ABSTRACT

To program parallel codes for task-based parallel execution, the full data dependency analysis between the tasks is required. However, finding all the flow, anti and output data dependencies is not an easy task for non-trial algorithms. In this paper, we present a simple C++ library that can analyze all the data dependencies between tasks and build the task graph automatically. Developing parallel dense linear algebra codes using our library and another two C++ libraries, Intel TBB and NICTA Armadillo, is simple and easy. The parallel codes developed by using our library are also efficient due to the efficient task scheduling of Intel TBB library and the fast matrix operations of NICTA Armadillo library.

Keywords

Graph-driven asynchronous execution; Task-based parallel execution; Rapid parallel programming; Automatic data dependency analysis; Intel TBB; NICTA Armadillo

1. INTRODUCTION

Task-based parallel codes based on the graph-driven asynchronous execution model is more efficient than loop-based parallel codes in general, because there are no extra unnecessary dependencies introduced between computation tasks. Another reason is that breaking the computation into tasks increases cache data locality. In fact, many have long been advocating blocked algorithms which view a matrix as a collections of submatrices and express the algorithms in term of operations on these submatrices. The blocked algorithms, also known as algorithms-by-blocks, fit the graph-driven asynchronous execution model nicely: the computation of a block (submatrix) using other blocks forms a task naturally.

To program in the graph-driven asynchronous execution model, however, requires analysis of all the flow (RAW), anti (WAR) and output (WAW) data dependencies between the tasks. To find these data dependencies is not an easy job except for trivial cases. For example, the analysis of data dependencies for Gauss-Jordan elimination can be very complicated [1]. There are systems of graph-driven asynchronous execution that find and satisfy data dependencies automatically such as SMPSs [2, 3] and SuperMatrix [4]. SuperMatrix uses FLAME functions in C [5] which have large number of arguments. SMPSs is a compiler to compile C programs with OpenMP-like directives for identifying tasks. The task graph is generated dynamically at run-time [2]. For parallelizing algorithms-by-blocks, it still needs to be used with complicated LAPACK or FLAME codes with large number of arguments [3]. In this paper, we present a simple C++ library built on top of another two publicly available C++ libraries, Intel Threading Building Blocks (TBB) [6, 7] and NICTA Armadillo [8], to analyze data dependencies and build the static task graph automatically for blocked dense linear algebra algorithms. Parallelizing serial blocked dense linear algebra algorithms using our library (and TBB and Armadillo) is much easier than SuperMatrix. It does not need new languages and compilers such as SMPSs either, because parallel codes using our library can be developed directly in C++ and compiled by any C++ compiler.

The experimental results show that the parallel codes developed with our library are very efficient and the overhead of automatic data dependency analysis is also very small: around 1% of the total 1-core execution time.

The rest of the paper is organized as follows. Section 2 introduces the concepts of task graph and data dependency and describes our algorithm for building the task graph. Section 3 describes the API of our library and shows how to use it to develop parallel dense linear algebra codes. Section 4 presents the performance and efficiency of the parallel codes developed with our library. Section 5 describes the work related to ours. Section 6 concludes the paper.

2. BUILDING TASK GRAPHS FOR DENSE LINEAR ALGEBRA ALGORITHMS

2.1 Dense Linear Algebra Algorithms-by-Blocks

Robert van de Geijn et al have been long advocating a high-level representation of dense linear algebra algorithm without use of indices [9]. Algorithms can progress by single element or $b \times b$ block. For example, the in-place blocked Gauss-Jordan elimination algorithm [1] expressed in the Geijn’s high-level representation is shown in Figure 1.

