Parallelizing the Floyd-Warshall Algorithm on Modern Multicore Platforms: Lessons Learned

Students of the *Parallel Processing Systems* course
School of Electrical & Computer Engineering
National Technical University of Athens

***Abstract*—The well known Floyd-Warshall (FW) algorithm solves the all-pairs shortest path problem on directed graphs. In this work, we parallelize the standard FW and two cache-friendly versions using three different parallel programming environments, namely OpenMP, Cilk and Threading Building Blocks. We experimented with multiple alternative parallel versions, in order to gain insight on the execution behavior of the parallelized algorithms on modern multicore platforms, and on the programmability of the aforementioned environments. We were able to significantly accelerate FW performance utilizing the full capacity provided by the multicore architectures used.**

I. Introduction

Since Moore firstly formulated his famous "Moore's Law" in the mid-60's, there was a rapid and massive evolution of computer hardware. From the introduction of integrated circuits with thousands of transistors on a single chip and the use of cache memory, to the implementation and maintenance of enormous supercomputers and multicore systems, there is a vast number of technological improvements and ground-breaking ideas that made the computer platform what we know today.

Computer architecture and its numerous pioneers have played important role in this evolution. Firstly, Von Neumann defined his classical Harvard model and proposed that a single storage structure for holding both instructions and data would be both feasible and efficient. Later, floating point arithmetic emerged and the first operating systems appeared. Progressively, main memory became bigger and faster by using more levels of cache and sophisticated architectures. Nowadays, computer scientists are pushing computers' performance even further. This led to the realization that clock frequency has already reached its peak and thus new optimizations are required. So, they focused on designing systems with multiple cores, leading to today's supercomputers with thousands of cores and hardware threads.

Clearly, the answers to many of the issues arising is parallel computing i.e., accelerating applications by using multiple cores, based on the notion that larger inputs may be divided into disjoint parts which may be solved almost independently. Once all these independent parts are solved, one must subsequently combine their intermediate answers. The key-observation is that we may parallelize those independent works by assigning each sub-problem to a different processor or thread. There are two main policies or *parallelizing strategies*: data centric and task centric. As the names indicate, the first concentrates on the computation of the data, meaning that each thread (or processor) contributes to the computation of a subset of the original data, whereas the second strategy tries to split the necessary work and assign it to all available processors.

To those familiar with parallel programming, it is commonly acknowledged that parallelizing applications is neither trivial, nor straightforward. Several algorithms need to be redesigned from scratch in order to be parallelized and effectively utilize vital system resources such as cache memory. The contribution of this work is the study of three alternative versions of the well known *Floyd-Warshall (FW)* algorithm by using three different parallel programming environments (namely OpenMP, Cilk and Threading-Building Blocks), and moreover draw conclusions about the drawbacks and the advantages of each of them, for sufficiently large input graphs.

The outline of this work is as follows: Section II presents information about FW and its alternative implementations. Section III focuses on the parallel programming environments used. The methodology of the most successful parallel implementations of FW are presented in Section IV. Experiments establishing the performance characteristics of each implementation are given in Section V. Finally, Section VI summarizes the conclusions drawn from this work.

II. Background

The FW is a classic dynamic programming algorithm that solves the *all-pairs shortest path (APSP)* problem on directed weighted graphs $G(V, E, w)$, where $V = \{1, ..., n\}$ is a set of nodes, $E \subseteq V \text{x} V$ are the edges and w is a weight function $E \rightarrow R$. The number of nodes is denoted by n and the number of edges by m. The output of the algorithm is typically in tabular form: The entry in i's row and j's column is the weight of the shortest path between nodes i and j. FW runs in $\Theta(V^3)$ time and is also used to compute the transitive closure of a graph. FW can be applied to graphs with negative weight edges to determine whether the graph has negative cycles or not [9]. FW is used in many real-life applications, such as bioinformatics for clustering correlated genes [6],[8], in database systems for optimizing SQL queries [7] or in data mining [2]. The standard FW algorithm in shown in Alg. 1.

Algorithm 1: The Floyd-Warshall (FW) algorithm

```
1: for k = 1 → n do
2:    for j = 1 → n do
3:       for i = 1 → n do
4:          A_k[i,j] = min(A_{k-1}[i,k] + A_{k-1}[k,j], A_{k-1}[i,j])
5:       end for
6:    end for
7: end for
```

Regarding its execution behavior, FW is memory bound since, in order to compute the A_k matrix, we need first to compute the A_{k-1} matrix. Two alternative implementations of FW have been proposed in order to improve its cache performance, the first based on recursion [3] and the second based on tiling [8]. The recursive implementation (*FW_SR*) performs automatic blocking at every level of the memory hierarchy. The pseudo-code of FW_SR is shown in Alg. 2. The initial call to the recursive algorithm passes the entire input matrix as each argument. Subsequent recursive calls pass quadrants of the input arguments as shown in Fig. 1. At any level of recursion, the arguments A, B and C point to the same or different subsets of the input matrix. The recursive FW algorithm contains certain computational dependencies which require a specific ordering of subsequent recursive calls. Note that the first four recursive calls

operate on the matrix from the top left quadrant to the bottom right quadrant and the last four calls in a reverse order of the first four calls. By tweaking the size of the base case according to cache memory size, we can effectively reduce the processor-memory traffic by a factor of B, where B is $\sqrt{cachesize}$, achieving the optimal lower bound on processor-memory traffic [8].

