise from left as shown in Figure 2.
2. At each PE  $(i, j)$ , update the local variable  $C_{(i,j)}$  by the following formula:

$$C_{(i,j)} = \min(C_{(i,j)}, A_{(i,k)} + A_{(k,j)}) \quad (2)$$

where  $A_{(i,k)}$  is the value received from the top and  $A_{(k,j)}$  is the value received from the left.

3. If  $i=k$ , pass the value  $C_{(i,j)}$  down, otherwise pass  $A_{(k,j)}$  down. If  $j=k$ , pass the value  $C_{(i,j)}$  to the right, otherwise pass  $A_{(i,k)}$  to the right.
4. Finally, when data elements reach the edge of the matrix, a loop around connection should be made such that  $A_{(i,N)}$  passes data to  $A_{(i,1)}$  and  $A_{(N,j)}$  passes data to  $A_{(1,j)}$  (see Figure 2).
5. The above computation results in the transitive closure of the input once all the input elements have been passed through the entire array exactly 3 times. We consider 1 of the 3 cycles of the operation for discussing the properties below.



**Fig. 2.** Systolic Array Implementation of Transitive Closure

**Properties of Dependency Graphs:** We need two properties of dependency graphs ([1], [10]). We will be using these to explain the properties of hierarchical dependency graphs in Section 2.3.

**Property of Parallelism:** This property states that, at a specific instant of time, a number of nodes can be processing data in parallel. There are many variations on how this parallelism is present depending on the dependency graph. For e.g. in Figure 2, nodes along the anti-diagonal can process data in parallel. Calling the top right corner node as  $(0, 0)$  and  $\hat{i}, \hat{j}$  axis as shown in the Figure, nodes  $(1, 0)$  and  $(0, 1)$  can process data in parallel. Similarly, nodes in each set  $[(2, 0), (1, 1), (0, 2)], [(3, 0), (2, 1), (1, 2), (0, 3)]$  etc. can process data in parallel.

**Property of Modularity, Regularity and Scaling:** The array consists of modular processing units with homogeneous interconnections. Moreover, the computing network may be extended indefinitely based on the problem size. For e.g. in Figure 2, the size of the dependency graph is the size of the adjacency matrix. Hence, for a problem of size of  $N$ , the dependency graph is of size  $N \times N$ .

A problem of size  $2N$  has a dependency graph of size  $2N \times 2N$ , which can be interpreted modularly as connecting four dependency graphs of a problem of size  $N$  (i.e. adjacency matrices of size  $N \times N$ ).

**Data Flow:** We differentiate the data flowing in the dependency graph into two types:

- Data that is updated at a node before being sent to the next node.
- Data that is sent as it is, without any modifications to the next node.

### Definitions

**Update Direction (UD):** We define UD as the direction in which data that is updated at a node, is sent to another node. In other words, this is the direction in which there is a dependency in data flow. The UD is represented by a unit vector along that direction. There can be more than one UD for a dependency graph. For instance, transitive closure (Figure 2) has two UDs: along  $\hat{i}$  and  $\hat{j}$  directions (i.e. data updated at a node is sent to the neighboring nodes along  $\hat{i}$  and  $\hat{j}$  directions) (These UDs will be used to cut the dependency graphs during the mapping process (discussed in the Section 2.4)).

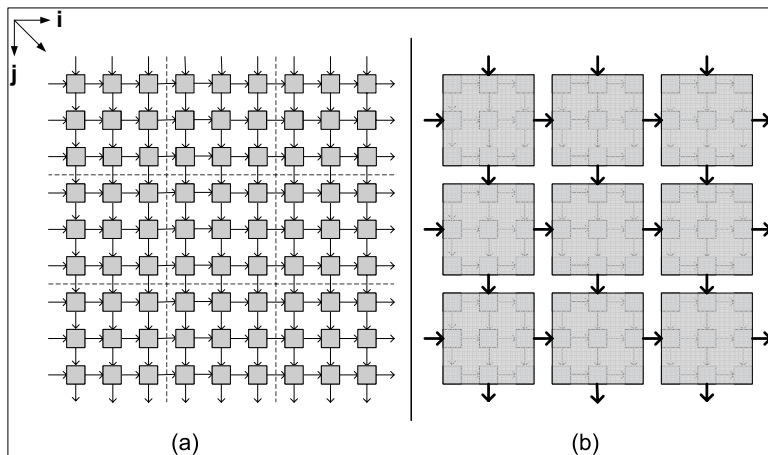
**Unified Update Direction (UUD):** In the case where a dependency graph has multiple UDs, we define UUD as the unit vector along the average of the UDs. Every dependency graph whether it has one UD or multiple UDs, has only a single UUD. In the case where the dependency graph has only a single UD, that itself will become the UUD. So the UUD for transitive closure is along  $(\hat{i} + \hat{j})/\sqrt{2}$  (The UUD will be used in scheduling of the hierarchical dependency graph during the mapping process (discussed in the Section 2.4)).