In blocked dense linear algebra algorithms, the $n \times n$ ma-
matrix is viewed as a $d \times d$ matrix of smaller $b \times b$ matrices, where $n = d \times b$. Each of the small matrices is called a block or tile. The blocked algorithms, therefore, work directly on these blocks (tiles). In NICTA Armadillo, a $b \times b$ tile or block can be declared as a matrix of type mat with dimension $(b, b)$ and the $d \times d$ matrix of the blocks can be conveniently defined as a field of matrices of type field<mat> with dimension $(d, d)$.

for $k = 0, d - 1$
  $A_{kk} = A_{kk}^{-1}$
  for $j = 0, d - 1 \land j \neq k$
    $A_{kj} = A_{kj} A_{kj}$
  endfor
  for $i = 0, d - 1 \land i \neq k$
    $A_{ij} = A_{ij} - A_{ik} A_{kj}$
  endfor
endfor

for $i = 0, d - 1 \land i \neq k$
  $A_{ik} = -A_{ik} A_{kk}$
endfor

2.2 Building Task Graphs

From the high-level representation as shown in Figure 1 and the corresponding algorithms-by-blocks with indices as in Figure 2, we can derive the following common features of blocked dense linear algebra algorithms:

- There are 2D matrices of blocks swept and updated $d$ times, where $d = n/b$ and $b$ is the block size and $n$ the problem size. This determines the outermost loop with index $k$ to range from 0 to $d - 1$.
- In each iteration of the outermost $k$ loop, each block of the matrix is updated at most once.

A matrix of blocks can be $d \times d$ square matrix with $n = d \times b$ as in Gauss-Jordan eliminations in Figure 1, $p \times d$ rectangular matrix with $m = p \times b$ as in LU factorization (algorithm not shown), or $d \times p$ rectangular matrix as in triangular system solver with multiple right-hand sides (algorithm not shown).

Most algorithms update one matrix of blocks, some update more than one matrix such as $LU$ factorization with pivoting, which factorizes the original matrix $A$ to $L$ and $U$ and produces a permutation matrix $P$ such that $PA = LU$.

If we identify the computation of updating the block $(i, j)$ of one matrix at iteration $k$ of the outermost loop as task $(k, i, j)$, the tasks form a 3D structure and there are as many as $d \times d \times d \times d \times m$ or $d \times d \times p \times d$ tasks. Depending on the algorithm, we can have $0 \leq i \leq d - 1, 0 \leq j \leq p - 1$ or $0 \leq i \leq p - 1, 0 \leq j \leq d - 1$, but always have $0 \leq k \leq d - 1$.

Each task can read and write several blocks from several matrices, causing flow, anti and output data dependencies among the tasks. The task graph of a dense linear algebra
algorithm is a direct acyclic graph \( G = (V, E) \), where \( V \) is the set of tasks \((k, i, j)\), with \(0 \leq k \leq d - 1\) and \(0 \leq i \leq d - 1, 0 \leq j \leq p - 1\) or \(0 \leq i \leq p - 1, 0 \leq j \leq d - 1\) (\(p\) may equal to \(d\)) and \( E \subseteq V \times V \) is the set of edges such that \((u, v) \in E\) if and only task \(v\) is flow, anti or output data dependent on task \(u\). A task \(v\) is flow data dependent on task \(u\) if task \(v\) reads the value from a block of matrix that is written by task \(u\). A task \(v\) is anti data dependent on task \(u\), if tasks \(v\) overwrites the value of a block of matrix that is read by task \(u\). A task \(v\) is output data dependent on task \(u\), if tasks \(v\) overwrites the value that is written by task \(u\).

We can find all the data dependencies and build the task graph by tracing the tiles read and written by all the tasks in their original sequential order of the algorithm. The algorithm of finding data dependencies is described as follows.

For each tile, there is a data structure called tile-foot to record (1) the index of the task that writes the tile most recently (recorded in writer) and (2) the indices of the tasks that read the tile since the most recent task writing the tile (recorded in reader-set). The task graph is built by registering each task following the original sequential order of the algorithm-by-blocks. To register a task, we provide (1) the index of the task, (2) the set of indices of the tiles read by the task and (3) the set of the indices of the tiles written by the task, as well as the function object of the task. When registering a task, we do the following in order:

1. For each tile read by the task
   (a) Update the tile-foot of the tile by adding the index of the task to its reader-set.
   (b) Add a flow dependency edge from the task that writes the tile most recently (recorded in writer) of the tile foot) to this task.