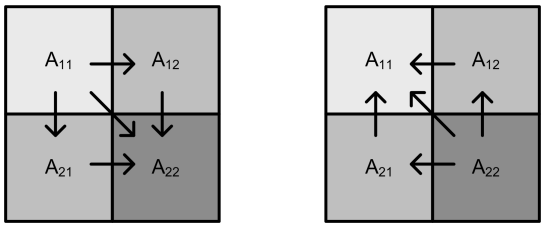


Fig. 1. FW_SR. The matrix divided into 4 submatrixes with names A_{11}, A_{12}, A_{21}, A_{22}

Algorithm 2: Recursive FW (FW_SR)

```
1: FWI (A, B, C)
2: for k = 1 → n do
3:    for j = 1 → n do
4:       for i = 1 → n do
5:          A[i][j] = min(A[i][j], B[i][k]+C[k][j]);
6:       end for
7:    end for
8: end for

9: FWR (A, B, C)
10: if base case then
11:    FWI (A, B, C)
12: else
13:    FWR (A11, B11, C11);
14:    FWR (A12, B11, C12);
15:    FWR (A21, B21, C11);
16:    FWR (A22, B21, C12);
17:    FWR (A22, B21, C12);
18:    FWR (A21, B21, C11);
19:    FWR (A12, B11, C12);
20:    FWR (A11, B11, C11);
21: end if
```

On the contrary, tiling is a commonly used technique to achieve higher data reuse in looped code. The tiled version of FW (*FW_TILED*) works as follows: Initially, the input matrix is divided into tiles of size B. During the k-th block iteration, the algorithm updates the k-th diagonal tile (black, *CR* tile in Figure 2) first, then it updates the tiles in the remainder of the k-th block row and block column (grey, *E, W, N, S* tiles) and finally it updates the remaining tiles of the matrix (white, *NE, NW, SE, SW* tiles). This way, all dependencies are satisfied. FW_TILED reduces processor-memory traffic by a factor of B (where B is the order of the cache size) and is asymptotically optimal among all implementations with respect to processor-memory traffic [8]. FW_TILED is shown in Alg. 3.

Algorithm 3: Tiled FW (FW_TILED)

```
1: for k = 0 → n step B do
2:    FW(CR);
3:    for tile in E, W, N, S do
4:       FW(tile);
5:    end for
6:    for tile in NE, NW, SE, SW do
7:       FW(tile);
8:    end for
9: end for
```

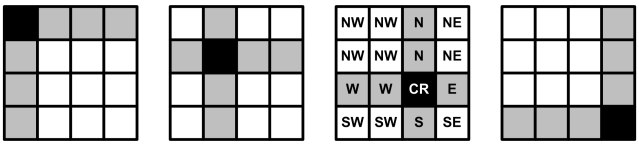


Fig. 2. FW_TILED. The matrix is divided into tiles of size B.

III. Parallel Programming Environments

In this section we briefly describe the parallel programming environments used in our experimentation and outline their most important characteristics. In order to illustrate the syntax used by each tool, we implement a standard for-loop and a Fibonacci implementation in all three platforms. The standard serial implementations are provided in code listings 1 and 2.

```
for (i=0; i!=100; ++i) Foo(a[i]);
```

Listing 1. Serial for-loop example

```
if(n==1) return 0;
if(n==2) return 1;
return fibonaci(n-1) + fibonaci(n-2);
```

Listing 2. Code snippet from serial Fibonacci calculation

A. OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) for parallel programming intended to work on shared-memory architectures. More specifically, it is a set of compiler directives, library routines and environmental variables, which influence run-time behaviour. OpenMP enables parallel programming in various languages, such as C, C++ and FORTRAN and runs on most operating systems. The current version of OpenMP is 3.1 and was released in 2011 [11].

The OpenMP API uses the fork-join model of parallel execution. Multiple threads perform tasks defined implicitly or explicitly by OpenMP directives. All OpenMP applications begin as a single thread of execution, called the initial thread. The initial thread executes sequentially, until it encounters a parallel construct. At that point, this thread creates a group of itself and zero or more additional threads and becomes the master thread of the new group. Each thread executes the commands included in the parallel region, and their execution may be differentiated, according to additional directives provided by the programmer. At the end of the parallel region, all threads are synchronized.

The runtime environment is responsible for effectively scheduling threads. Each thread, receives a unique id, which differentiates it during execution. Scheduling is performed according to memory usage, machine load and other factors and may be adjusted by altering environmental variables. In terms of memory usage, most variables in OpenMP code are visible to all threads by default. However, OpenMP provides a variety of options for data management, such as a thread-private memory and private variables, as well as multiple ways of passing values between sequential and parallel regions. Additionally, recent OpenMP implementations introduced the concept of *tasks*, as a solution for parallelizing applications that produce dynamic workloads. Thus, OpenMP is enriched with a flexible model for irregular parallelism, providing parallel while loops and recursive data structures. The *parallel_for* construct on OpenMP along with Fibonacci implementation, are shown in code listings 3 and 4.

```
#pragma omp parallel for
for (int i=0; i!=100; ++i) Foo(a[i]);
```

Listing 3. OpenMP parallel for

```
if (n<16) return serialFibonacci(n);
else {
   #pragma omp task shared(x)
      x = RecFib(n-1);
   #pragma omp task shared(y)
      y = RecFib(n-2);
   #pragma omp taskwait
   return x+y;
}
```

Listing 4. OpenMP code snippet for parallel Fibonacci

B. Cilk

Cilk is a linguistic and runtime technology for algorithmic multithreaded programming developed at MIT. The philosophy behind Cilk is that programmers should focus on structuring their program to expose parallelism and exploit locality and Cilk's runtime system will take care of efficiently scheduling threads on a given platform. Cilk's runtime system is capable of handling issues, such as load balancing, synchronization and communication protocols. Cilk is algorithmic since its runtime system guarantees efficient and predictable performance [5].

