Transitive Closure on the Cell Broadband Engine: A study on Self-Scheduling in a Multicore Processor [*]

Sudhir Vinjamuri
Department of Electrical Engineering
University of Southern California
3740 McClintock Avenue EEB-244
Los Angeles USA 90007
sudhir.vinjamuri@usc.edu

Viktor K. Prasanna
Department of Electrical Engineering
University of Southern California
3740 McClintock Avenue EEB-200C
Los Angeles USA 90007
prasanna@ganges.usc.edu

Abstract

In this paper, we present a mapping methodology and optimizations for solving transitive closure on the Cell multicore processor. Using our approach, it is possible to achieve near peak performance for transitive closure on the Cell processor. We first parallelize the Standard Floyd Warshall algorithm and show through analysis and experimental results that data communication is a bottleneck for performance and scalability. We parallelize a cache optimized version of Floyd Warshall algorithm to remove the memory bottleneck. As is the case with several scientific computing and industrial applications on a multicore processor, synchronization and scheduling of the cores plays a crucial role in determining the performance of this algorithm. We define a self-scheduling mechanism for the cores of a multicore processor and design a self-scheduler for Blocked Floyd Warshall algorithm on the Cell multicore processor to remove the scheduling bottleneck. We also present optimizations in scheduling order to remove synchronization points. Our implementations achieved up to 78GFLOPS.

1 Introduction

The Cell BE processor ([1]) is one of the first heterogeneous multicore processor that has given the programmer explicit control of memory management, low level communication primitives between the cores on the chip and other utilities. 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 256KB 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 3.2GHz, 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).

Linear algebra kernels ([15], [18], [13]), matrix operations ([14], [17], [16]) are one of the most important problems in scientific computing research. There is considerable interest in using the Cell processor to push the frontiers of multicore and heterogeneous computing in these areas. There has been work on extracting performance for problems such as solving linear equations ([6]), FFT ([7]), graph algorithms ([5]) and scientific computing kernels ([8]). But to the best of our knowledge, there has been no work on solving transitive closure (the All-Pairs shortest path problem) on the Cell processor. Transitive closure is a fundamental problem in a wide variety of fields, most notably network routing and distributed computing. It is also used in construction of parsing automata in compiler construction ([10], [11]). Efficient transitive closure computation has been recognized as a significant sub-problem in evaluating recursive database queries ([12]). We use the Floyd Warshall algorithm to solve transitive closure.

Floyd Warshall algorithm, a dynamic programming approach, involves updating all elements of an NxN matrix at each step for N steps, where N is the number of vertices in the input graph. The operations involved are comparison and add which makes the peak performance achievable to be 102.4GFLOPS on the Cell processor. We are also faced with dependencies that require us to update the entire NxN

[*]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.

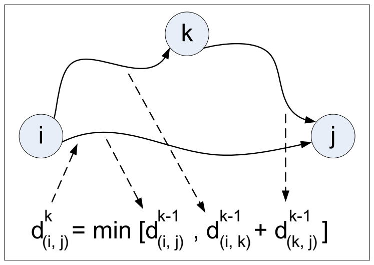

k
i
j
$d^{k}_{(i,j)} = \min [d^{k-1}_{(i,j)}, d^{k-1}_{(i,k)} + d^{k-1}_{(k,j)}]$

Figure 1. Floyd Warshall algorithm

```
1. n = [W]
2. $D^0$ = [W]
3. for (k = 1 to n)
4.     for (i = 1 to n)
5.         for(j = 1 to n)
6.             $d^k(i, j) = \min(d^{k-1}(i, j), d^{k-1}(i, k) + d^{k-1}(k, j))$
7. return $D^n$
```

Figure 2. Floyd Warshall algorithm

array at a single iteration before moving to the next. These dependencies mean that although computational complexity of the Floyd Warshall algorithm is $O(N^3)$, equivalent to matrix multiply, often transitive closure displays much longer times on conventional processors ([2]). On the Cell architecture, where the problem of memory coherency has to be explicitly managed by the programmer, this poses extra challenges.

The main contributions of this paper are as follows: (a) analysis for both performance and scalability of an algorithm on a multicore processor and show why ignoring one of them can be misleading (Sections 3.2 and 6), (b) define the policy of self-scheduling of the cores of a multicore processor to solve the problem of scheduling, a common and important performance bottleneck on a multicore processor, leading to breakdown of scalability of an algorithm (Section 5.2) (c) methodology to achieve near peak performance for transitive closure on the Cell processor (Sections 4.2, 5.2 and 5.3).

The remainder of the paper is organized as follows: In Section 2, we compute a theoretical lower bound on the execution time of Floyd Warshall algorithm on the Cell processor based only on its computations. However, transitive closure and Floyd Warshall algorithm fall into the categories of graph traversal and dynamic programming kernels. A generic study by IBM researchers ([9]) has shown both these problems to have data communication as a bottleneck for performance. In Section 3, we show that this is the case with Standard Floyd Warshall algorithm ([4]) by parallelizing it on Cell. We solve the problem of memory access in Section 4 by parallelizing a cache optimized Blocked Floyd Warshall algorithm for a single core processor from recent literature ([3], [2]). For this algorithm, we also propose a new scheduling methodology to reduce the number of synchronization points. Since both Standard Floyd Warshall algorithm and Blocked Floyd Warshall algorithm have same computations, we analyze the computation time in Section 2. However, we analyze the data communication, scheduling and synchronization times of the two algorithms in their individual Sections (Sections 3 and 4). For similar reasons, when we use the term Floyd Warshall algorithm we refer to the computations which are same for both the Standard algorithm and Blocked algorithm.

Though the algorithmic optimization solves the memory access problem, it leads to synchronization and scheduling problems. In Section 5, we firstly use a hardware primitive for synchronization giving a minor improvement and finally design a self-scheduling mechanism for the SPEs to eliminate the scheduling problem. We also perform code level optimizations to achieve 78GFLOPS for the Floyd Warshall algorithm on the Cell processor. In Section 6 we discuss how the ideas in this paper can be extended to a cluster of multicore processor systems and conclude the paper.