$\Theta$  (**Theta**): The maximum angle between any two UDs in a dependency graph.

### 2.3 Abstraction

We explain the abstraction of hierarchical dependency graphs with an example. Consider the dependency graph of size  $9 \times 9$ , which is similar to the transitive closure design, in Figure 3(a). Consider the time instant at which all the nodes in the graph are busy processing data. At the beginning of each cycle, each node gets inputs from top and left directions, processes data and outputs updated data to the right and bottom directions. Similarly for every cycle, there is input from top and left directions and output to the right and bottom directions flowing for the overall dependency graph. Similar to transitive closure design, there are two UDs for this dependency graph along the  $\hat{i}$  and  $\hat{j}$  directions and the UUD is along  $(\hat{i} + \hat{j})/\sqrt{2}$ .

We have re-drawn Figure 3(a) in Figure 3(b) with the following modifications. Cut the dependency graph along six lines parallel to the UDs, three in each direction as shown as dashed lines. Represent each partition with a shaded



**Fig. 3.** Hierarchical Systolic Array

big node (name it macro node), encompassing the 3 x 3 matrix of nodes (name them micro nodes) inside it. Represent the data flow into and out of each macro node with a bold arrow. For e.g., for the top left macro node, the arrow from above represents input coming to all the top 3 micro nodes inside it, the arrow from left represents input coming into all 3 micro nodes on the left. Similarly, the right and bottom arrows represent output from all three right and bottom micro nodes respectively, inside it. This is true for all macro nodes in Figure 3(b).

**Formalizing the Abstraction:** The idea of bisecting a dependency graph into parts and representing each partition by a macro node is called **Hierarchical Dependency Graphs** (HDGs). In this paper, we discuss about HDGs of one level of hierarchy: a **macro dependency graph** consisting of a set of macro nodes, which is nothing but the complete dependency graph being studied from the point of view of the macro nodes. Each macro node has a **micro dependency graph** (one macro node with micro nodes) inside it. While this abstraction some similarity to tiling ([4]), this idea can be more easily extended to multiple levels of hierarchies, for new generation high performance computing and supercomputing systems with various levels of parallelism and compute power organization.

## 2.4 Mapping Methodology

In this Section, we first explain the mapping technique and provide an argument for its viability. In both these instances, we will be discussing in terms of the number of cycles of operation of macro nodes. So we first characterize this idea.

**Characterization of  $c$  cycles of a macro node:** A dependency graph operates in cycles. This means, in each cycle, a node gets input at the beginning of a cycle, processes the data and outputs data at the end of the cycle. We refer to  $c$  cycles of operation of a macro node as, all micro nodes inside the macro node execute  $c$  cycles of operation each, taking care of the dependencies between the macro node and the remaining part of the macro dependency graph and also, dependency between the micro nodes themselves inside the macro node.

**Steps for mapping a Dependency Graph to a Multicore Processor:**

The following are the steps for generating a mapping for a dependency graph on to a multicore processor:

- Cut the dependency graph along the UDs and represent each partition as a macro node.
- Schedule the macro nodes along the UUD taking care of the dependencies between macro nodes, by assigning  $c$  cycles of operation of a macro node to each core.
- It is possible to schedule in parallel, all macro nodes perpendicular to the UUD. This should be done step by step, along the UUD.