Cilk is a faithful extension of the C programming language that adds three new keywords for parallel execution and synchronization. The keyword *cilk* identifies a Cilk procedure definition, which is the parallel analogue of a C function and consists of a list of arguments and a function body. A Cilk procedure may also spawn subprocedures in parallel and synchronize them upon completion. Specifically, one may create new Cilk procedures (tasks) by using the keyword *spawn*. When a Cilk procedure is spawned, its parent thread continues to execute in parallel with the child, and the Cilk scheduler is responsible for monitoring all subsequent spawned procedures. The *sync* statement is used to define local *barriers*, in the sense that when sync is used the parent procedure suspends and does not resume until all its children complete their work. Listing 5 presents a recursive implementation of the Fibonacci function in Cilk.

```
if (n<16) return serialFibonacci(n);
else {
   x = spawn fib (n-1);
   y = spawn fib (n-2);
   sync;
   return (x+y);
}
```

Listing 5. Cilk code snippet for parallel Fibonacci

Since July 2009, Cilk language and its trademarks has been acquired by Intel Corporation. Intel advanced the technology further and released a commercial implementation along with Intel's compilers, featuring additional data parallel constructs with the name Intel Cilk Plus [4] (Sep. 2010). Cilk Plus adds array extensions, eliminating the need for several keywords and adding a new one. Intel Cilk Plus uses only three keywords (*cilk_spawn*, *cilk_sync* and *cilk_for*) while also adding the ability to spawn function involved in reduction operations. Listing 6 shown the *parallel_for* construct supported by Cilk Plus. In August 2011, Intel has announced that it is maintaining Cilk Plus as a branch of gcc 4.7. The runtime library is available dual-licensed, including a BSD-3 license. Cilk Plus [5] provides additional parallel programming features like the *inlet* and *abort* keywords.

```
cilk_for(int i=0; i!=100; ++i) foo(a[i]);
```

Listing 6. Cilk Plus parallel for

C. Threading Building Blocks

Threading Building Blocks (TBB) is a programming tool, initially developed by Intel in 2004, which takes advantage of multi-core processors over shared memory platforms. TBBs is not a new programming language and therefore, does not require any customized compilers. Consequently, it is portable to several operating systems and architectures. TBB is actually a C++ template library which provides algorithm templates and data structures for writing parallel applications. Intel launched version 1.0 in 2006 and version 2.0 on July, 2007 when the product actually became open source. As of version 2.2 TBBs supports lambda expressions, a new feature of C++ that enables the use of anonymous functions.

TBBs treats operations as tasks and distributes them to cores using two basic techniques; recursive splitting and task stealing. The library recursively splits the load to small chunks until a minimum limit (*Grain-size*) is reached. Tasks are allocated to individual cores, which execute them depth-first in order to optimize cache memory use. Additionally, idle cores (i.e, with no work assigned to their queues) 'steal' tasks from busy cores and either execute them or further split them. The *work stealing* technique is effectively used for load balancing and is implemented breadth-first in order to maximize parallelism. TBBs are using tasks instead of threads, because tasks are light-weight and therefore problems arising from creating and synchronizing threads are avoided. TBBs provide a wide variety of algorithm templates and data structures. Some of them are *task_group*, *parallel_for*, *parallel_reduce*, *parallel_scan* and *parallel_pipeline*.

Listings 7 and 8 show a *parallel_for* implementation for TBBs. Initially we have to define a new class ApplyFoo and construct a function object which contains the work to be executed in every loop subspace.

```
class ApplyFoo {
   float *const my_a;
   public: ApplyFoo(float *a) : my_a(a) {}
   void operator() (const blocked_range<int>& r)
   const {
      float *a=my_a;
      for (int i=r.begin(); i!=r.end(); ++i)
         Foo(a[i]);
   }
}
```

Listing 7. TBB parallel for

In order to use the new class ApplyFoo in the a function:

```
parallel_for(blocked_range<int>(0,100), ApplyFoo(a));
```

Listing 8. TBB parallel for usage

In the previous example, *blocked_range<int>* is a class that indicates an one dimensional range over the type int. Additionally, TBBs provide blocked_range2D for two-dimensional ranges, in order to allow programmers to define their own block range classes. Furthermore parallel_for constructs may have an optional argument, namely the partitioner, which controls splitting. The *simple_partitioner* uses recursive binary splitting, the *affinity_partitioner* uses ranges which optimize the cache locality and *auto_partitioner* uses a heuristic method to define the right range.

Finally, Listing 9 shows the combination of lambda expressions and *task_group* construct in the Fibonacci algorithm implementation. In this function, if $n > 16$, a tbb task group is initialized and the two recursive calls of ParallelFib are assigned to two new tasks that are generated by the *g.run* method. The parameter [&] indicates that

variables x, y are passed by reference. Method *g.wait()* synchronizes the two tasks.

Threading Building Blocks are also compatible with other threading packages, meaning that C++ applications may contain alternately TBBs structures, OpenMP directives or Cilk code. TBBs promise high performance and scalable applications and are especially focused on programmers who are not thread experts, since the library uses inherently high-level programming logic.

```
if (n<16) return serialFibonacci(n);
else {
    tbb::task_group g;
    g.run( [&] { x= ParallelFib(n-1);} );
    g.run( [&] { y= ParallelFib(n-2);} );
    g.wait();
    return x+y;
}
```

Listing 9. TBB code snippet for parallel Fibonacci

IV. Methodology

In this section we present several parallel versions of FW, FW_SR and FW_TILED. In the classic FW, one can easily notice, that the nested i and j for-loops are totally independent and therefore parallelizable. A *parallel_for* construct, applied on the outer loop of i yields the first, straightforward parallelization of FW shown in Listing 10.

```
for(k=0; k<n; k++)
    parallel_for(i=0; i<n; i++)
        for(int j=0; j<n; j++)
            A[i][j]=min(A[i][j], A[i][k] + A[k][j])
```