2. For each tile written by the task
   (a) Add an anti dependency edge from each of the task reading the tile since the previous write (recorded in reader-set of the tile foot) to this task.
   (b) Add an output dependency edge from the task writing the tile most recently (recorded in writer of the tile-foot) to this task.
   (c) Update writer of the tile-foot with the index of this task and remove all the tile indices in reader-set of the tile-foot by making it empty.

3. RAPID DEVELOPMENT OF MULTICORE CODES

In this section, we describe the API of our C++ library called gae.h for developing parallel dense linear algebra codes.

A tile is a sub-matrix, and, therefore, is fully defined by the name of the matrix and the corresponding tile index; a pair of integers, used to address it. In this paper, tile and block are used interchangeably.

In gae.h, a tile index is an object of class t_index defined as follows:

class t_index {
  public:
    int m_i, m_j;
    t_index(int i, int j) : m_i(i), m_j(j) {}
  }

where \(m_i\) and \(m_j\) are the \((i, j)\) index of the tile. A tile is an object of class tile defined as follows:

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

where \(D\) and \(ti\) are the pointer to the matrix of tiles and the index of the tile, respectively. A set of tiles is of type tile_set, defined by typedef vector<tile> tile_set.

Each task is a function object overloading the \(\) operator with a write tile set and a read tile set as its two arguments: operator\(()\)(tile_set* writeset, tile_set* readset).

For example, in the blocked Gauss-Jordan elimination in Figure 2, there are four kinds of computations that can be wrapped to four types of task function objects:

1. the task to update \(A_{kk}\) with \(A_{kk} := A_{kk}^{-1}\)
2. the task to update \(A_{kj} (j \neq k)\) with \(A_{kj} := A_{kk}A_{kj}\).
3. the task to update \(A_{ij} (j \neq k\) and \(i \neq k)\) with \(A_{ij} := A_{ij} - A_{ik}A_{kj}\)
4. the task to update \(A_{ik} (i \neq k)\) with \(A_{ik} := -A_{ik}A_{kk}\).

For instance, the function object to update \(A_{kk}\) in Figure 2 named as inversion_task_kk, can be defined by the code in Figure 4. The function objects for other tasks, inversion_task_kj, inversion_task_ij, and inversion_task_ik, are defined similarly.

All these task objects do is to extract the write tile and the read tiles from the arguments, read the tiles, perform the computation according to the algorithm, and update the write tile with a new value.

To create the multicore parallel codes using gae.h, all you have to do is to:

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:
   ...
   }
   ```

   where D and ti are the pointer to the matrix of tiles and the index of the tile, respectively. A set of tiles is of type tile_set, defined by typedef vector<tile> tile_set.

   Each task is a function object overloading the \(\) operator with a write tile set and a read tile set as its two arguments: operator\(()\)(tile_set* writeset, tile_set* readset).

   For example, in the blocked Gauss-Jordan elimination in Figure 2, there are four kinds of computations that can be wrapped to four types of task function objects:

   1. the task to update \(A_{kk}\) with \(A_{kk} := A_{kk}^{-1}\)
   2. the task to update \(A_{kj} (j \neq k)\) with \(A_{kj} := A_{kk}A_{kj}\).
   3. the task to update \(A_{ij} (j \neq k\) and \(i \neq k)\) with \(A_{ij} := A_{ij} - A_{ik}A_{kj}\)
   4. the task to update \(A_{ik} (i \neq k)\) with \(A_{ik} := -A_{ik}A_{kk}\).

   For instance, the function object to update \(A_{kk}\) in Figure 2 named as inversion_task_kk, can be defined by the code in Figure 4. The function objects for other tasks, inversion_task_kj, inversion_task_ij, and inversion_task_ik, are defined similarly.

   All these task objects do is to extract the write tile and the read tiles from the arguments, read the tiles, perform the computation according to the algorithm, and update the write tile with a new value.

   To create the multicore parallel codes using gae.h, all you have to do is to:

   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:
   ...
   }
   ```

Figure 4: Function Object for task t_kk of Gauss-Jordan Elimination
void inversionReg(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++) {
        inversion_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;
            inversion_task_kj 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;
                inversion_task_ik 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);
            }
        }
    }
}

The first argument of the constructor is the vector of pointers to the matrices of tiles used in the algorithm. The first \texttt{n\_write} matrices in the vector are write-read or write-only matrices and the rest are read-only matrices. \texttt{k\_dim}, \texttt{i\_dim} and \texttt{j\_dim} are the three dimensions of the 3D task graph discussed in Section 2.2.