By stating properties of HDGs below, we will show that there is enough parallelism between the macro nodes that many macro nodes can be scheduled in parallel to many cores which can operate independently. Also, the above methodology will become clearer by examining these steps in the case studies in Section 3. There is also one more issue: the value of  $c$ . This will also be discussed after the properties of HDGs.

**Viability of the mapping technique:** We describe properties of the Hierarchical Dependency Graphs to show the viability of our mapping technique.

**Property of Parallelism between Macro Nodes:** Extending the property of parallelism in Section 2.2, at a single instant of time, several macro blocks can be processing data in parallel.

**Basis for the property:** We know that the conception of dependency graphs and systolic array designs is to extract the parallelism in the algorithm. At a single instant of time, many nodes in a DG can be processing data in parallel and passing data between each other at the end of each cycle (property of parallelism in Section 2.2). The above property is merely extending that parallelism from micro nodes to the level of macro nodes.

**Property of Independent Operation of a Macro node:** There exists at least one scheduling order by which, each of the macro blocks can be processed independent of the other for  $c$  cycles of operation of the macro node. The value of  $c$  is discussed below.

**Basis for the property:** This property is extension of the property of modularity, regularity and scaling in Section 2.2.

**Value of  $c$ :** The value of  $c$  is of important concern for us since, it decides the number of cycles for which a macro node can be processed independently by a core. This is directly related to taking care of the dependencies between macro nodes and hence automatically parallelizing the complete dependency graph on to the multicore processor. There can be two variations in the value that  $c$  can take based on the  $\Theta$  of a dependency graph.

- $0^\circ \leq \Theta \leq 90^\circ$  : We show the two boundary cases in the two case studies, and prove that  $c$  can take the total number of cycles of the dependency graph in this scenario. If there is only one UD for the dependency graph, it falls into the category of  $\Theta = 0^\circ$ . This being the case of convolution, transitive closure has  $\Theta = 90^\circ$ .
- $90^\circ < \Theta \leq 180^\circ$  : The value  $c$  can take varies depending on the DG design and how the dependencies between nodes and macro blocks are arranged. Also, most common DG designs fall into the previous category. In this paper, we do not consider this case and plan to study this in future work.

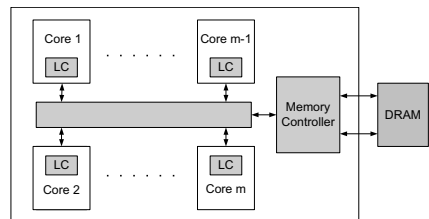
The two cases  $180^\circ < \Theta \leq 270^\circ$  and  $270^\circ < \Theta \leq 360^\circ$  can be interpreted as  $90^\circ < \Theta \leq 180^\circ$  and  $0^\circ < \Theta \leq 90^\circ$  respectively.

### 3 Case Studies

We present case studies of two algorithms on two architectures and we show scalable results in all four cases. First, we present a simple generic model of a multicore processor. We use this model to explain the mapping of the algorithm to a generic multicore processor, hence providing support to our claim that our methodology of mapping dependency graphs is applicable to multicore processors in general. Then we give a brief description of the two multicore platforms and the details of their architectures. In Sections 3.2 and 3.3, we present the mapping of transitive closure and convolution to multicore processors using the generic model as explained above. In Section 3.4, we discuss the experimental results for the two algorithms on the two platforms.

#### 3.1 Architecture Summaries

The multicore processor model is shown in Figure 4. The chip has  $m$  cores, each core having a local cache (LC). These local cache access main memory via a memory controller. We give a brief description of the architectures below and explain more details wherever necessary later in the paper (Section 3.4).