Listing 10. Parallel FW with parallelized loop i

As far as the recursive FW is concerned, Fig. 1 shows that the antidiagonal blocks A12 and A21 of the input matrix may be computed in parallel. For this purpose, we implemented a task-centric approach. Two parallel tasks are spawned (each of them executes computations on the anti-diagonal A12 and A21 blocks) which must be synchronized before continuing the execution of the remaining blocks (Listing 11).

```
FW_SR(A11);
task_spawn FW_SR(A12);
task_spawn FW_SR(A21);
synchronize();
FW_SR(A22);
FW_SR(A22);
task_spawn FW_SR(A21);
task_spawn FW_SR(A12);
synchronize();
FW_SR(A11);
```

Listing 11. Parallel FW_SR with tasks

The prospects of parallelizing the tiled FW algorithm have already been implied by the analysis in Section II and Figure 2. At the k-th iteration, after the central block is updated, all cross blocks (E, W, S, N) may be executed concurrently, as their data-dependent NE, NW and SE blocks have already been evaluated up to the (k-1)-th iteration. Next, all exterior blocks may be executed concurrently, as the formulation of the algorithm ensures that all data-dependencies within the k-th iteration are satisfied. Listing 12 is an example of a parallel implementation of the tiled version. At each iteration of the outermost loop k, FW algorithm is executed on the current block. A new task is spawned for each of the cross blocks. After those tasks are synchronized, new tasks are spawned, one for each of the NW, NE, SW and SE exterior blocks, and synchronized.

After our analysis of the three different FW algorithms and the extracted guidelines for their parallelization presented above, our research group split into three teams, in order to experiment with the proposed parallel versions of the FW algorithm, on the platforms environments presented in Sec. III. In the following paragraphs we discuss the parallel implementations that provided interesting performance results.

```
for (k=0; k<n; k++)
   FW(CR);
    foreach (tile in E, W, N, S)
        task_spawn FW(tile);
    synchronize();
    foreach (tile in NE, NW, SE, SW)
        task_spawn FW(tile);
    synchronize();
```

Listing 12. Parallel FW_TILED with tasks

A. OpenMP

As far as OpenMP is concerned, the versions implemented that caught our attention were four. In the classic FW version, all iterations of the i-loop can be performed totally in parallel. Using the directive *#pragma omp parallel for* above the i-loop (Listing 10), the iterations were distributed to a group of threads by one master thread. The variables A, k were declared shared, whereas j was declared private, since each thread uses a separate copy of this variable.

The recursive FW algorithm may be parallelized by applying the *#pragma omp task* OpenMP directive, instead of using parallel for loops, which led to poor performance for this particular algorithm. As mentioned previously, some function calls need not wait for their turn in the serial order, in order to take place (Listing 11). By placing the directive *#pragma omp task* and embody the parallel calls, we create a separate task for each call and place it in the parallel pool. After all tasks are declared, each available thread 'grabs' a task from the task pool in order to execute it. When all such tasks are completed, the threads must synchronize (OpenMP directive *taskwait*) before moving on to the rest of the program. Note that in the recursive FW, it was more efficient to check the parallel pool - and grab idle threads from it, the fewer possible times, in order to avoid the additional overhead of creating new threads. Therefore, the use of *#pragma omp parallel* was limited down to one, in the beginning of the code.

As for the tiled FW algorithm, the best parallel version required the use of *parallel for with dynamic scheduling*, i.e., once a particular thread finishes its loop iterations, it returns to get another one from the iterations that are left - in each for-loop of the algorithm. Firstly, we experimented on calculating all the chunks of the matrix, that form the formerly mentioned cross, simultaneously, and afterwards calculating the rest of the matrix, for each k iteration. The best results were achieved when the calculation of the cross and the rest of the matrix split into five and four parts respectively (Fig. 2). This slight adjustment enabled us to parallelize the for-loops that appear in the tiled FW algorithm, thus lowering the overhead of creating more threads. Another worth mentioning parallel implementation of the tiled FW version, used exclusively tasks. The directive *#pragma omp task* is placed in each for-loop, and the variables involved, are declared first private. This way, data is private to each thread, but is initialized using the value of the variable using the same name from the master thread. Although this version did not achieve very good results, it was still compared with the rest, in the following sections.

Overall, despite the fact that several experimental measurements had to be studied in order to find the best results for each parallel version of the FW algorithm, once we understood the way OpenMP

operated in a lower level, it was quite easy to increase performance. Consequently, it is obvious that OpenMP is a suitable platform for quick, easy and safe parallelizing of ones already existing code. It is easy to learn, quick to use and supported by most of the existing compilers. Nevertheless, we should always be careful with some details that have significant impact on performance, such as the decent use of parallel pool, as mentioned previously.

B. Cilk

Regarding the classic FW algorithm, each iteration of the i loop may be performed in parallel with the rest, i.e., each row of the input matrix may be computed independently. A first approach was to distribute the $n = |V|$ iterations of the i loop to a group of Cilk processes. Each spawned Cilk process, undertakes the task to compute the rows it has been committed to. We call this version *cilk_fw_parallel_for_i*. The second version was the *cilk_fw_parallel_for_i_j* version. Since our version of Cilk, does not support *cilk_for*, we alternatively used the following pseudocode:

```
Cilk void run_loop(first, last)
{
  if (last - first) < grainsize)
     for (int i=first; i<last ++i) LOOP_BODY;
     else {
        int mid = (last-first)/2;
        cilk\_spawn run_loop(first, mid);
        run_loop(mid, last);
     }
}
```

Listing 13. Pseudocode for cilk_fw_parallel_for_i_j

At the first level, we keep "cutting" matrix A horizontally and spawning Cilk processes (one process per slice), following the pattern of the above pseudocode in order to distribute the iterations of the i loop. When these slices become too small, we spawn one Cilk process per slice row. At a second level, each Cilk process that was spawned at this point, uses the same pattern of the above pseudocode to fill in the row it has been committed.