2 Transitive Closure and Floyd Warshall Algorithm

2.1 Algorithm description

Suppose we have a directed graph G with N vertices labeled 1 to N and E edges. Transitive Closure of the graph computes for every vertex in the graph, the subset of vertices to which it is connected. Here we define connected as, "Vertex A is connected to vertex B if there exits a path in the graph to go from vertex A to vertex B". Since it is a directed graph, vertex A may be connected vertex B but the converse may not be true. To solve transitive closure, we use Floyd Warshall algorithm, a dynamic programming algorithm, which calculates the shortest distances between nodes given the adjacency matrix W representing the edge weights (eqn 1) of an N-vertex graph ([4]).

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

The algorithm computes a series of N, NxN matrices where D^k is the k^{th} matrix and is defined as follows: $D^k(i,j)$ is the shortest path from vertex i to vertex j composed of a subset of vertices labeled 1 to k. The matrix D^0 is the original adjacency matrix W. We can think of the algorithm composed of N steps. At each k^{th} step, we compute D^k using the data from D^{k-1} in the manner shown in Figure 1. The steps of the algorithm are shown in Figure 2.

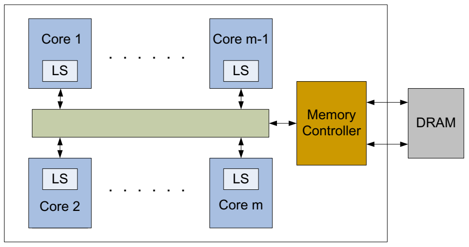


Figure 3. Model of a Multicore processor

2.2 Theoretical bound on performance

We compute a theoretical lower bound on execution time for Floyd Warshall on Cell by considering only the compute work in the algorithm. This bound holds even when all data communication, scheduling and synchronization are hidden. We use an architecture model (see Figure 3) for a multicore processor with parameters as defined below:

- Number of cores (SPEs) = m.
- Speedup due to SIMD/vectorization = s.
- Bandwidth between DRAM and cores = bw.
- Clock Speed = clk.

The algorithm characteristics we use in the analysis are:

- Problem Size = N.
- Block Size = BxB.

We have a 2D array with NxN elements where all elements of this array have to be updated for N iterations. To start with, assume each update takes a unit time, equation 2 gives the order of the computation time.

$$Total\ computation\ time\ = \Theta(N^3/m). \qquad (2)$$

The exact time for computations is given by the equation 3, where c is the number of cycles/operations of line 6 in Figure 1

$$Total\ execution\ time = (N^3 * c)/(clk * s * m). \qquad (3)$$

We consider matrix sizes that are large enough that they don't fit into the local store but can fit into the main memory of the Cell processor. To get an idea of the time estimate we plug in actual values for a single problem size: N = 2048, clk = 3.2*e9, s = 4 and m = 1 to 8.

The operation in line 6 when vectorized takes 3 cycles when completely and efficiently pipelined (there are 3 operations involved in line 6 - addition, comparison, selection). Therefore with c = 3, time taken for the case m = 8, is 0.2516 seconds. The theoretical bounds for m varying from 1 to 8 is given in Figure 4.

In the remaining part of the paper, whenever we do analysis of performance, we clearly break up the execution time into computation, data communication, scheduling and synchronization time to show the affect each of them have on the overall performance.

Difference between data communication and scheduling time: By data communication time we mean the time taken for copying of blocks of matrix between the local stores of the SPEs and main memory. Scheduling time is the time taken for passing the information of the order in which operations are to be executed. Also, in later sections, we discuss results for problem size N =2048. We have also tested with different problem sizes but due to space constraints we show our methodologies for the case N = 2048 only.

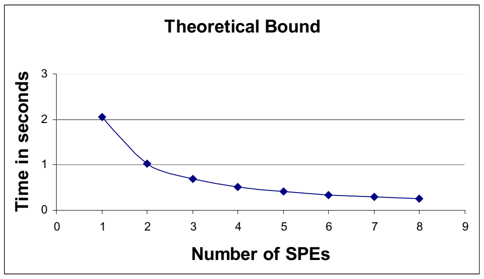


Figure 4. Theoretical bound for N = 2048

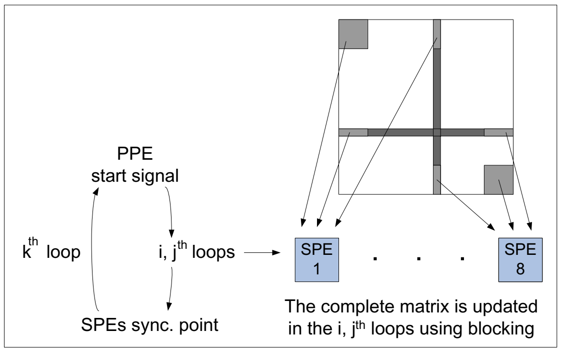

PPE
start signal
k^th loop
i, j^th loops
SPEs sync. point
SPE 1
SPE 8
The complete matrix is updated in the i, j^th loops using blocking

Figure 5. Parallelism for Standard Floyd Warshall Algorithm

3 Parallelization of Floyd Warshall Algorithm

The two inner loops (i^{th} and j^{th} in Figure 2), in which all elements of the matrix are updated by using the k^{th} row and k^{th} column, can be parallelized since updating one element in the matrix is independent of updating another element. The matrix can be divided into blocks ([2], [3]) and each block is updated by each SPE independent of the other part. We can make the allocation of blocks to SPEs statically and introduce it before compile time. Figure 5 shows the parallelism of the steps in Figure 2 on the SPEs of the Cell processor.

3.1 Analysis

The same analysis for computation as in Section 2.2 applies here. Below we present analysis for data communication, synchronization and scheduling.