**Fig. 4.** Generic model of a multicore processor

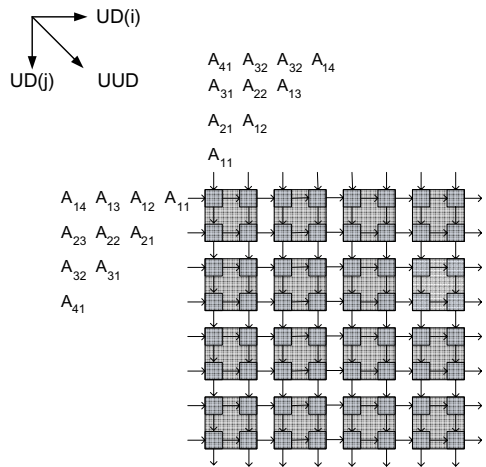
**Intel Quad Core Processor:** One of our platforms is a state-of-the-art homogeneous multicore processor system: Intel Xeon quadcore system. It contains two Intel Xeon x86\_64 E5335 processors, each having four cores. The processors run at 2.00 GHz with 4 MB cache and 16 GB memory. The operating system is Red Hat Enterprise Linux WS release 4 (Nahant Update 7). We installed GCC version 4.1.2 compiler and Intel C/C++ Compiler (ICC) version 10.0 with streaming SIMD extensions 3 (SSE 3), also known as Prescott New Instructions (PNI).

**Cell Broadband Engine:** The Cell BE processor ([9]) is one of the first heterogeneous multicore processors that has given the programmer explicit control of memory management and low level communication primitives between the cores on the chip. It consists of a traditional microprocessor (PPE) that controls 8 SIMD co-processing units (SPEs), a high speed memory controller and a high bandwidth bus interface (EIB), all integrated on a single chip. Each SPE consists of a synergistic processor unit (SPU) and a memory flow controller (MFC) and 256 KB of local store (LS) where the MFC is a software controlled unit serving the memory management between the LS and main memory. This utility of software controlled LS allows more efficient use of memory bandwidth than is possible with standard prefetch schemes on conventional cache hierarchies, but also poses extra challenges for the programmer. At 3.2 GHz, the single precision peak performance of the Cell processor is 204.8GFLOPS with Fused Multiply Add (two ops) primitive and 102.4GFLOPS without it (single op).

### 3.2 Case 1: Transitive Closure

**Algorithm Description:** The algorithm and the dependency graph design for transitive closure are described in Section 2.2. So, we directly describe the mapping. We consider here sample sizes of the problem and the cut sets for ease of illustration. The actual numbers can be varied depending on the real world problem sizes. Examples of these experiments are given in the Section 3.4.

Consider transitive closure problem of size  $N = 8$ . So we have an adjacency matrix of size  $8 \times 8$ . Figure 5 shows this  $8 \times 8$  adjacency matrix. As described previously, there are two UD's along  $\hat{i}$  and  $\hat{j}$  and the UUD is along  $(\hat{i}+\hat{j})/\sqrt{2}$ . Also, as previously explained, we refer to one cycle of operation of a node as, the node taking input from top and left directions, processes it and send output to the left and bottom



**Fig. 5.** Hierarchical Dependency Graph for Transitive Closure

directions at the end of the cycle. The total number of cycles of operation of each node in the graph is equal to  $N = 8$ .

**Mapping:** We follow the steps of mapping as described in Section 2.4. First, cut the dependency graph with 8 cut lines, 4 in each direction parallel to the UDs. Figure 5 shows the 2 x 2 macro blocks shaded, after cutting the graph. Observe that this Figure is very similar to Figure 3(b) where the difference is that the macro blocks are of size 3 x 3, and the problem size is  $N = 9$  in that Figure. Next, schedule the macro nodes along the UUD and execute each macro node for " $N$  cycles of the macro node" (Section 2.4). Therefore,  $c$  is equal to the complete cycles of the macro node. Also, schedule macro nodes perpendicular to the UUD in parallel to multiple cores, which process those macro nodes independently.

Numbering the top left macro node and micro node  $(0, 0)$  and  $\hat{i}$  and  $\hat{j}$  axis in Figure 5, the above steps of mapping have the following interpretation:

1. A single core processes macro node  $(0, 0)$  ( here processes means update the macro node for all  $N$  cycles, i.e., micro node  $(0, 0)$  is processed for 8 cycles, then micro nodes  $(1, 0)$  and  $(0, 1)$  are processed for 8 cycles, then micro node  $(1, 1)$  is processed for 8 cycles).
2. Process macro nodes  $(0, 1)$  and  $(1, 0)$  in parallel by two cores independently.
3. Process macro nodes  $(0, 2)$ ,  $(1, 1)$  and  $(2, 0)$  in parallel by three cores independently.
4. Process macro nodes  $(0, 3)$ ,  $(1, 2)$ ,  $(2, 1)$  and  $(3, 0)$  in parallel by four cores independently.
5. Process macro nodes  $(1, 3)$ ,  $(2, 2)$  and  $(3, 1)$  in parallel by three cores independently.
6. and so on ...