Regarding the recursive FW algorithm, since the anti-diagonal blocks A12 and A21 may be computed in parallel, we spawn Cilk processes for each of them and sync them appropriately, as shown in Listing 11. Cilk has performed very well on recursion and this fact is mirrored in the analysis of the experimental results for this particular version (cilk_fw_sr).

Finally, regarding the tiled FW algorithm, we developed two additional versions: a) The first version, combines the recursive and tiled algorithms, by setting the central, the cross and the exterior blocks of the tiled version, to execute the recursive FW algorithm. We refer to this version as *cilk_fw_tiled_sr*. b) In the second version, which we call *cilk_fw_tiled_parallel_for_i* we grouped the tiles that lie outside the cross in four groups. Not surprisingly, the tiled-recursive approach yields the best performance, as is further discussed in the experimental section.

C. TBBs

Concerning the classic FW algorithm, we built three different versions, all based on the *parallel_for* construct. The first two focus on the parallelization of the i-loop. Their only difference is the type of partitioner used. The first version (*tbb_fw-parallel_for_i-simple*) uses the *simple_partitioner*. The second one (named *tbb_fw-parallel_for_i-affinity*) uses *affinity_partitioner*. The third implementation (*tbb_fw-parallel_for_i_j-affinity*) parallelizes both i and j loops by applying parallel_for on the two-dimensional range object defined by the loops' bounds. As in the one-dimensional case, the two-dimensional plane is recursively subdivided, simultaneously in both dimensions. An affinity partitioner is used in this case, as well.

In the recursive FW algorithm, we have implemented two similar parallel versions, where concurrent tasks are created for the execution of Floyd-Warshall on the anti-diagonal blocks A12 and A21. The first version spawns two tasks, each one executing the recursive parallel algorithm on blocks A12 and A21, respectively. The rest of the blocks need to be processed sequentially, so the *wait()* method is used to synchronize concurrent tasks before entering a sequential region. This version is referred to as *tbb_fw_rec-tasks*. To decrease the time spent on task creation and potential task stealing, the second version *tbb_fw_rec-tasks_one* spawns a new task only for one of the anti-diagonal blocks and leaves the execution of the second one to the initial thread.

The parallelization scheme of the tiled FW algorithm unfolded numerous possible implementation options, by employing tasks and parallel_for operations, or even blending them in a nested fashion. Four of the implementations exhibited high performance levels and are worth mentioning. The first version (*tbb_fw_tiled-tasks_med_grain*) is depicted in Fig. 3. At first, the central block is being processed by a single task. Then, the edges of the cross are mapped to four tasks and processed in parallel. Finally, the exterior planes are cut horizontally into rows of blocks, and each such row is assigned to a different task. Creation, execution and synchronization of tasks between the three phases have been described earlier.

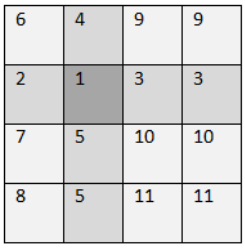


Fig. 3. TBB: tbb_fw_tiled-tasks_fine_grain - Distribution of blocks to tasks

The second version (*tbb_fw_tiled-tasks_fine_grain*) is a variant of the previous one. Instead of having a single task processing all blocks on an edge of the cross, we spawn a new task for each different block. The exterior blocks are mapped in the same way as in *tbb_fw_tiled-tasks_med_grain*. Apparently, *tbb_fw_tiled-tasks_fine_grain* employs parallelism at a finer granularity compared to *tbb_fw_tiled-tasks_med_grain*. Figure 4 displays the tasks spawned in this version. The third version (*tbb_fw_tiled-tasks_parallel_for*) employs nested parallelism. At the higher level, a different task is created for each edge of the cross and each one of the exterior planes. Internally, using the parallel_for construct, each edge or plane is being recursively split into smaller chunks which in their turn are mapped to multiple tasks. Effectively, the assignment of blocks to tasks is the same as in the case of *tbb_fw_tiled-tasks_fine_grain*.

The fourth version (*fw_tiled-parallel_for_mixed_tiles*) uses solely the parallel_for construct for implicit task creation. To minimize parallelization and synchronization costs, the four loops originally processing the edges of the cross are now merged into a single loop. Likewise, the loops processing the exterior planes are combined into a single, double-nested for-loop. A *parallel_for* construct is applied on each of the new loops, but to enable efficient exception of blocks that should not be processed in a given phase (e.g. all the cross blocks in the third phase) we devised block-level indexing along with conditional execution. Fig. 5 shows an example distribution of blocks

8	5	11	11
2	1	3	4
9	6	12	12
10	7	13	13

Fig. 4. TBB: fw_tiled-tasks_med_grain, fw_tiled-tasks-parallel_for - Distribution of blocks to tasks

in this scheme.

5	2	5	5
2	1	3	4
6	3	6	6
7	4	7	7

Fig. 5. TBB: fw_tiled-parallel_for_mixed_tiles - Distribution of blocks to tasks

V. Experiments

In this section we present performance results, in terms of execution time, for the various parallel versions of the FW algorithm and aforementioned parallel programming environments. This section is structured as follows: In Sec. V-A, we briefly describe the experimental methodology along with the architecture of the multicore testbeds. In Sec. V-B, V-C, and V-D, we present graphs of the execution times for parallel versions created with OpenMP, Cilk and TBBs respectively, which exhibited maximum speedup or otherwise noteworthy behavior. In Sec. V-E we compare the best performing versions produced with each tool. Moreover, we record the maximum acceleration achieved in comparison to the sequential classic FW algorithm.