Data communication costs: Each element of the NxN matrix has to be updated in a single k^{th} iteration. As the complete matrix does not fit into the local store we used blocking with a maximum block size that can fit into the local store of the SPE. All blocks have to be copied into one of the SPE's cache, updated and copied back to main memory. Copying back into main memory is necessary after each iteration because of the dependency of each iteration on the previous one. So that, k^{th} row and k^{th} column of the matrix used to update all the elements of the matrix, are ones that have been generated in the $k-1^{th}$ iteration. Assume the time taken to copy each element of the matrix is 1 time unit.

$$Total\ memory\ to\ SPE\ traffic = \Theta(N^3). \quad (4)$$

Scheduling and Synchronization Costs: At the beginning of each k^{th} iteration, the PPE gives a start signal to all SPEs. Each SPE updates its own part of the matrix, by copying chunks of BxB blocks along with the corresponding $2B$ elements from the k^{th} row and column from the main memory. After an SPE completes updating its part of the matrix, it sends back a finish signal to the PPE. Once the PPE gets the finish signal from all SPEs, it gives the start signal for the $(k+1)^{th}$ iteration. There is no scheduling cost because the partition of all work to the SPEs is embedded inside them and is done at compile time (static scheduling). There is a synchronization point after the i^{th} and j^{th} loops. But it takes very less time due to the following reasons:

- The compute work is divided equally among the SPEs. So the SPEs take almost the same amount of time for computation.
- There is only one synchronization point after working on the complete matrix (data communication and computing of the order $O(N^2)$). So the time of computation and data communication are far greater than synchronization.

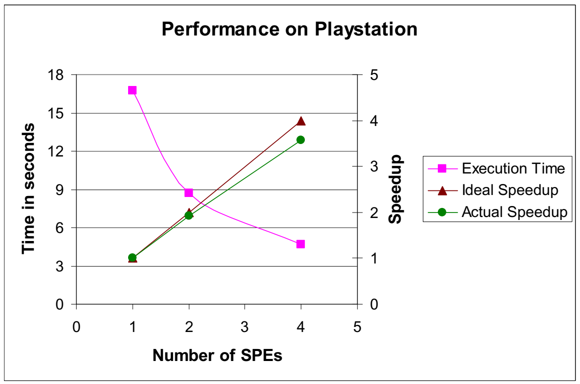
Estimate for data communication time: We compute the exact data communication cost estimate with a simple analysis. We measure time by dividing the overall data exchange between SPEs and main memory by the bandwidth and do not consider complexities such as time taken for maintaining memory coherency etc. We assume we operate on single precision floating point values (4 bytes). The overall data exchange in the form of BxB blocks being read and written back is $2N^3$. Also, for updating a single BxB block, an extra $2B$ elements are copied (parts of the k^{th} row and column used for updating the block). Hence for N^3 amount of updated data, extra $2N^2$ amount of data is copied.

Hence, the time taken for data communication is:

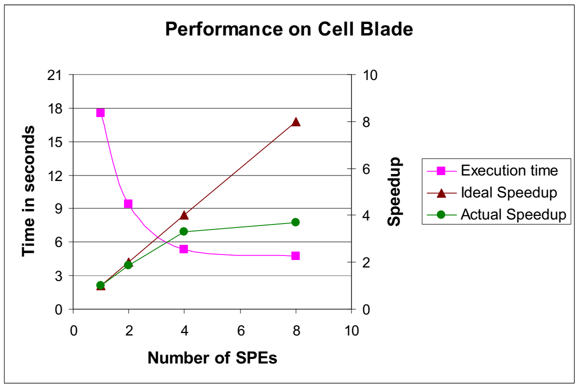
$$Data\ comm.\ time\ = (((2N)^3 + 2(N)^2) * 4)/bw. \quad (5)$$

Plugging in actual values of bw = 25GB/s and N = 2048 we get the value to be 2.75 seconds (primarily dominated by the N^3 term).

The above analysis shows that even though the time for computations scales with the number of cores, the stress on memory traffic is still remaining. So depending on how optimal the computation time is, the scalability is going to break as we increase the number of cores when computation time falls below data communication time. In the next section, we present experimental results to support our analysis.



(a) Scalable Performance on the 4 cores of the Playstation



(b) Scalability breaking on the Cell Blade

Figure 6. Performance results on the Playstation and Cell Blade (N = 2048)

3.2 Experimental Results

Figure 6(a) shows results of the parallel algorithm on the Playstation. These results show that specious scalability can be achieved as a result of code with bad performance in computation time. In other words, scalability can be seen up to a certain number of cores, if the computation part of the code running on a single SPE is not efficient and is greater than data communication and other overheads.

To verify the above argument, we looked at the performance of the code on a Cell processor on a blade which has 8 SPEs. Figure 6(b) shows that the scalability is breaking at 4 SPEs. We profiled the code separately to compare computation, data communication and synchronization to the overall execution time and obtained the following results on the 8 cores of the Cell processor in a blade:

- Time for computation: 2.722s
- Time for data communication: 4.108s
- Time for synchronization: 0.040s
- Overall execution time: 4.150s

The above results and all other profiling results in later sections are got by performing control experiments. i.e, Except for the part to be profiled, the remaining part of the code is commented and the execution time is measured. We declared the data as volatile so that the compiler does not optimize the code by removing any computation or communication. The measurements are accurate to up to $\pm$ 0.05s. The results can be interpreted as follows:

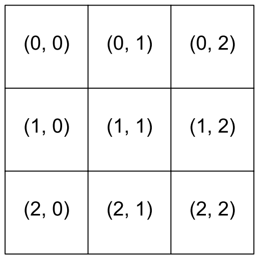
- The overall execution time is dominated by the data communication time.
- Due to double buffering, there is an overlap of computation and data communication. This leads to masking of computation time by data communication time in the overall execution time.
- The execution time can be no less than the data communication time. This is because while the computations scale with the number of SPEs, data communication cost does not scale accordingly. This corroborates our analysis from equation 4. In fact as number of cores increase the stress on bandwidth increases when all cores try to access the memory simultaneously.
- Synchronization time is very small compared to computation and data communication time.

