

# Rapid Development of Parallel Blocked All-Pairs Shortest Paths Code for Multi-core Computers

Peiyi Tang

Department of Computer Science  
University of Arkansas at Little Rock  
Little Rock, AR 72204  
Email: pxtang@ualr.edu

**Abstract**—In this paper, we present the rapid development of efficient multi-core parallel code for the blocked Floyd-Warshall algorithm, using the C++ library `gae.h` [4], on top of the Intel C++ Threading Building Blocks (TBB) library. We demonstrate that developing parallel multi-core code using `gae.h` is easy and fast. The parallel code developed with `gae.h` is also efficient. The efficiency of the parallel code developed for the blocked Floyd-Warshall algorithm on an 80-core HP ProLiant DL980 G7 multi-core machine is above 70% if less than 70 cores are used and above 90% if less than 20 cores are used.

## I. INTRODUCTION

Multi-core computers have become mainstream platforms, while parallel programming for multi-core computers has not. The major issues are two-fold: parallel programming should be as easy to implement as in mainstream languages such as C++ and the parallel code generated should be efficient to exploit the full potential of multi-core computers. The Intel C++ Threading Building Blocks (TBB) library [1] has made tremendous progress in this direction. It shields the programmer from having to write tedious multi-threaded programs using the Pthread library and provides many generic algorithms such as `parallel_for`, `parallel_reduce` and `parallel_scan` for parallel programming. The parallel code generated by TBB are also efficient. TBB also supports a task-graph programming model [2], also known as graph-driven asynchronous execution or a superscalar sequence pattern [3], where the computation is organized as a collection of tasks and the tasks are ordered only by data dependencies between them. To program superscalar sequences in TBB still requires significant effort: programmers need to analyze and specify data dependencies between the tasks. In our previous work, we developed a C++ library called `gae.h` to ease the superscalar sequence programming in TBB [4] for dense linear algebra algorithms. The data dependencies between the tasks are detected and recorded automatically when tasks are *registered* in the task graph by following the order in the original serial algorithms. Programming superscalar sequences in `gae.h` is easy and fast and the parallel code generated is efficient.

The Floyd-Warshall algorithm for finding all-pairs shortest paths shares a lot of common features with dense linear algebra algorithms such as Cholesky factorization and Gauss-Jordan elimination: they all update an  $n \times n$  two-dimensional matrix  $n$  times through the outermost loop and each element of the matrix is updated at most once in each iteration of the loop. All dense linear algorithms have their blocked versions [5]–

[7]. Blocked algorithms use the blocked matrix [8] to increase cache locality. Blocked algorithms are also essential for forming tasks of larger granularity in parallel programs, to reduce the overhead of parallel tasks. In blocked algorithms, the  $n \times n$  matrix is blocked into a  $d \times d$  matrix of smaller matrices, each of which is a  $b \times b$  matrix, where  $b = n/d$  (we assume  $d$  divides  $n$ ) and  $b$  is called the *block size*. The outermost loop is blocked to a loop of  $d$  iterations, each of which is to update the blocked matrix. While the Floyd-Warshall algorithm does not have blocked version, Venkataraman et al. proposed in [9] a blocked algorithm which produces the same result (the shortest distances of all-pairs) as the original Floyd-Warshall algorithm.

In this paper, we are going to present the rapid development of superscalar sequence code in TBB using `gae.h` for the blocked all-pairs shortest paths algorithm. We also use NICTA Armadillo, an efficient C++ library for linear algebra algorithms [10], for the rapid development of code. The performance evaluation on an 80-core computer shows that the parallel code generated is very efficient: the efficiency, the speedup divided by the number cores used, is above 70% if the number of cores used is below 70. The efficiency is above 90% if less than 20 cores are used.

The rest of this paper is organized as follows. Section II introduces the blocked Floyd-Warshall algorithm for the all-pairs shortest paths problem to be parallelized in this paper. Section III describes the rapid development of parallel code using `gae.h`. Section IV shows the performance results of the parallel code on an 80-core computer. Sections V and VI discuss related work and concludes the paper, respectively.