A. Experimental Setup

The execution times of all parallel versions of the FW algorithm were measured on two different platforms: a clovertown-based multicore node with 2 clovertown CPUs and a dunnington-based multicore node with 4 dunnington CPUs. Henceforth, the clovertown-based multicore node will be referred to as *clovertown* and the dunnington-based multicore node as *dunnington*. The architecture of *clovertown* is depicted in Fig. 6(a). This node contains 2 clovertown CPUs which share one main memory module (2GB). Each clovertown CPU contains 4 cores (Intel® Xeon® E5335 @ 2.00GHz). Each core has a private 32KB Level 1 cache and each pair of adjacent cores shares a 4MB Level 2 cache. The architecture of *dunnington* is depicted in Fig. 6(b). This node contains 4 dunnington CPUs which also share one main memory module (30GB). Each dunnington CPU contains 6 cores (Intel® Xeon X7460® @ 2.66GHz). Each core has 32KB Level 1 cache, each pair of adjacent cores shares a 3MB Level 2 cache and every 4 cores share a 16MB Level 3 cache.

We measured serial and parallel execution times for matrices of sizes 1024×1024, 2048×2048, and 4096×4096 and block sizes of $8, 16, 32, 64, 128, 256, 512$. For each parallel version, we took 3 time measurements for each architecture and calculated the mean in order to obtain an accurate time measurement for the respective version. Due to space limitations, we present measurements for *clovertown*. However, as explained earlier, experimentation was not limited to this node and any noteworthy behaviour of a parallel version observed on *dunnington* will also be mentioned.

B. OpenMP Experimental Results

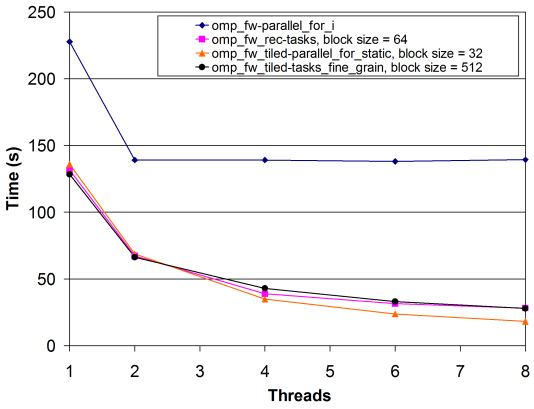


Fig. 7. Execution times for matrix size 4096×4096, for the best performing parallel versions created with the OpenMP framework (clovertown)

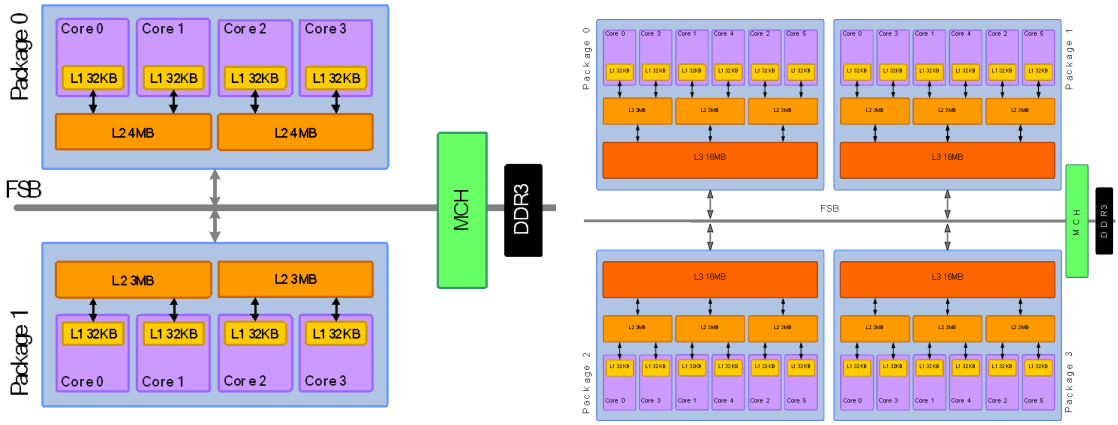
Figure 7 shows the execution times on *clovertown* of the best performing parallel versions created using the OpenMP framework, for matrix size 4096×4096. For the parallel classic FW algorithm by using parallel-for (*omp_fw-parallel_for_i*), we observe that our implementation does not scale for more than two cores on *clovertown*. This was expected, because the Level 2 cache memory of the clovertown CPUs is not big enough to fit the input matrix. Therefore, transactions with the main memory are frequent, thus the time advantage of parallel execution is negated.

The parallel recursive FW algorithm, in which we used tasks with block size 64 (*omp_fw_rec-tasks*), scales much better than *omp_fw-parallel_for_i*, since now each chunk fits in the Level 2 caches. Specifically, we achieved an overall decrease in time, from *131.65*s in one core, to *28.26*s when using 8 cores. However, in this implementation, we should not open the parallel pool many times, because in this case the overhead of creating new threads is big enough to reduce performance.

Finally, for the parallel tiled FW algorithm, two implementations are worth mentioning. The first one uses parallel-for with blocksize 32 (*omp_fw_tiled-parallel_for*), and is - by far - our most efficient implementation achieving time of *18.29*s. Note, that in this version, we used static scheduling, since we know the number of iterations beforehand and therefore static is preferable to dynamic scheduling. The second alternative version uses tasks - just like in the parallelization of the recursive version- with dynamic scheduling and block size 512 (*omp_fw_tiled-tasks_fine_grain*). This version, scales in a manner similar to *omp_fw_rec-tasks*, which is less than that for *omp_fw_tiled-parallel_for*. This may be due to the overhead of task creation and may not be a disadvantage in systems with greater processing power. Note that since tasks are a relatively new feature in OpenMP, we may see further improvement in the future.

C. Cilk Experimental Results

Figure 8 shows the execution times on *clovertown* of the best performing parallel versions created using the Cilk language, for matrix size 4096×4096. For the parallel classic FW algorithm, the best times were achieved with the *cilk_fw-parallel_for_i_j* for grain size b=256 (*125.44*s, workers=4). Although we observe that our parallel program is faster than its serial counterpart, this happens



(a) Architecture of the clovertown-based multicore node (b) Architecture of the dunnington-based multicore node

Fig. 6. Multicore architectures used as experimental testbeds.