There is a synchronization point after each step, which can be removed once the number of macro nodes processed in parallel in a step exceeds the number of cores,  $m$  on the chip.

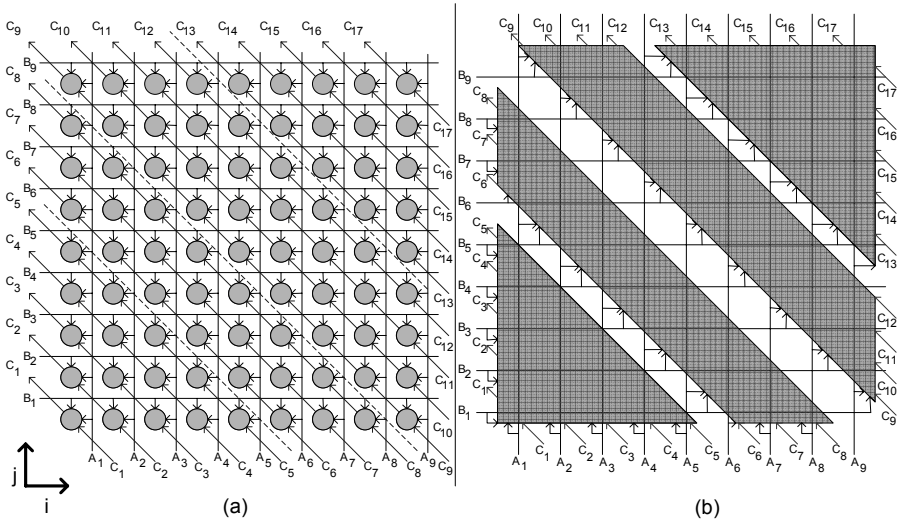
The above mapping and scheduling takes care of the dependencies between all nodes of the dependency graph in Figure 2. Also, observe that the HDG satisfies the properties described in Section 2.2. The first property states that there will be parallelism between the macro nodes. This is true since, with large problem sizes, the number of macro blocks that can be scheduled in parallel increases, and all  $m$  cores will be busy. The second property states that, there exists a scheduling order by which, the macro blocks can be processed independent of each other for  $c( = N)$  cycles of operation of the macro node. This is also true since, all the macro blocks along the perpendicular to the UUD have no dependencies and can be processed for  $c( = N)$  cycles of operation. Also by respecting the dependencies between macro blocks along the UUD by a synchronization point wherever necessary, the macro blocks along the UUD are also scheduled for  $N$  cycles of operation.

### 3.3 Case 2: Convolution

**Algorithm Description:** Convolution is one of the most important kernels in scientific computing. It is of fundamental importance [11] in signal processing, image processing, communication systems, computer vision and pattern recognition algorithms. Variations of convolution [5] are also used to solve integer multiplication and polynomial multiplication problems. There has been a lot of interest for high performance convolution computation recently [7], [8]. We describe the mapping of 1D convolution dependency graph to a multicore processor using our technique. A good feature of the 1D dependency graph design is that, 2D and 3D designs are exact symmetric and modular extensions of the 1D design. So our mapping technique is directly applicable to 2D and 3D convolution also.

Consider two digital signals  $A$  and  $B$  of dimension  $1 \times N$ . The convolution  $C$  of  $A$  and  $B$ , represented by  $C = A \otimes B$ , is a  $1 \times 2N - 1$  given by equation 3,

$$C(i) = \sum A(i).B(N - i) \tag{3}$$



**Fig. 6.** Dependency Graph (DG) and Hierarchical DG Abstraction for Convolution

Consider a problem of size  $N = 9$ . The dependency graph for computing the convolution of  $A$  and  $B$  is shown in Figure 6(a). Unlike transitive closure where all data flowing in the dependency graph is being updated, here not all data is updated. So, as described in Section 2.2, we differentiate the two types of data flowing. So in the Figure, we show the data that is not being updated (signals  $A$  and  $B$ ) with black coloured lines and data being updated (signal  $C$ ) with blue coloured lines. With  $\hat{i}$  and  $\hat{j}$  axis as shown, there is a single UD along  $(-\hat{i} + \hat{j})/\sqrt{2}$ . Since there is only one UD, that itself will become the UUD.