## II. BLOCKED FLOYD-WARSHALL ALGORITHM

Consider a directed graph  $(V, E)$ , where  $V$  and  $E$  are the node and edge sets, respectively, and a function  $W : E \rightarrow \mathcal{R}$  such that  $W(e)$  for every  $e = (i, j) \in E$  is the distance from node  $i$  to node  $j$ . Assume that  $|V| = n$  and the nodes in  $V$  are numbered as  $1, 2, \dots, n$ . The function  $W$  can be represented by an  $n \times n$  matrix  $W = (w_{ij})$ , where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ W((i, j)) & \text{if } (i, j) \in E \\ \infty & \text{if } (i, j) \notin E \end{cases}$$

The all-pairs shortest paths problem is to find the paths of the shortest distance for all pairs of nodes  $(i, j)$ ,  $1 \leq i, j \leq n$ .

```

Initialize  $D^{(0)}$  with distance matrix  $W = (w_{ij})$ 
for  $k = 1$  to  $n$  do
  for  $i = 1$  to  $n$  do
    for  $j = 1$  to  $n$  do
       $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)});$ 
return  $D^{(n)}$ 

```

Fig. 1. Floyd-Warshall Algorithm

```

Initialize  $D$  with distance matrix  $W = (w_{ij})$ 
for  $k = 1$  to  $n$  do
  for  $i = 1$  to  $n$  do
    for  $j = 1$  to  $n$  do
       $d_{ij} = \min(d_{ij}, d_{ik} + d_{kj});$ 
return  $D$ 

```

Fig. 2. Equivalent Floyd-Warshall Algorithm

Floyd-Warshall algorithm for the all-pairs shortest-path problem [11] is shown in Figure 1.  $D^{(k)} = (d_{ij}^{(k)})$  in Figure 1 is the matrix of shortest path distances from  $i$  to  $j$  through nodes in  $\{1, \dots, k\}$ . The shortest distance from  $i$  to  $j$  is in  $d_{ij}^{(n)}$ , when the triple-nested loop is completed. Since  $D^{(k)}$  is calculated by using only  $D^{(k-1)}$  and  $D^{(k-1)}$  is only used to calculate  $D^{(k)}$ , we can use two matrices: one to store  $D^{(0)}, D^{(2)}, \dots$  and the other to store  $D^{(1)}, D^{(3)}, \dots$ . However, further investigation shows that we can use one matrix instead of two for this algorithm. It can be proved that  $d_{kk}^{(1)} = d_{kk}^{(2)} = \dots = d_{kk}^{(n)} = 0$  for any  $k$ ,  $d_{ik}^{(k)} = d_{ik}^{(k-1)}$  for any  $i \neq k$ , and  $d_{kj}^{(k)} = d_{kj}^{(k-1)}$  for any  $j \neq k$ .<sup>1</sup> In other words, we can replace  $d_{ik}^{(k-1)}$  with  $d_{ik}^{(k)}$  and  $d_{kj}^{(k-1)}$  with  $d_{kj}^{(k)}$ , in the right-hand side,  $\min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ , of the assignment. As a result, we can use one matrix  $D = (d_{ij})$  instead of two to store all  $D^{(0)}, \dots, D^{(n)}$ . The equivalent algorithm using one matrix  $D = (d_{ij})$  is shown in Figure 2.

Let the original  $n \times n$  matrix  $D$  be blocked into a  $d \times d$  matrix of smaller matrices  $A = (A_{ij})$  where each  $A_{ij}$  ( $1 \leq i, j \leq d$ ) is a  $b \times b$  matrix with  $b = n/d$  (we assume  $d$  divides  $n$ ). We reformat the blocked Floyd-Warshall algorithm in [9] and express it rigorously in Figure 3.

Function  $floyd(C, A, B)$  in Figure 3 updates matrix  $C$  using the matrices  $A, B$  and  $C$ , where  $C, A$  and  $B$  need not be different and all of them are of size  $b \times b$ . It updates  $C$   $b$  times through the outermost loop  $k$ , using the equation taken from the original Floyd-Warshall algorithm in Figure 2. The triple-nested loop in the blocked Floyd-Warshall algorithm updates the blocked matrix (the matrix of matrices)  $A = (A_{ij})$   $d$  times, where  $d = n/b$ . Each time, every block  $A_{ij}$  ( $1 \leq i, j \leq d$ ), is updated once by calling function  $floyd()$ . Figure 4 shows how the blocks are updated in order. First, the block  $A_{kk}$