2. Register all the tasks using the original serial blocked algorithm modified as follows: Instead of executing the code of the task, we first form the write tile set and the read tile set of type \texttt{tile\_set} and register the task by calling the template function

\begin{verbatim}
template <class Body>
void reg_task(t_graph *g, task_index ti,
              Body body,
              tile_set* writeset,
              tile_set* readset)
\end{verbatim}

with the task index \texttt{ti}, the task function object \texttt{body} and the pointers to the write and read tile sets, \texttt{writeset} and \texttt{readset}, respectively.

The template function \texttt{reg\_task()} is provided by \texttt{gae.h}. What it does is to create a TBB task in the task graph and analyze and record the data dependencies caused by this task. Figures 5 shows the function to register the tasks of Gauss-Jordan elimination according to its serial blocked algorithm.

3. Invoke the function

\begin{verbatim}
void exec_graph(t_graph* g)
\end{verbatim}

provided by \texttt{gae.h} to complete and execute the graph in the order of the data dependencies.

Figure 6 shows the multicore parallel code implementing the three steps above for Gauss-Jordan elimination.

\begin{verbatim}
field<mat> inversion_tbb2(const field<mat>& D) {
    field<mat> E = D;
    int idim = D.n_rows;
    int jdim = D.n_cols;
    int kdim = idim;
    vector<field<mat>*> dset;
    dset.push_back(&E);
    t_graph* g = new t_graph(dset, 1, kdim, idim, jdim);
    inversionReg(g, E);
    exec_graph(g);
    return E;
}
\end{verbatim}

Figure 6: Multicore Parallel Code for Gauss-Jordan Elimination

In summary, to develop multicore parallel codes for dense linear algebra algorithms using \texttt{gae.h}, one only needs to (1) create task function object classes such as in Figure 4 to wrap the computation codes from the original algorithm, (2) covert the original serial blocked code as in Figure 3 to the one as shown Figure 5 to register all the tasks by using function \texttt{reg\_task()} provided by \texttt{gae.h}, and (3) call function \texttt{exec\_graph()} provided by \texttt{gae.h} to perform the asynchronous execution of the task graph.
4. PERFORMANCE EVALUATION

To evaluate the efficiency of parallel codes developed with gau.h library, we run the parallel codes for Cholesky factorization (code is not shown in this paper) and Gauss-Jordan elimination compiled with GNU g++ compiler on an 8-core AMD Opteron(tm) 2.8 GHz 8220 processor with 64 GByte memory. According to our early experiments with blocked

parallel execution times $T_p, p = 1, \ldots, 8$, in Figure 7(a). The execution time and the efficiency of Cholesky factorization are shown in Figures 8(a) and 8(b), respectively.

First of all, we observe that $T_{ser}$ is very close to $T_1$ for Gauss-Jordan elimination. In fact, $T_1$ is slightly less than $T_{ser}$ for problem sizes 512, 1024 and 2048, giving the efficiencies of 1.0029, 1.0061 and 1.0039, respectively. We attribute this above-1 efficiency (or super-linear speedup) to the superior cache locality tuned in the TBB scheduling. This also means that the overhead of TBB over the serial code is very small due to the efficient and low-cost scheduling of TBB.

As the number of cores increases, the efficiencies for all problem sizes decrease. This is because more cores join the work-stealing scheduling, resulting in large scheduling overhead and lower cache locality.

For problem size 6148, the efficiency of Gauss-Jordan elimination and Cholesky factorization using 8 cores is 0.9381 and 7.50, respectively.

Figure 7: Performance of Gauss-Jordan Elimination

Gauss-Jordan elimination [1], the optimal block (tile) size for this machine is 64. We have run both programs with block size 64 for problem sizes 512, 1024, 2048, 3072, 4096, 5120 and 6144. For each problem size, we run both the serial blocked code as in Figure 3 and the parallel code as in Figure 6, using 1, 2, 3, 4, 5, 6, 7 and 8 cores. We run each test 5 times and take the average of the execution times.