All of this corroborates our analysis and shows that data communication is the bottleneck for the Standard Floyd Warshall algorithm.

4 Blocked Floyd Warshall Algorithm

4.1 Algorithm description

We partition W, the original adjacency weight matrix, into blocks of size BxB. B is the blocking factor. Figure 7(a) shows a blocked 12x12 matrix (blocking factor is 4). The Figure also shows the numbering scheme (row major)



(a) Blocked Matrix

```
1. Start with $D^0$
2. for (round = 1; round < N/B; round++)
3.     Phase 1: Update block (round, round) for B iterations
4.     Phase 2: Update block in row (round) and column (round) for B iterations
5.     Phase 3: Update remaining blocks of the matrix for B iterations
6.Return $D^n$
```

(b) Steps of the algorithm

Figure 7. Blocked Floyd Warshall Algorithm

used for the blocks. The Blocked Floyd Warshall algorithm will perform B iterations of the outermost loop of Floyd Warshall algorithm (Figure 2) on each block of Figure 7(a) before advancing to the next B iterations. Each set of B iterations is divided into 3 Phases. For example, in Phase 1 of the first set of B iterations, the equation in Figure 1 is used to compute D^B for the elements in block (0, 0). Since these B iterations access only the elements within block (0, 0), it is called a self-dependent block in the first B iterations. In Phase 2 of the first B iterations, D^B for the remaining blocks (0, *) and (*, 0) that are on the same row and column as the self dependent block, is computed using the elements in the blocks being updated and the self dependent block. In Phase 3, D^k is computed for all the remaining blocks (i.e., for blocks that are not on the same row and column as the self dependent block). Then the next set of B iterations are computed with the self dependent block being block (1, 1) in Figure 7(a), and so on.

The Blocked Floyd Warshall algorithm is shown in Figure 7(b). For a more detailed description of the ideas and proof of correctness of the algorithm, we ask the reader to refer to [3], [2]. We concentrate more on analysis of the algorithm in the context of a multicore system in the next few sections. The key ideas to take away from the above description are:

- Once a block is copied from main memory to cache, B iterations are performed on it before writing it back to the main memory.
- In a single round, 1 block is updated in Phase 1, $2(N/B)-1$ blocks in Phase 2 and $(N/B)^2-2(N/B)$ in Phase 3.
- The self dependent block only is required to update itself in Phase 1.

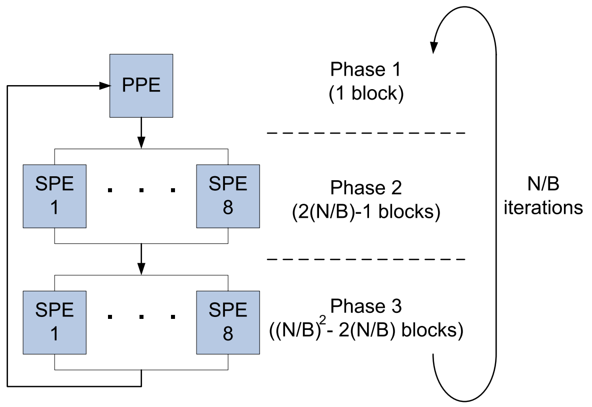


Figure 8. Parallelism for Blocked Floyd Warshall algorithm

- The self dependent block and the block itself are required to update a block in Phase 2.
- Elements from 2 corresponding blocks of Phase 2 are required to update a block in Phase 3.

4.2 Parallelization

Similar to the standard algorithm layout, the adjacency matrix is laid out in blocks ([2], [3]) in the memory. This suites the design of the algorithm too, since it is block oriented.

By a careful observation of the operations in a single round, it is evident that once the self dependent block is updated, blocks in Phase 2 can be updated in parallel. Once Phase 2 blocks are updated, blocks in Phase 3 can be updated in parallel. Hence, considering that (N/B) is large compared to the number of cores on the chip, there is enough parallelism in Phase 2 and Phase 3 of the algorithm. Figure 8 shows the possible mapping of the algorithm on to the Cell processor.

4.3 Analysis

The analysis for computation work in Section 2.2 still applies because the computations are not different for the Standard and Blocked Floyd Warshall algorithm. We analyze the data communication, synchronization and scheduling time below.

4.3.1 Data Communication cost

The data communication cost is reduced by a factor of B because, once a block is brought in, B iterations are performed on it before writing it back to the main memory. In Section 4.3.4 we compute a more exact formula using hardware parameters to measure the bound on the data communication time.

$$Total\ data\ communication\ cost = \Theta(N^3/B). \qquad (6)$$

4.3.2 Scheduling and Synchronization costs

We initially started with a dynamic scheduler where the PPE assigns Phase 2 and Phase 3 tasks by passing addresses of blocks according to the schedule to the SPEs as and when they become free. It does through mailbox messaging ([9]). So when an SPE completes updating a block it notifies the PPE by sending a message and the PPE in return sends the address of the new blocks needed for the next operation.

There are 3 synchronization points dictated by the algorithm (see Figure 8). But through intelligent scheduling, the synchronization points between Phase 2 and Phase 3, Phase 3 and Phase 1 can be eliminated. This can be done by scheduling the Phase 2 and Phase 3 operations in the following order:

- Execute Phase 2 blocks to the right of self dependent block in the same row.
- Execute Phase 2 blocks below the self dependent block in the same column.
- Execute Phase 2 blocks above self dependent block in the same column.
- Execute Phase 2 blocks to the left of self dependent block in the same row.
- Execute Phase 3 blocks to the right and below of self dependent block.
- Execute Phase 3 blocks to the right and above the self dependent block.
- Execute Phase 3 blocks to the left and above the self dependent block.
- Execute Phase 3 blocks to the left and below self dependent block.

With the above ordering, for the case of large graphs where number of blocks in a row is greater than the number of cores, the barriers between Phases 2 and 3 and Phases 3 and 1 can be removed.