<sup>1</sup>Note that  $d_{kk}^{(0)} = 0$  for all  $1 \leq k \leq n$ , and we have  $d_{kk}^{(1)} = 0$  for all  $1 \leq k \leq n$  because  $d_{kk}^{(1)} = \min(d_{kk}^{(0)}, d_{k1}^{(0)} + d_{1k}^{(0)}) = d_{kk}^{(0)} = 0$ . Similarly, we have  $d_{kk}^{(2)} = d_{kk}^{(3)} = \dots = d_{kk}^{(n)} = 0$  for all  $1 \leq k \leq n$ . Thus, we have  $d_{kk}^{(k)} = \min(d_{kk}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) = d_{kk}^{(k-1)}$  for any  $i \neq k$ , because  $d_{kk}^{(k-1)} = 0$ . Similarly, we have  $d_{kj}^{(k)} = \min(d_{kj}^{(k-1)}, d_{kk}^{(k-1)} + d_{kj}^{(k-1)}) = d_{kj}^{(k-1)}$  for any  $j \neq k$ .

```

function  $floyd(C, A, B)$  {
  Let  $C, A$  and  $B$  be  $b \times b$  matrices
   $(c_{ij}), (a_{ij})$  and  $(b_{ij})$ , respectively;
  for  $k = 1, b$ 
    for  $j = 1, b$ 
      for  $i = 1, b$ 
         $c_{ij} = \min(c_{ij}, a_{ik} + b_{kj});$ 
}

```

```

Initialize blocked matrix  $A = (A_{ij})$  with distance matrix  $W$ ;
for  $k = 1, d$  {
   $floyd(A_{kk}, A_{kk}, A_{kk});$ 
  for  $j = 1, d \wedge j \neq k$ 
     $floyd(A_{kj}, A_{kk}, A_{kj});$ 
  for  $i = 1, d \wedge i \neq k$  {
     $floyd(A_{ik}, A_{ik}, A_{kk});$ 
    for  $j = 1, d \wedge j \neq k$ 
       $floyd(A_{ij}, A_{ik}, A_{kj});$ 
  }
}

```

Fig. 3. Blocked Floyd-Warshall Algorithm

	j=1	2	3	4	5	6
i=1						
2						
3			Akk		Akj	
4						
5			Aik		Aij	
6						

Fig. 4. Updating different blocks

in the un-shaded area is updated using  $A_{kk}$  itself by calling  $floyd(A_{kk}, A_{kk}, A_{kk})$ . Then, blocks  $A_{kj}$  with  $j \neq k$  and blocks  $A_{ik}$  with  $i \neq k$  in the dark-shaded areas, are updated using the newly-updated  $A_{kk}$ , and  $A_{kj}$  and  $A_{ik}$  itself, by calling  $floyd(A_{kj}, A_{kk}, A_{kj})$  and  $floyd(A_{ik}, A_{ik}, A_{kk})$ , respectively. Finally, blocks  $A_{ij}$  with  $i \neq k$  and  $j \neq k$  in the light-shaded area are updated using the newly updated  $A_{ik}$  and  $A_{kj}$  by calling  $floyd(A_{ij}, A_{ik}, A_{kj})$ .

Note that each element of matrix  $C$  is updated  $b$  times in function call  $floyd(C, A, B)$ . Thus, each element of each block is thus updated  $d \cdot b = n$  times. Let the blocked matrix  $A = (A_{ij})$  be represented by an  $n \times n$  matrix  $\mathcal{A}$  and  $\mathcal{A}(i, j)$  be the  $ij$  element of  $\mathcal{A}$  so that we have

$$A_{ij}(i', j') = \mathcal{A}((i-1)b + i', (j-1)b + j')$$

Let each new value of  $A_{ij}(i', j')$  or  $\mathcal{A}((i-1)b + i', (j-1)b + j')$  be denoted by  $\mathcal{A}^{(k)}((i-1)b + i', (j-1)b + j')$ ,  $k = 1, \dots, n$ . While the blocked algorithm in Figure 3 does not guarantee  $\mathcal{A}^{(k)}(i, j)$  to be equal to  $D_{ij}^{(k)}$  in the original Floyd-Warshall algorithm (Figure 1) for all  $k$ , Venkataraman et al. did prove in [9] that  $\mathcal{A}^{(k)}(i, j) = D_{ij}^{(k)}$  for all the  $k$  that are multiples of  $b$ , including  $n = db$ , i.e.