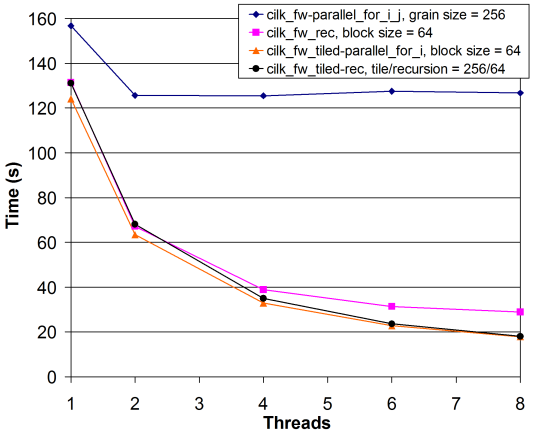


Fig. 8. Execution times for matrix size 4096×4096, for the best performing parallel versions created using the Cilk language (clovertown)

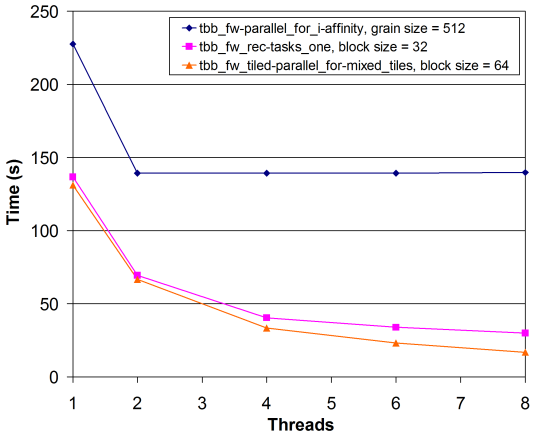


Fig. 9. Execution times for matrix size 4096×4096 for the best performing parallel versions created using the TBB library (clovertown)

only if the number of workers remains less than four. For bigger values of workers the execution time remains the same or becomes greater. So, the Cilk implementation of the classic FW algorithm does not scale.

For the parallel recursive FW algorithm, the best performance (*28.96*s) was achieved by *cilk_fw_rec* for grain size b=64. This is a significant improvement due to the fact that computations were executed in parallel, so that the utilization of cache memory is more efficient. Unfortunately for more than 6 cores, this implementation does not scale further.

For the parallel tiled FW algorithm, two different implementations are worth mentioning, namely *cilk_fw_tiled-rec* and *cilk_fw_tiled-parallel_for_i*. Figure 8 shows that those two implementations provided the best acceleration achieved. The best times (*17.96*sec) were achieved by using *cilk_fw_tiled-parallel_for_i* for b=64. The fact that the parallel tiled FW version provided the best performance gains was expected, since we use our caches optimally, thus avoiding unnecessary memory transactions.

D. TBBs Experimental Results

Figure 9 shows the execution times on *clovertown* for the best performing parallel versions created using the TBB library, for matrix size 4096×4096. For the parallel classic FW algorithm, the best performance on *clovertown* was achieved by *tbb_fw-parallel_for_i-affinity* for block size 512. However, this parallel version does not scale beyond 2 cores on this architecture, because a matrix of 4096×4096 integers exceeds the Level 2 cache capacity of the clovertown CPUs. On the contrary, results on *dunnington* showed, that this version scales almost linearly, due to the increased Level 2 cache capacity. However, even for *dunnington*, parallelizing any of the two inner for loops of the classic FW algorithm, without specifying the use of an affinity partitioner, significantly degrades performance. Hence the affinity partitioner provided by the TBB library can indeed efficiently allocate loop iterations to threads and ensures maximum cache utilization.

For the parallel recursive FW algorithm, the best performance on *clovertown* was achieved by *tbb_fw_rec-tasks_one* for block size 32. However, results showed that both versions *tbb_fw_rec-tasks_one* and *tbb_fw_rec-tasks* exhibit similar performance and the difference between execution times does not exceed *300*ms, on any of the two architectures, for matrix size 4096 x 4096 and for optimal block sizes. This observation demonstrates that task creation using the TBB library has small overhead. For optimal block sizes, these versions scale linearly for up to 2 cores, sublinearly for up to 8 cores, but then no further improvement is achieved for larger number of cores (as *dunnington* results showed). This is due to the relatively small amount of work that can be executed in parallel for the recursive FW algorithm.

For the parallel tiled FW algorithm, the best performance (*16.87*s) on *clovertown* was achieved by *tbb_fw_tiled-parallel_for-mixed_tiles* for block size 64. It should be noted here that this parallel version exhibits similar performance to *tbb_fw_tiled-tasks_parallel_for*. These

Tool	Parallel Version	Fastest Time (s)
OpenMP	tbb_fw_tiled-parallel_for-mixed_tiles	18.30
Cilk	cilk_fw_tiled-parallel_for_i	17.96
TBBs	tbb_fw_tiled-parallel_for-mixed_tiles	16.87

TABLE I
FASTEST EXECUTION TIMES ACHIEVED ON *clovertown* FOR EACH TOOL FOR MATRIX SIZE 4096×4096

parallel versions scale almost linearly for all available cores. This can be attributed to the fact that the tiled FW algorithm is cache oblivious and therefore the majority of the memory accesses are cache hits and all data may be computed in parallel.