4.3.3 Scalability analysis

Assuming the scheduling and synchronization overheads are small, the computation and data communication costs decide the scalability of the algorithm. Therefore, Total execution time = computation cost (eqn. 2) + data communication cost (eqn. 6).

$$= \Theta(N^3/m) + \Theta(N^3/B). \qquad (7)$$

The asymptotic analysis shows that the computation cost scales as the number of cores while the data communication cost scales as the block size (which is dependent on the

```
1. Get event, address of first buffer from PPE (scheduling)
2. DMA in first set of buffers (communication)

3. for (till event != stop)
   {
4.    Get event, address of second buffer from PPE (scheduling)
5.     DMA in next set of buffers - double buffering (communication)

      // Operate on current buffer (computation)
6.    for (k = 1 to B)
7.       for (i = 1 to B)
8.          for (j = 1 to B)
9.             d(i, j) = min ( d(i, j), d(i, k) + d(k, j) )

10.   DMA out updated buffer (communication)
   }

// Operate on last buffer
10. for (k = 1 to B)
11.    for (i = 1 to B)
12.       for (j = 1 to B)
13.          d(i, j) = min ( d(i, j), d(i, k) + d(k, j) )

14. DMA out updated buffer (communication)
(event is to indicate whether it's a phase 2 or phase 3 operation)
```

Figure 9. SPE code for Blocked Floyd Warshall algorithm

cache size of each core). We already computed an estimate for a bound on the computation time in Section 2.2. In the next section we estimate bounds on the data communication time.

4.3.4 Estimate for data communication time

Figure 9 shows the SPE code. Lines 5, 11 and 16 are the data communication part of the code where the SPE copies the blocks from the main memory. We compute the time taken for the transfers of the 4 BxB size blocks between the cores and main memory (3 because Phase 3 blocks require 3 blocks and 1 block is updated and written back). We assume we operate on single precision floating point values (4 bytes). This is done $(N/B)^2$ times in i^{th} and j^{th} loops and N/B times in k^{th} loop (referring to the overall algorithm). The estimate for data communication time is given by:

$$((4 * B^2 * 4)/bw) * (N/B)^2 * (N/B). \qquad (8)$$

Plugging in actual values of B = 64, bw = 25GB/s and N = 2048 we get the data communication time to be 0.0858 seconds.

The above analysis shows that this algorithm mapping can be scalable as the estimated time for computation is greater than that for data communication. One constraint is that synchronization and scheduling time should be less compared to the overall execution time.

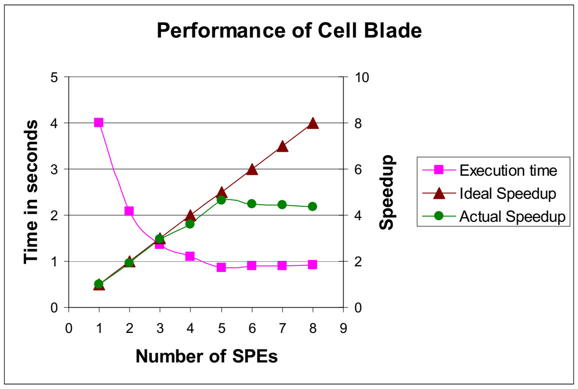


Figure 10. Experimental results showing scalability breaking after 5 SPEs on Cell blade(N = 2048)

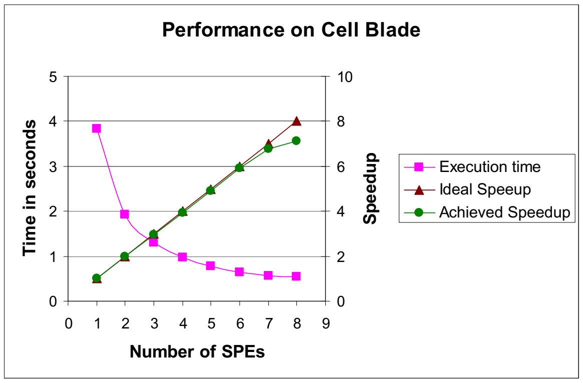


Figure 11. Experimental results showing scalability for 7 SPEs (N = 2048)

4.4 Optimizations for SPE Computations

Apart from the algorithmic optimizations for data communication and co-ordination discussed in Section 4.3, we also did code level optimizations for the computation part of the algorithm (lines 6-10 in Figure 9). By ensuring that the computation part of the algorithm runs fast, we remove the possibility of specious scalability mentioned in Section 3.2. We did the following code level optimizations: Double buffering, SIMDization, high loop unrolling factor (8) and code movement to remove dependency stalls.

4.5 Experimental Results

The experimental results in Figure 10 show that scalability is breaking at 5 SPEs. We profiled the execution time and obtained the following results:

- Total execution time for 6 SPEs: 0.81sec.
- Total scheduling and synchronization time: 0.60sec

The above results show that the scheduling and synchronization time to pass messages around from the PPE to SPEs is overshadowing the computation and data communication time. This implies PPE scheduling is bottleneck because all SPEs are interacting with the PPE and the PPE is acting as the single master control. Because of the dynamic scheduling, the nature of the code written was such that we could isolate scheduling and synchronization together but could not isolate computation or data communication from synchronization and scheduling. We were only left with options to use better methods for synchronization and scheduling to reduce their effect before we could make any comments about computation or data communication time after 6 SPEs.

5 Optimizing Synchronization and Scheduling

5.1 Synchronization

To improve synchronization between PPE and SPEs, we used Fenced DMA ([9]). This led to a reduction in scheduling and synchronization time to about 0.43 seconds. Figure 11 shows scalable results of our implementations on the Cell processor in a Cell blade up to 7 SPEs. This has shown good results and near scalability but we still couldn't make analysis of computation and data communication because 0.43 seconds is still a high percentage of the overall execution time.