$$\mathcal{A}^{(k)}(i, j) = D_{ij}^{(k)}, k = 0, b, 2b, \dots, db$$

### III. RAPID DEVELOPMENT OF PARALLEL MULTI-CORE CODES

To develop the parallel multi-core code for the blocked Floyd-Warshall algorithm in Figure 3, we could use **parallel for** loops provided by TBB to parallelize the algorithm as shown in Figure 5. However, **parallel for** loops incur barrier

```
Initialize blocked matrix  $A = (A_{ij})$  with distance matrix  $W$ 
for  $k = 1, d$  {
    floyd( $A_{kk}, A_{kk}, A_{kk}$ );
    parallel for  $j = 1, d \wedge j \neq k$ 
        floyd( $A_{kj}, A_{kk}, A_{kj}$ );
    parallel for  $i = 1, d \wedge i \neq k$  {
        floyd( $A_{ik}, A_{ik}, A_{kk}$ );
        parallel for  $j = 1, d \wedge j \neq k$ 
            floyd( $A_{ij}, A_{ik}, A_{kj}$ );
    }
}
```

Fig. 5. Parallel Blocked Floyd-Warshall Algorithm

synchronization overhead. A more efficient approach is to create a task graph where each node is a computational task and edges are data dependencies between the tasks, and execute the task graph in the order determined only by the data dependencies. While TBB does have mechanism to create task graphs, programmers still need to analyze and find all the flow, anti and output data dependencies between all the tasks and this is not an easy job for non-trivial algorithms. We recently de-

```
void floyd(mat& c, mat& a, mat& b) {
    int n = c.n_rows;
    for (int k = 0; k < n; k++)
        for (int j = 0; j < n; j++)
            for (int i = 0; i < n; i++)
                c(i, j) = min(c(i, j), a(i, k) + b(k, j));
}

void floyd_warshall_B_seq(field<mat>& A) {
    int dim = A.n_rows;
    for (int k=0; k<dim; k++) {
        floyd(A(k,k), A(k,k), A(k,k));
        for (int j=0; j<dim; j++) {
            if (j==k) continue;
            floyd(A(k,j), A(k,k), A(k,j));
        }
        for (int i=0; i<dim; i++) {
            if (i==k) continue;
            floyd(A(i,k), A(i,k), A(k,k));
            for (int j=0; j<dim; j++) {
                if (j==k) continue;
                floyd(A(i,j), A(i,k), A(k,j));
            }
        }
    }
}
```

Fig. 6. Blocked Floyd-Warshall Implementation using Armadillo

veloped the C++ library `gae.h` (Graph-driven Asynchronous Execution) which can analyze data dependencies between the tasks and build the task graph in TBB automatically for dense linear algebra algorithms [4]. Developing the TBB task-graph

based parallel code using our library is straightforward and fast.

To ease programming, we also use Armadillo, a linear algebra C++ library for computationally intensive applications. Armadillo supports MATLAB-like matrix expressions and blocked matrices. A blocked matrix (i.e., a matrix of matrices) can be defined as a *field* of matrices in Armadillo. In Armadillo, `mat` is of type double matrix and `field<mat>` is of type matrix of double matrices. Figure 6 is the implementation of the blocked Floyd-Warshall algorithm in Figure 3 using Armadillo.

A block (`mat`) in a blocked matrix (`field<mat>`) is fully defined by its index, a pair of integers. In `gae.h`, class `tile` is the type for such blocks. The definition of class `tile` is as follows:

```
class tile {
public:
    field<mat>* D;
    t_index ti;

    tile(field<mat>* D_, t_index ti_) :
        D(D_), ti(ti_) {}
}
```

It is simply a wrapper of the tile index and a pointer to the blocked matrix.

Class `t_index` is the type for a tile index, which is a wrapper of two integers:

```
class t_index {
public:
    int m_i, m_j;

    t_index(int i, int j) :
        m_i(i), m_j(j) {}
}
```

A set of tiles is of type `tile_set`, defined as a vector of tiles: `typedef vector<tile> tile_set`.

A task in `gae.h`, is a function object overloading operator `()` with its write tile set and read tile set: `operator()(tile_set* writeset, tile_set* readset)`. The write and read tile sets are the sets of tiles that the task is going to write and read, respectively. The tiles in the write(read) tile sets have to be distinct.

For the blocked Floyd-Marshall algorithm, we have two types of tasks that wrap the function call of `floyd(mat, mat, mat)`. For the function call `floyd(A(k,k), A(k,k), A(k,k))` in Figure 6, we have only one write tile and one read tile, both are `A(k,k)`. We create a task type for these calls called `task_kk`. For other function calls, we have one write tile and two read tiles. For example, function call `floyd(A(k,j), A(k,k), A(k,j))` has one write tile `A(k,j)` and two read tiles `A(k,k)`, `A(k,j)` in order. We create another task type for these function calls called `task_others`.