**Mapping:** Using the steps of mapping as described in Section 2.4, cut the dependency graph along the UD. Figure 6(b) shows the dependency graph after this operation where the four partitions resulting from three cut lines (shown in dashed lines in Figure 6(a)): after  $C_5$ ,  $C_8$  and  $C_{12}$ , are shown in shaded blocks. Each partition, a macro block, is represented by a shaded region with its inputs and outputs. A single line can be drawn perpendicular to the UUD which will pass through all macro nodes, which means there is no dependency between them and all the four macro nodes can be scheduled in parallel to four cores independently. Hence, when there are  $m$  cores, the dependency graph is partitioned into  $m$  macro blocks, with equal computational load, each of which will be processed by each core independently. Similar to the transitive closure case, observe that the HDG for convolution satisfies the two properties in Section 2.2.

### 3.4 Experimental Results and Discussion

We provide experimental results below proving that the mapping techniques give scalable results on multicore processors. We comment about the performance (single precision) and optimizations in Section 3.4.

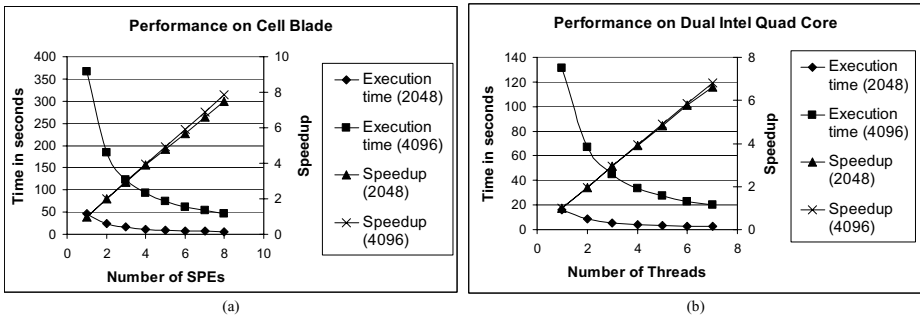


Fig. 7. Experimental results for Transitive Closure

**Transitive Closure:** We considered problems of sizes  $N = 2048$  and  $4096$ . Each macro block is of size  $64 \times 64$  by using 32 and 64 cut lines in each direction parallel to the UDs respectively for the two problem sizes. Hence, there are a total of  $32 \times 32$  and  $64 \times 64$  macro blocks for problem sizes 2048 and 4096 respectively. Figure 7(a) shows the scalable results on Cell blade. The performance achieved is 4.42GFLOPS. Figure 7(b) shows the scalable results on the dual Intel quad core platform. We achieved 10.54GFLOPS on this platform.

**Convolution:** We considered problem of size  $N = 32K$ . Figure 8(a) shows performance results on Cell Blade. We achieved 0.801GFLOPS on this platform. Figure 8(b) shows the results on the Intel quad core platform. The performance achieved is 17.5GFLOPS. Both the results are scalable as can be observed from the Speedup plots.