5.2 Self-Scheduler

Self-scheduling in a multicore processor is the mechanism by which, cores schedule themselves in accordance with the ongoing execution of the algorithm on the whole multicore chip. In our implementation of Floyd Warshall algorithm on the Cell processor, this is achieved by an initial setup and distribution of schedule information to the SPEs (prior to the start of execution of the algorithm) after which, each SPE has independent access to its own schedule information for the overall execution of the algorithm and the intermediate synchronization points. The SPEs continue to work on their own without interacting with the PPE for the algorithm scheduling information until they reach a global synchronization point at which time they would signal the PPE. Once the PPE gets the signal from all SPEs, it sends a start signal and the SPEs start processing the next set of operations until they reach the next global synchronization point.

We designed a self-scheduler for the SPEs so that all the dynamic scheduling through message passing is removed and SPEs don't interact with the PPE for task scheduling. The self-scheduler has the schedule information which is nothing but indices of the blocks of the Phase 2 and Phase 3 operations of the Blocked Floyd Warshall algorithm in the order discussed in Section 4.3.2, embedded into the SPEs. This schedule is distributed to the SPEs so that SPEs execute it in that order. The algorithm description is shown in Figure 12.

```
1. Start with $D^0$ and initial setup of schedule information
2. SPEs copy schedule information for first round(= 0)
3. for (round = 0; round < N/B; round++)
4.     PPE updates Phase 1 (round, round) block
5.     PPE sends start signal to SPEs and waits for completion signal
6.     SPEs double buffer next round schedule independently
7.     SPEs update blocks of Phases 2 and 3 independently
8.     SPEs send completion signal back to PPE
9. Return $D^n$
```

Figure 12. Algorithm description of Blocked Floyd Warshall with Self-Scheduling

To illustrate the idea, we give part of the schedule information for the following sample problem: Let N = 512, B = 64, Number of partitions = 8 x 8. The schedule for round = 7 of the blocked algorithm is such that:

- First the Phase 1 block(3, 3), is updated by the PPE.
- Next Phase 2 blocks (3, 4), (3, 5), (3, 6), (3, 7), (4, 3), (5, 3), (6, 3), (7, 3), (0, 3), (1, 3), (2, 3), (3, 0), (3, 1), (3, 2) are updated by the SPEs.
- Phase 3 blocks - remaining blocks that are not updated in the previous two steps - are updated.

The self-scheduler contains the above information, distributed to the SPEs as follows:

- SPE 0 updates (3, 4), (0, 3), (4, 6), (6, 6), (0, 6), (2, 6), (2, 0), (5, 2).
- SPE 1 updates (3, 5), (1, 3), (4, 7), (6, 7), (0, 7), (2, 7), (2, 1), (6, 0).
- SPE 2 updates (3, 6), (2, 3), (5, 4), (7, 4), (1, 4), (0, 0), (2, 2), (6, 1).
- SPE 3 updates (3, 7), (3, 0), (5, 5), (7, 5), (1, 5), (0, 1), (4, 0), (6, 2).
- SPE 4 updates (4, 3), (3, 1), (5, 6), (7, 6), (1, 6), (0, 2), (4, 1), (7, 0).
- SPE 5 updates (5, 3), (3, 2), (5, 7), (7, 7), (1, 7), (1, 0), (4, 2), (7, 1).
- SPE 6 updates (6, 3), (4, 4), (6, 4), (0, 4), (2, 4), (1, 1), (5, 0), (7, 2).
- SPE 7 updates (7, 3), (4, 5), (6, 5), (0, 5), (2, 5), (1, 2), (5, 1).

Such information is constructed for all rounds and distributed to the SPEs, prior to the start of the algorithm execution (step 1 in Figure 12). This allows SPEs to execute their part of the algorithm, independent of other SPEs, and

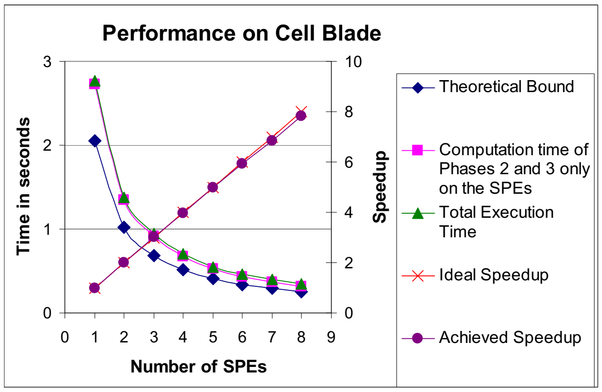


Figure 13. Experimental results on Cell Blade showing scalability for 8 SPEs (N = 2048)

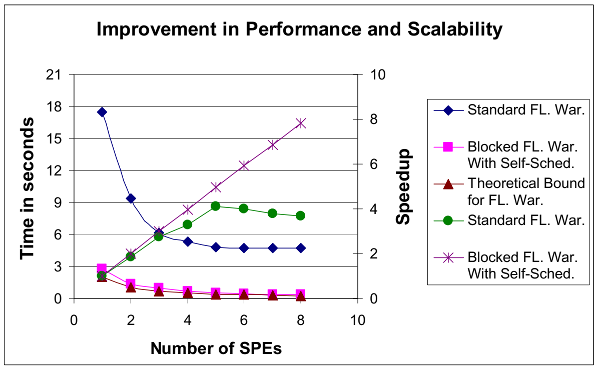


Figure 14. Summary of performance and scalability (N = 2048)

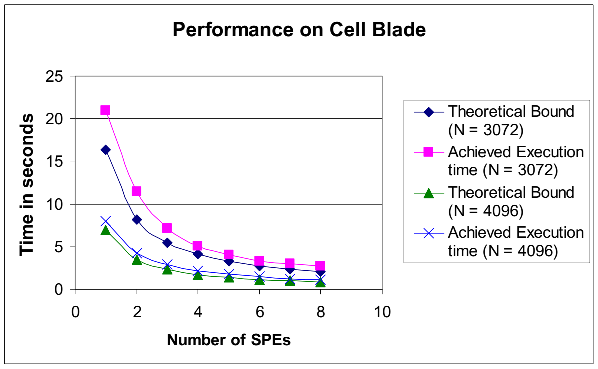


Figure 15. Performance for N = 3072, 4096

without any communication with the PPE, except the synchronization point at the end of a round.