The correct contents of write and read sets of tasks are important as they will be used to analyze all the data dependencies between the tasks by library `gae.h`. The definition of

```

class floyd_task_kk{
public:
    void operator() (tile_set* writeset,
                    tile_set* readset) {
        field<mat>& W = *((*writeset)[0].D);
        t_index w = (*writeset)[0].ti;
        field<mat>& R = *((*readset)[0].D);
        t_index r = (*readset)[0].ti;

        floyd(W(w.m_i,w.m_j), R(r.m_i,r.m_j),
              R(r.m_i,r.m_j));
    }
};

class floyd_task_others{
public:
    void operator() (tile_set* writeset,
                    tile_set* readset) {
        field<mat>& W = *((*writeset)[0].D);
        t_index w = (*writeset)[0].ti;
        field<mat>& R0 = *((*readset)[0].D);
        t_index r0 = (*readset)[0].ti;
        field<mat>& R1 = *((*readset)[1].D);
        t_index r1 = (*readset)[1].ti;

        floyd(W(w.m_i,w.m_j), R0(r0.m_i,r0.m_j),
              R1(r1.m_i,r1.m_j));
    }
};

```

Fig. 7. Function objects for floyd tasks

the two task types for the blocked Floyd-Warshall algorithm is shown in Figure 7. The `operator()` extracts all the write and read tiles from its arguments and then calls function `floyd()` with the right tiles.

All that is necessary to develop parallel multi-core code using `gae.h` is the following:

- 1) Create a task graph of type `t_graph` defined as

```

class t_graph {
public:
    int k_dim, i_dim, j_dim;
    t_graph(vector<field<mat>*> D_,
            int n_write_, int k_dim_,
            int i_dim_, int j_dim_)
private:
    ...
}

```

The first argument of the constructor is the vector of pointers to the blocked matrices used in the algorithm. The first `n_write_` blocked matrices in the vector are write-read or write-only matrices and the rest are read-only matrices. `k_dim_`, `i_dim_` and `j_dim_` are the three dimensions of the 3D task graph.
- 2) Register all the tasks using the original serial blocked algorithm modified as follows: instead of executing the code of the task, we register the task which wraps the code with its write and read tile sets, by invoking template function `reg_task()` provided by `gae.h` as follows:

```

template <class Body>
void reg_task(t_graph *g, task_index ti,
              Body body,

```

```

tile_set* writeset,
tile_set* readset)

```

with the task index `ti`, the task function object `body` of type `Body` and the pointers to the write and read tile sets of the task.

- 3) Invoke the function

```
void exec_graph(t_graph* g)
```

provided by `gae.h` to complete and execute the graph in the order of the data dependencies.

The code to register the tasks for parallel blocked Floyd-Warshall algorithm is shown in Figure 8. All it does is to replace the computations (i.e. function calls to `floyd()`) in the serial blocked Floyd-Warshall code in Figure 6 with the code to create the write and read tile sets and register the tasks by passing task indexes, task objects and the write and read tile sets to function call `reg_task()`. Finally, the parallel code for blocked Floyd-Warshall algorithm is shown in Figure 9.

#### IV. PERFORMANCE EVALUATION

We evaluate the performance of both the serial and parallel Floyd-Warshall algorithms on an 80-core HP ProLiant DL980 G7 Server with 4 terabytes of shared memory.

We first evaluate the speedup of the blocked serial Floyd-Warshall algorithm in Figure 6 for different block sizes, over the unblocked algorithm in Figure 2. Figure 10(a) shows that for the problem size of 4800, we see a speedup more than 2 for all the block sizes. The speedups for problem sizes of 2800, 3840 and 5760 are lower than that of 4800. For a problem size 960, the speedup is close to 1. For all the problem sizes except 5760, the best block size is 32. For a problem size of 5760, the best block size is 48.

However, a small block size means more tasks and less computation in each task. Figure 10(b) shows the execution time of the parallel blocked code in Figure 9 using 80 cores. It shows that a block size of 32 increases the execution time dramatically. The execution times do not differ much for block sizes of 64, 80 and 96. Therefore, a block size of 64 was chosen for a performance evaluation of speedup and efficiency.

Figure 11(a) shows the execution times of the parallel code in Figure 9 as well as the serial blocked code in Figure 6, for different problem sizes, but with fixed block size 64. The numerical labels on the x-axis are the number of cores used in the parallel execution. The label "Serial" is for the execution time of the serial blocked code in Figure 6.