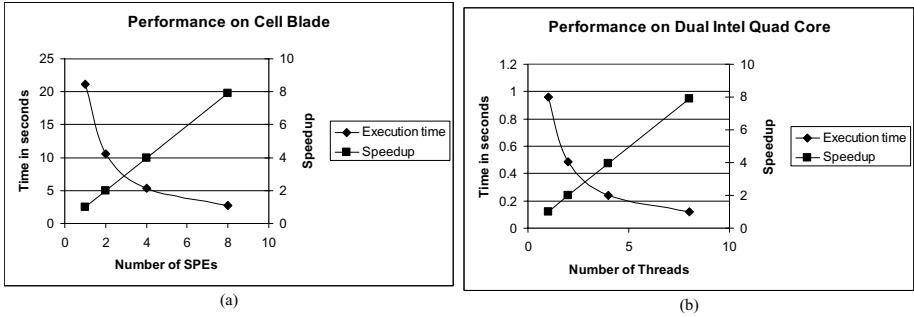


Fig. 8. Experimental results for Convolution

**Discussion:** Before commenting about the performance itself, we need to describe the differences and factors in architecture and programming models for the Cell and Intel multicore processors that effect the performance. First, in the Cell processor, as mentioned in Section 3.1, the complete memory management between the main memory and local stores of SPEs is in the hands of the programmer. While this increases the complexity of programming, it gives an advantage for achieving performance in cases where the operating system and cache coherency protocols cause heavy overhead and limit the performance. Whereas, in the Intel quad core, memory management is completely handled by the operating system and compiler. This difference also allows us to do a baseline implementation comparison (Figure 10) with our techniques on the Intel processor but not on the Cell (here baseline implementation means the algorithm implementation without our parallelization techniques and only with compiler optimizations). This is because, unlike the Intel processor where the compiler can execute a piece of code, on the Cell processor, we need to parallelize and take care of memory management right for the most basic implementation.

Second, the SPEs in the Cell processor are highly specialized for SIMD processing units but with pre-condition that the data should be aligned in memory. SIMDization is not possible with mis-aligned data and which means operations on mis-aligned data ([9]) should be loaded in a single preferred slot of a vector register, the data is processed and written back to memory from the preferred slot. This causes a heavy overhead decreasing the performance by orders of magnitude ([9]). SIMDization is more easy on the Intel quad core, in fact the Intel ICC compiler is good enough to auto-simdize operations quite efficiently.

Lastly, Cell SPEs have a bad branch prediction units compared to the Intel’s cores. This is because SPE’s architecture is optimized for streaming data ([9]).

**Performance of Transitive Closure:** We achieved a high performance of 10.54GFLOPS on the Intel Quad core processor. The compiler auto-simdized the code. We manually SIMDized the code on the Cell processor by grouping operations on micro node into a vector operation. But at the end of  $N$  cycles, one value (the lowest) out of the four should be assigned to the micro node. There is

no vector primitive that does this on the Cell leading to a non-SIMD operation and also a branch prediction operation. This lead to the drop in performance and we achieved 4.42GFLOPS. With an improved branch prediction unit and more hardware to take care of non-SIMD operations in future versions of Cell ([9]), there is a very promising possibility to achieve peak performance of the algorithm on the Cell.

**Performance of Convolution:** We achieved 17.5GFLOPS on the Intel platform, whereas we achieved only 0.801GFLOPS on the Cell. This is because, the convolution algorithm necessitates operations on non-aligned data in every clock cycle. As these operations are not SIMDized, they lead to heavy loss in performance on the Cell processor. As mentioned above, this is one aspect that has to be addressed in future versions of the Cell processor to improve its potential to wide range of algorithms. Also, our performance of 17.5GFLOPS on the Intel platform is higher than other recent research work for high performance computation of convolution. In different contexts of the same convolution operation, researchers have achieved 2.16GFLOPS per node ([7]) on a 4-rack Blue Gene system (4096 nodes leading to overall 7TFLOPS) and 3.16GFLOPS ([8]) on an FPGA platform.

**Comparing with Compiler Optimizations:**

To measure the impact of our parallelization with the compiler optimizations, we ran a baseline implementation for convolution which involved straight coding of the algorithm with compiler optimizations (-O2, -O3, -O4, -msse3 etc.) both with GCC and Intel’s ICC compiler. The pseudo codes for the baseline implementation and our parallelized version are shown in Figure 9. Figure 10 shows the results from this experiment.