We calculate efficiency $E_p$ using $p$ cores as follows:

$$E_p = \frac{T_{ser}}{pT_p}$$

where $T_{ser}$ is the execution time of the serial blocked codes as in Figure 3, $T_p$ the execution of the parallel codes as in Figure 6 using $p$ cores. Note that $T_1$ is the parallel execution time using 1 core and $T_{ser}$ is the execution time of the serial code.

We plotted the execution time and the efficiency of Gauss-Jordan elimination in Figures 7(a) and 7(b), respectively. The serial execution time $T_{ser}$ is also plotted along with the
Table 1 shows this fraction for Cholesky factorization and Gauss-Jordan elimination with block size 64 and different problem sizes. The fraction increases as the problem size increases. This is because, with fixed block size, increasing problem size increases the size of task graph and, thus, the number of data dependencies. We notice that the fractions for Cholesky factorization are greater than those for Gauss-Jordan elimination. This is because the tasks of Cholesky factorization have less amount of computation than those of Gauss-Jordan elimination. At problem size 6144, the overhead of automatic data dependency analysis is under 0.85% in Gauss-Jordan elimination and 1.22% in Cholesky factorization.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cholesky</th>
<th>Gauss-Jordan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>0.00829</td>
<td>0.00635</td>
</tr>
<tr>
<td>2048</td>
<td>0.00833</td>
<td>0.00747</td>
</tr>
<tr>
<td>3072</td>
<td>0.01145</td>
<td>0.00814</td>
</tr>
<tr>
<td>4096</td>
<td>0.01104</td>
<td>0.00834</td>
</tr>
<tr>
<td>5120</td>
<td>0.01293</td>
<td>0.00857</td>
</tr>
<tr>
<td>6144</td>
<td>0.01213</td>
<td>0.00842</td>
</tr>
</tbody>
</table>

Table 1: Fraction of Analysis time in Total Time with Block Size 64

5. RELATED WORK

The idea of inspector-executor for parallelizing irregular applications can be traced back to the early work by Saltz et al. [10].

The work closest to ours includes the PLASMA extension to LAPACK and the SuperMatrix extension to FLAME [5], as well as SMPSSs [3]. All these works use their in-house dynamic data dependency analysis embedded in their multicore scheduling run-time systems. In particular, they find data dependencies by inspecting the memory footprint of tiles as the execution goes. Our work of gae.h takes advantage of the regular structure of dense linear algebra algorithms and builds static data dependence task graphs by simulating the original algorithm on the tile feet using task indexes instead of memory footprint. Lastly, by using another two C++ library: TBB and Armadillo, we can develop efficient multicore parallel codes of dense linear algebra algorithms directly in C++.

In terms of expressing and parallelizing algorithms-by-blocks, our work is also related to Hierarchical Tile Array (HTA), a C++ library for programming with recursive tiled algorithms [11]. However, the programming model of HTA is not graph-driven asynchronous execution but rather loop-based synchronous parallel execution. While HTA allows one to define multi-level tiled arrays, we found that two-level tiled arrays are sufficient for all the dense linear algebra algorithms. The field of matrices in Armadillo library used in our library gae.h is very easy to use and fast.

6. CONCLUSION

In this paper, we introduce and present a simple C++ library called gae.h using the Intel TBB and NICTA Armadillo libraries for rapid development of parallel dense linear algebra codes for multicore computers. Programming the parallel codes for blocked dense linear algebra algorithms using this library is simply to (1) create function objects for the tasks by wrapping the different computations, (2) register the tasks using the original blocked algorithm to build the static task graph and (3) invoke a function to execute and evaluate the task graph. This is a simple and fast process and can be accomplished with very few code changes from the original serial blocked codes.

The parallel codes generated by our library is also very efficient due to the facts that (1) the tasks are spawned and executed according to the static task graph created by registering them before hand and (2) the very efficient Intel TBB work-stealing task scheduling is used. The experimental data shows also that the overhead of building the static task graph is only a very small fraction (around 1%) of the total execution time using one core.

7. REFERENCES