Figure 11(b) shows the speedup of parallel execution time over the serial execution time defined as

$$S_p = \frac{T_{serial}}{T_p} \quad (1)$$

where  $T_{serial}$  and  $T_p$  are the execution times of serial blocked code and parallel code using  $p$  cores, respectively. The block size is fixed at 64. For large problem sizes of 6144, 5120 and 4096, the speedup increases steadily as the number of cores used increases. The speedup of using 80 cores for problem sizes of 6144, 5120 and 4096 are 54.8, 53.8 and 51.4, respectively.

```

void floyd_warshallReg(t_graph *g,
                      field<mat>& F) {
    int kdim = g->k_dim;
    int idim = g->i_dim;
    int jdim = g->j_dim;

    for (int k=0; k<kdim; k++) {
        floyd_task_kk t_kk;
        tile w[] = {tile(&F, t_index(k,k))};
        tile_set* ws = new tile_set(w,
                                     w + sizeof(w)/sizeof(tile));
        tile r[] = {tile(&F, t_index(k,k))};
        tile_set* rs = new tile_set(r,
                                     r + sizeof(r)/sizeof(tile));
        reg_task(g, task_index(k,k,k),
                 t_kk, ws, rs);

        for (int j = 0; j < jdim; j++) {
            if (j == k) continue;
            floyd_task_others t_kj;
            tile w[] = {tile(&F, t_index(k,j))};
            tile_set* ws = new tile_set(w,
                                         w + sizeof(w)/sizeof(tile));
            tile r[] = {tile(&F, t_index(k,k)),
                       tile(&F, t_index(k,j))};
            tile_set* rs = new tile_set(r,
                                         r + sizeof(r)/sizeof(tile));
            reg_task(g, task_index(k,k,j),
                     t_kj, ws, rs);
        }

        for (int i = 0; i < idim; i++) {
            if (i == k) continue;
            floyd_task_others t_ik;
            tile w[] = {tile(&F, t_index(i,k))};
            tile_set* ws = new tile_set(w,
                                         w + sizeof(w)/sizeof(tile));
            tile r[] = {tile(&F, t_index(i,k)),
                       tile(&F, t_index(k,k))};
            tile_set* rs = new tile_set(r,
                                         r + sizeof(r)/sizeof(tile));
            reg_task(g, task_index(k,i,k),
                     t_ik, ws, rs);

            for (int j = 0; j < idim; j++) {
                if (j == k) continue;
                floyd_task_others t_ij;
                tile w[] = {tile(&F, t_index(i,j))};
                tile_set* ws = new tile_set(w,
                                             w + sizeof(w)/sizeof(tile));
                tile r[] = {tile(&F, t_index(i,k)),
                           tile(&F, t_index(k,j)),
                           tile(&F, t_index(i,j))};
                tile_set* rs = new tile_set(r,
                                             r + sizeof(r)/sizeof(tile));
                reg_task(g, task_index(k,i,j),
                         t_ij, ws, rs);
            }
        }
    }
}

```

Fig. 8. Registering tasks for blocked Floyd-Warshall Algorithm

```

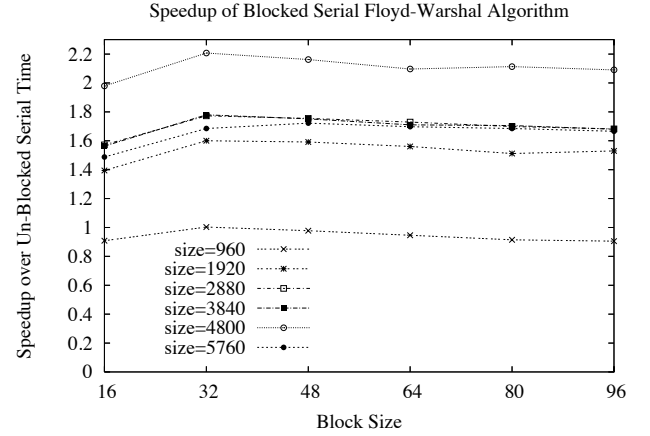
void floyd_warshall_B_tbb(field<mat>& D) {
    int idim = D.n_rows;
    int jdim = D.n_cols;
    int kdim = idim;

    vector<field<mat>*> dset;
    dset.push_back(&D);
    t_graph* g = new t_graph(dset, 1,
                             kdim, idim, jdim);

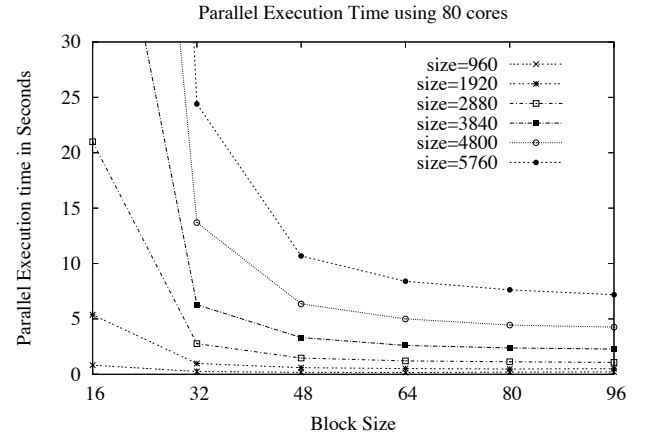
    floyd_warshallReg(g, D);
    exec_graph(g);
}