E. Overall Comparison

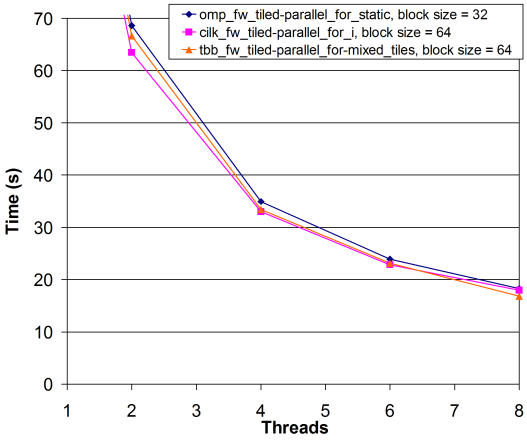


Fig. 10. Execution times for matrix size 4096×4096, for the best performing parallel versions created utilizing all tools (clovertown)

Figure 10 shows the execution times on *clovertown* of the best performing parallel versions constructed using the the three tools, for matrix size 4096×4096. Note, that the best parallel versions for all tools exhibit almost similar performance. Table I shows the fastest execution times achieved for each tool, for the same matrix size. Since, the sequential classic FW algorithm for matrix size 4096×4096 requires *145.11*s on *clovertown*, the maximum overall speedup relative to the sequential classic algorithm achieved was *8.6* and was achieved using TBBs.

F. Summary

In this section, we have experimented with multiple parallel versions of the FW algorithm and the available parallel platforms. Results showed that for input matrix of size 4096×4096, the classic FW algorithm's performance in *clovertown* does not increase beyond 2 cores. This was expected because a matrix of this size does not fit into L2 memory cache of clovertown CPUs. Conversely, for *dunnington* due to larger L2 memory, performance increases for larger numbers of cores. For the recursive FW algorithm, results are better and we achieve significant acceleration, up to 6-8 cores. Adding more cores does not further increase performance, because for matrices of the aforementioned size, a little work may be done in parallel beyond a certain number of cores. Finally, the tiled FW algorithm achieved the best results, due to its cache oblivious nature. In this case, we achieved computation times of *17-18*s for all available platforms. Although TBBs achieved the best results (speedup greater than 8), similar results were achieved for all platforms.

As far as ease of use is concerned, OpenMP it is a little more mature (it is the older tool after all) and was the easiest tool for adapting our existing codebase to its parallel counterpart. TBBs and Cilk's support of *parallel_for* constructs is relatively new or partly incomplete and therefore those two platforms required additional work to adapt our serial code to parallel.

VI. CONCLUSIONS

In this work we have experimented with the well known Floyd-Warshall algorithm, which solves the all-pairs shortest path problem on directed graphs. We parallelized the three most well known serial implementations of this algorithm (classic FW, recursive FW and tiled FW) by using three different parallel programming environments, namely OpenMP, Cilk and Threading Building Blocks. Our experimentation on two separate architectures and with a multitude of alternative parallel versions has showed that the cache friendly nature of the tiled FW algorithm makes it more suitable for parallelization. Moreover, although the various parallel platforms used had varying degrees of learning curves, all platforms exhibited quite similar performance and therefore it is up to the programmers to decide which platform suits their needs best.

ACKNOWLEDGMENTS

The following students contributed effort for this paper (in alphabetical order): Ioanna-Maria Alifieraki, Athanasios Andreou, Christos Andrikos, Dimitrios Bliablias, Athanasios-Akanthos Chasapis, George Chatzikonstantis, Georgios Chatzopoulos, Alexandros Daglis, Alexandros Efentakis, Nikolaos Eftaxiopoulos-Sarris, Athena Elafrou, Nikolaos Grivas, Nefeli Halastani, Natalia Hering, George Karachalias, Athanasios Karmas, Evdokia Kassela, Stefanos Koffas, Dimitris Konomis, Emmanouil Koukoutos, Sofia-Ira Ktena, Leonidas Lampropoulos, Dionysios Manousakas, Iosif Moulinos, Konstantinos Mouzakitis, Vasileios Nakos, Evridiki-Vasileia Ntagiou, Avraam Papadopoulos, Nikela Papadopoulou, Lydia Polyzou, Vasilios Priskas, Georgios Psaropoulos, Alexios Pyrgiotis, Georgios Retsinas, Nikolaos Sarris, Zisis-Iason Skordilis, Ioannis Spiliopoulos, Konstantinos Stamatoukos, Panagiotis Traganitis, Xaris Tsiboukakos, Antonis Tsigkanos, Nikolaos Tsironis, Ilias Tsitsimpis, Vassilis Tsounis, Vasileios Tsoutsouras, Thomas Vaiou, Georgios Zervakis, Ioannis Zobolas.

REFERENCES

[1] Robert D. Blumofe and Charles E. Leiserson, *Scheduling Multithreaded Computations by Work Stealing*, http://supertech.csail.mit.edu/papers/steal.pdf
[2] Borgwardt, Kriege,*Shortest-path kernels on graphs.*
[3] Jeremy D. Frens and David S. Wise, *Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code.* In Proc. of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, June 1997.
[4] Cilk 5.4.6 Manual, http://software.intel.com/en-us/articles/intel-cilk-plus-specification
[5] Intel Cilk Plus, http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf
[6] Akihiro Nakaya, Susumu Goto, Minor Kanehisa,*Extraction of Correlated Gene Clusters by Multiple Graph Comparison*
[7] C. Papadimitriou, M. Sideri,*On the Floyd-Warshall Algorithm for Logic Programs.*
[8] J.S. Park, M. Penner, and V. K. Prasanna, *Optimizing Graph Algorithms for Improved Cache Performance* IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 15, NO. 9, SEPTEMBER 2004.
[9] T.H. Cormen, C. Stein, R.L. Rivest and C.E. Leiserson, *Introduction to Algorithms*, McGraw-Hill Higher Education
[10] R.D. Blumofe and C.E. Leierson, *Scheduling Multithreaded Computations by Work Stealing*, http://supertech.csail.mit.edu/papers/steal.pdf
[11] OpenMP, http://openmp.org