Doing this optimization led to orders of magnitude of reduction in scheduling and synchronization time and helped achieve excellent results as shown in Figure 13. We point out 2 issues here:

1. Referring to Figure 13, data is obtained for increasing number of SPEs, for every SPE added. There are alignment issues to be taken care of when constructing the scheduler at compile time to distribute jobs to the SPEs. This is because it is straight forward to distribute jobs if the number of SPEs is 1, 2, 4 or 8. But it is not so straight forward if the number of SPEs are 3, 5, 6 or 7. This required us to add dummy jobs as a padding to construct the scheduler. We have also taken care of this aspect and were able to do this with negligible affect on performance and scalability.
2. The scheduling information is actually large that it does not fit into the local store of the SPE. So we double buffered parts of the schedule information and got the appropriate part at the appropriate time into the local store of an SPE.

The following are the time taken for individual components in the overall execution time for 8 SPEs:

- PPE computation (Phase 1): 0.01279s
- Scheduling and synchronization: 0.01285s
- Data communication: 0.09623s
- SPEs computation (Phases 2, 3): 0.3127s
- Computation time (All Phases): 0.3246s
- Total execution time: 0.3422s

For an algorithm to be scalable, one of the important issues is the scheduling and synchronization times should be much lesser than the computation time. In Figure 11, though the implementation is scalable till 7 SPEs, due to synchronization and scheduling costs, it breaks at the 8th SPE and cannot be scaled with larger number of cores. Whereas we achieved true scalability with the self-scheduler (as shown in Figure 13). We will comment on the scalability of our final implementation in the Section 6.

5.3 Performance

We unrolled the loops with an automatic code generator and used the IBM's Assembly Visualizer tool for code movement. This lead to highly optimized computation time and helped us to achieve the performance shown in Figure 13. In the same Figure, we also compare the achieved execution time with the theoretical lower bound computed in Section 2.2.

One issue to mention here is that the theoretical bound is computed by parallelizing computations of Floyd Warshall algorithm on the 8 SPEs. Our mapping of the Blocked Floyd Warshall algorithm involves execution of Phase 1 blocks on the PPE. We support our methodology because the amount of work in Phase 1 (1 block) is far less compared to the amount in Phases 2 and 3 ($(N/B)^2 - 1$ blocks). This is also evident from Figure 13 which shows that the overall execution time is dictated by the time taken for Phase 2 and Phase 3 computations only. This shows that we paral-

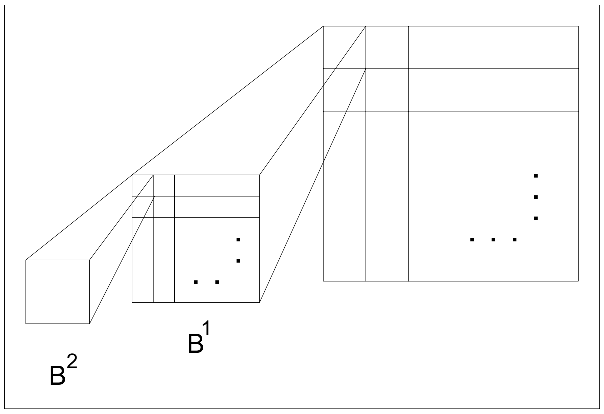


Figure 16. Multi-level tiling for Blocked Floyd Warshall algorithm schedule and layout

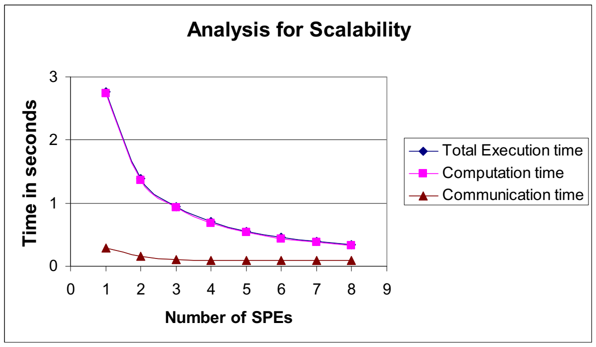


Figure 17. Variation of computation and data communication time with number of SPEs (N = 2048)

lelized the computations of the algorithm taking care of data communication, synchronization, scheduling overheads and also the part of the computations being executed on the PPE.

We achieved this performance by removing the stalls and optimizing only the inner most loop. For the scope of this paper, we focused on parallelizing the algorithm and did not concentrate on optimizations at the level of single SPE. If the code on the SPEs is completely optimized, it is possible to reach even closer to 102.4GFLOPS. This is supported by the algorithmic analysis and the above experimental results showing that computation time $>$ communication time.

6 Discussion and Conclusion

We comment briefly on extending the ideas in this paper for a cluster of multicore systems. The main feature of a cluster, apart from the distributed compute power, is the multi-level memory organization. Consider an example of the memory organization of the cluster in which, there is a shared memory for all the systems of size M (level-1), each system has a DRAM of size M^1 (level-2) and the local store is of size M^2 (level-3). When faced with such a multi-level memory hierarchy, one could consider a multi-level blocking method ([2]) for both the schedule and the data layout. Consider a multi-level blocking method such as the method shown in Figure 16. In this method, B^2 would be chosen as big as possible to fit into M^2 to minimize data communication between M^2 and M^1. Similarly, B^1 would be chosen to minimize the data communication between M^1 and M. The self-scheduler is directly related to the blocking size, as discussed in Section 5.2, and is constructed for each level of the memory hierarchy (multi-level scheduling). So, a self-scheduler is constructed at level-1 for the complete matrix, with blocking of size B^1. Level-2, needs a self-scheduler constructed with block size B^2. Here the top level dictates the number of iterations in its lower level. So whenever the level-1 schedules B^1 iterations on the level-2, the level-2 block will complete the B^1 iterations of the Blocked Floyd Warshall algorithm, with blocking of size B^2. This modular nature of the algorithm mapping technique can also be extended for more levels of memory hierarchies. Below, we comment on the performance and scalability we achieved and conclude the paper.