```

Fig. 9. Parallel blocked Floyd-Warshall code



(a) Speedup of serial blocked over serial unblocked code



(b) Parallel execution time of 80 cores

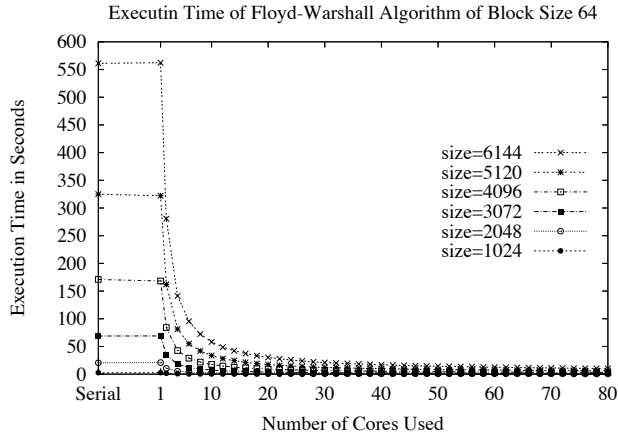
Fig. 10. Impact of Block Size

Figure 11(c) show the efficiency of parallel execution defined as

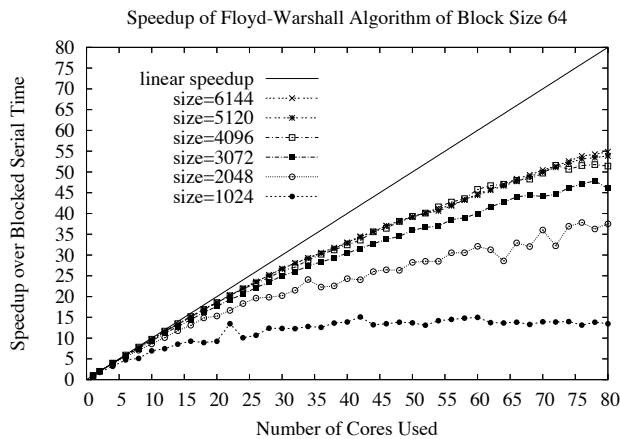
$$E_p = \frac{S_p}{p} \quad (2)$$

That is, the efficiency is the speedup  $S_p$  divided by the number of cores  $p$ . Figure 11(c) shows that the efficiencies for large problem sizes 6144, 5120, 4096 are above 70 % if less than 70 cores are used. Their efficiencies are above 90% if less than 20 cores are used. They decrease almost linearly as the number of cores used increases. When 80 cores (the full capacity of the machine) are used, the efficiencies for problem sizes 6144,

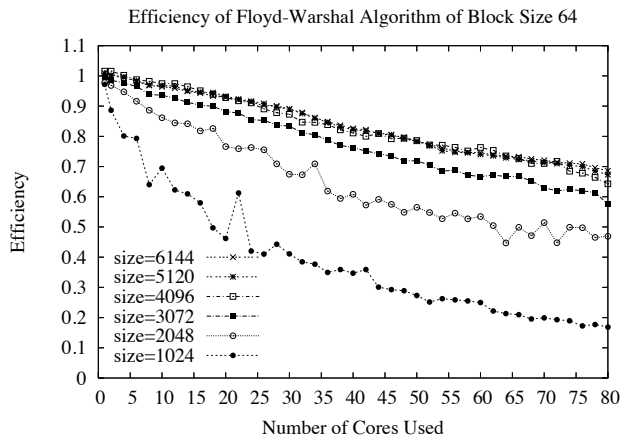
5120 and 4096, are 68.6%, 67.3% and 64.2%, respectively. The efficiency starts to degrade when the problem size drops below 4096 with a fixed block size of 64, because parallelism is reduced.