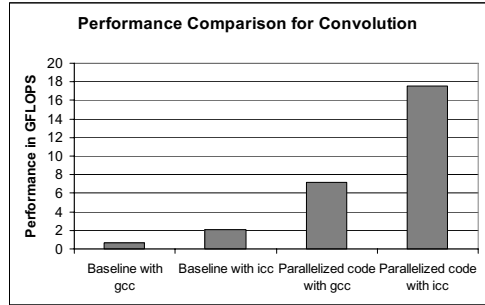
<pre> main {     float A[N], B[N], C[2*N-1]     // Values initialized for vectors A and B     // C vector initialized to 0      begin(measure time)     for(i = 0; i &lt; N; i++)     {         for(j = 0; j &lt; (i+1); j++)         {             C[i] = C[i] + A[j]*B[i - j];         }     }     for(i = N; i &lt; (2*N-1); i++)     {         for(j = (i-(N-1)); j &lt; N; j++)         {             C[i] = C[i] + A[j]*B[i-j];         }     }     end(measure time) }                 </pre>	<pre> float A[N], B[N], C[2*N-1]; // Globally accessible to main and all threads  Thread 1's function { Does work of macro block 1 }  Thread 2's function { Does work of macro block 2 }  ...  Thread m's function { Does work of macro block m }  main {     // Values initialized for A and B     // Vector C initialized to 0      begin(measure time)     // Create m-pthreads and call their     // respective functions.     // m can take values between 1 to 8     // (cores on the chip)     // Wait for threads to join     end(measure time) }                 </pre>
(a) Base line code	(b) Parallelized code

**Fig. 9.** Pseudo code for convolution

The best possible performance achieved by a compiler is 2.2GFLOPS (Baseline for ICC) whereas with our parallelization, we achieved 17.5GFLOPS. This shows that our mapping technique has given the compiler more opportunities to parallelize and optimize the dependency graph computations on the multicore processor.

## 4 Conclusion

We summarize the contributions made by this paper. Starting from a seminal problem of mapping systolic arrays to multicore processors, we made the observation that mapping of dependency graphs is more fundamental and should be studied rather than systolic arrays. We defined the abstraction of Hierarchical Dependency Graphs, using which we proposed a mapping methodology to map and parallelize a dependency graph to a multicore processor. We presented two case studies and achieved scalable results and good performance illustrating our methodology. In future, we plan to conduct more case studies of mapping dependency graphs to multicore processors and also possibly integrate these techniques into compilers.



**Fig. 10.** Comparison with Compiler Optimizations

## References

1. Kung, S.Y.: VLSI Array Processors. In: Kailath, T. (ed.) Prentice-Hall, Englewood Cliffs (1988)
2. Ullman, J.D.: Computational aspects of VLSI. Computer Science Press (1983)
3. Kung, H.T., Leiserson, C.E.: Systolic arrays (for VLSI). In: Sparse Matrix Symposium, pp. 256–282. SIAM, Philadelphia (1978)
4. Penner, M., Prasanna, V.K.: Cache Friendly Implementations of Transitive Closure. In: Proc. of PACT (2001)
5. Leighton, F.T.: Introduction to Parallel Algorithms and Architectures: Arrays, Trees and Hypercubes. Morgan Kaufmann, San Francisco (1992)
6. Rao, S.K., Kailath, T.: Regular Iterative Algorithms and their Implementation on Processor Arrays. Proc. of the IEEE 76, 259–269 (1988)
7. Nukada, A., Hourai, Y., Nishada, A., Akiyama, Y.: High Performance 3D Convolution for Protein Docking on IBM Blue Gene. In: Stojmenovic, I., Thulasiram, R.K., Yang, L.T., Jia, W., Guo, M., de Mello, R.F. (eds.) ISPA 2007. LNCS, vol. 4742, pp. 958–969. Springer, Heidelberg (2007)
8. Huitzil, C.T., Estrada, M.A.: Real-time image processing with a compact FPGA-based systolic architecture. Journal of Real-Time Imaging (10) 177–187 (2004)
9. Arevalo, A., Matinate, R.M., Pandlan, M., Peri, E., Ruby, K., Thomas, F., Almond, C.: Prog. the Cell Broadband Engine: Examples and Best Practises, IBM Redbooks
10. Karp, R.M., Miller, R.E., Winograd, S.: The Organization of Computations for Uniform Recurrence Equations. Jour. of ACM 14(3), 563–590 (1967)
11. Oppenheim, A.V., Schafer, R.W., Buck, J.R.: Discrete-Time Signal Processing. Prentice Hall Signal Processing Series (2004)