We achieved performance close to the theoretical bound for the Floyd Warshall algorithm on the Cell processor by showing the parallelization and optimization of the Blocked Floyd Warshall algorithm. Figure 14 shows this comparison. Figure 15 shows the results for N = 3072, 4096.

As number of cores increase, the scalability of the final implementation (Figure 13) will not be affected by synchronization and scheduling overhead. This is because, firstly after the initial distribution of the schedule of each core to the core itself, there is one synchronization point after 3 phases of the blocked algorithm. Therefore the synchronization time is negligible compared to the computation. Secondly, regarding scheduling of a core which is managed by the core itself, we have already shown the case where scheduling information is large and has to be double buffered with the computations. So, no matter how large the graph is or how many number of cores the processor has, each core can divide its schedule into modules and double buffer it with computations, hence not affecting the scalability. Also, the scheduling information, a list of indices of the blocks in the matrix, is very small compared to the actual data in the adjacency matrix. This is also evident in the time profiling results in Section 5.2. Hence, we are more concerned about the data communication time of the matrix rather than the scheduling information.

Analyzing the computation time and communication time as shown in Figure 17 by increasing the number of SPEs, the computation time scales with the number of SPEs as expected. The communication time initially decreased and remained fairly constant from 4 to 8 SPEs. The initial decrease is because a better usage of bandwidth as number of SPEs increase. After 4 SPEs, the data communication time is around 0.1077 seconds which means the rate of data is exchange is around 20GB/s (peak being 25.4GB/s). From eqn. 7 we know that while the computation scales as the number of cores, data communication scales as the square root of the amount of local store memory for each core (Section 4.3.3). Hence, as number of cores on the Cell processor increase, to achieve good scalability (computation, not data communication to be the bound), the amount of local store should also be increased accordingly.

The ideas discussed in this paper, being based on a generic architecture model (Figure 3), can be applied for other multicore platforms such as Intel/AMD. But apart from variations in the values taken by the model parameters, Cell processor gives the control of memory management to the programmer. Whereas we have to work with the operating system and compiler closely to control the memory management in the case of Intel/AMD multicore platforms. In future we plan to extend these ideas to a cluster of Cell processors, large graphs and different multicore platforms.

7 Acknowledgment

The authors thank Ling Zhuo and Yinglong Xia for assisting this research work. The authors acknowledge David Bader, Virat Aggarwal, Samuel Williams, Fabrizio Petrini, Mike Kistler and members of IBM Cell forums for their suggestions on Cell programming.

References

[1] J. A. Kahle, M. N. Day, H. P. Hosftee, C. R. Johns, T. R. Maeurer, D. Shippy, *Introduction to the Cell multiprocessor*, IBM Jr. of Research and Develepment 49(4/5)(2005)589-604.

[2] M. Penner, V. K. Prasanna, *Cache Friendly Implementations of Transitive Closure*, Proc. of PACT, 2001.

[3] G. Venkataraman, S. Sahni, S. Mukhopadhyaya, *A blocked all-pairs shortest-paths algorithm*, ACM Journal on Experimental Algorithmics, Article 5, Vol. 8, 2004.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, Second Edition, The MIT Press 2001.

[5] O. Villa, D. P. Scarpazza, F. Petrini, J .F. Peinador, *Challenges in Mapping Graph Algorithms on Advanced Multi-core Processors*, Proc. of IPDPS, 2007.

[6] J. Kurzak, J. Dongarra, *Implementation of Mixed Precision in Solving Systems of Linear Equations on the Cell Processor*, Concurrency and Computation: Practice and Experience, Volume 19, July 2007.

[7] D. Bader, V. Agarwal, *FFTC: Fastest Fourier Transform on the IBM Cell Broadband Engine*, Proc. of HiPC, 2007.

[8] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, K. Yelick, *Scientific Computing Kernels on the Cell Processor*, IJPP, 2007.

[9] A. Arevalo, R. M. Matinate, M. Pandlan, E. Peri, K. Ruby, F. Thomas, C. Almond, *Programming the Cell Broadband Engine: Examples and Best Practises*, IBM Redbooks.

[10] A. V. Aho, R. Sethiand, J. D. Ullman, *Compilers Principles Techniques and Tools*, Addison Wesley, Reading, Mass.

[11] S. Sippu, E. Soisalon Soininen, *Parsing Theory volume I Languages and Parsing*, Springer Verlag Berlin 1988.

[12] S. Ceri, G. Gottlob, L. Tanca, *What you always wanted to know about Datalog (and never dared to ask)*, IEEE Trans. on Knowledge and Data Engineering,1(1):146-166,1989.

[13] O. Beaumont, A. Legrand, F. Rastello, Y. Robert, *Dense linear algebra kernels on heterogeneous platforms: redistribution issues*, Parallel Computing, 28:155-185, 2002.

[14] X. Liao, Y. Yang et al, *Analysis of algorithms for matrix multiplication*, Journal on Numerical Methods and Computer Applications, Vol. 6, No. 4, 1985, pp. 215-222.

[15] Y-J. Lee, P. Diniz, M. Hall, R. Lucas, *Empirical Optimization for a Sparse Linear Solver: A Case Study*, IJPP, Vol 33, No. 2-3, , 165-181.

[16] P. Raghavan, M. A. James, J. C. Newman, B. R. Seshadri, Scalable Sparse Matrix Techniques for Modeling Crach Growth, Proc. of PARA'02, pp. 588-602, 2002.

[17] Q. Lu, S. Krishnamoorthy, P. Sadayappan, *Combining analytical and empirical approaches in tuning matrix transposition*, Proc. of PACT, 2006.

[18] J. Dongarra et al, *An Updated Set of Basic Linear Algebra Subprograms (BLAS)*, ACM Trans. on Mathematical Software, 28(2):135-151, 2002.