(a) Execution Times



(b) Speedup over serial blocked code



(c) Efficiency of Parallel Code

Fig. 11. Performance of Parallel Floyd-Warshall Code

## V. RELATED WORK

There have been past works on parallelizing the blocked all-pairs shortest paths Floyd-Warshall algorithm [9]. Bondhugula et al. parallelized the blocked Floyd-Warshall algorithm for Field Programmable Gate Array (FPGA) machines [12]. Matsumoto and Sedukhin proposed a parallel implementation for the Cell Broadband Engine processor [13]. Katz and Kider parallelized it for GPU based computation [14]. A parallelization for hybrid CPU-GPU was presented in [15]. All these works are parallelizing the blocked Floyd-Warshall algorithm for special parallel machines. Programming these parallel machines requires significant efforts. None of the aforementioned research addresses easing programming burden. Parallelizing the unblocked (original) Floyd-Warshall algorithm for shared-memory machines using OpenMP3.0 and TBB were reported in [16] and [17], respectively.

## VI. CONCLUSION

In this paper, we show the ease of programming with `gae.h`, a C++ library on top of Intel C++ Threading Building Blocks (TBB), for the Floyd-Warshall algorithm. The parallel code developed is also efficient. The efficiency of our code on an 80-core HP ProLiant DL980 G7 machine is above 70% if less than 70 cores are used and above 90% if less than 20 cores are used.

## REFERENCES

- [1] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. Oreilly Associates, 2007.
- [2] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurr. Comput. : Pract. Exper.*, 22(1):15–44, January 2010.
- [3] Michael McCool, Arch D. Robinson, and James Reinders. *Structured Parallel Programming – Patterns for Efficient Computation*. Morgan Kaufmann, Elsevier Inc., 2012.
- [4] Peiyi Tang. A C++ library for rapid development of efficient parallel dense linear algebra codes for multicore computers. In *Proceedings of the 51-st Annual Association for Computing Machinery Southeast Conference (ACMSE'13)*, pages 10:1–10:6, Savannah, GA, USA, April 2013.
- [5] Margreet Louter-Nool. Block-cholesky for parallel processing. *Applied Numerical Mathematics*, 10:37–57, 1992.
- [6] Paolo Bientinesi, Brian Gunter, and Robert A. van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Softw.*, 35(1):3:1–3:22, July 2008.
- [7] Ling Shang, Serge Petiton, and Maxime Hugues. A new parallel paradigm for block-based gauss-jordan algorithm. In *Proceedings of the 2009 International Conference on Grid and Cooperative Computing*, pages 193–200, 2009.
- [8] Neungsoo Park, Bo Hong, and Viktor K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Trans. Parallel Distrib. Syst.*, 14:640–654, July 2003.
- [9] Gayathri Venkataraman, Sartaj Sahni, and Srabani Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. *Journal of Experimental Algorithmics*, 8, December 2003.
- [10] Conrad Sanderson. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, NICTA, 2010.
- [11] Thomas H. Corman, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introductions to Algorithms, 3rd Edition*. MIT Press, 2009.
- [12] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan. Parallel fpga-based all-pairs shortest-paths in a directed graph. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10 pp.–, 2006.

- [13] Kazuya Matsumoto and Stanislav G. Sedukhin. A solution of the all-pairs shortest paths problem on the cell broadband engine processor. *IEICE Transactions*, E92-D(6):1225–1231, June 2009.
- [14] Gary J. Katz and Joseph T. Kider, Jr. All-pairs shortest-paths for large graphs on the gpu. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '08, pages 47–55, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.
- [15] K. Matsumoto, N. Nakasato, and S.G. Sedukhin. Blocked all-pairs shortest paths algorithm for hybrid cpu-gpu system. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 145–152, 2011.
- [16] Eid Albalawi, Parimala Thulasiraman, and Ruppan Thulasiram. Task level parallelization of all pair shortest path algorithm in OpenMP 3.0. In *The 2nd International Conference on Advances in Computer Science and Engineering (CSE 2013)*, pages 109–113. Atlantis Press, 2013.
- [17] Jian Ma, Ke ping Li, and Li yan Zhang. A parallel floyd-warshall algorithm based on tbb. In *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*, pages 429–433, 2010.