國立臺灣大學理學院應用數學科學研究所

碩士論文

Institute of Applied Mathematical Sciences

College of Science

National Taiwan University

Master Thesis

以分級平行多層舒爾法於 CUDA 叢集解線性系統

Scalable Hierarchical Schur Linear System Solver with Multilevel Parallelism on CUDA Enabled Clusters

王柏川

Pochuan Wang

指導教授：王偉仲教授

Advisor: Weichung Wang, Ph.D.

中華民國 105 年 1 月

January, 2016

國立臺灣大學碩士學位論文
口試委員會審定書

以分級平行多層舒爾法於 CUDA 叢集解線性系統
Scalable Hierarchical Schur Linear System Solver with Multilevel Parallelism on CUDA Enabled Clusters

本論文係王柏川君 (R02246011) 在國立臺灣大學應用數學科學研究所完成之碩士學位論文，於民國 105 年 1 月 22 日承下列考試委員審查通過及口試及格，特此證明

口試委員：

____

________________　　________________

________________　　________________

________________　　________________

________________　　________________

所　長：________________________

摘要

解稀疏線性系統是科學計算領域中最核心的問題之一，隨著問題的規模增加，有效率的解稀疏線性系統是必要的。在現今的計算機架構中，計算核心的時脈由於物理限制而難以大幅提升，計算單元的設計傾向以多核心平行達到提升效能的目的。為了最大化計算資源的使用以達到更高的效率，演算法必須針對平行的架構設計演算法。多層舒爾法藉由分析以多重巢狀分割法重排後稀疏矩陣的分塊結構，將矩陣直接解法的分解過程拆解為可平行的子問題，各個子問題中再適當地以不同的方法平行加速。在此之上，經由分析各個子問題的所需計算量，再以優化過的機制將子問題均衡的分配到不同的處理器上，進而達到更高的可擴展性。

Abstract

Sparse linear system solver is one of the core of scientific computing. As the scale of problem increases, to solve sparse linear systems efficiently is necessary. In recent computer architecture, the frequency of computation core is bounded by physical limitations, thus current design of computatation unit as CPU and GPU use multiple cores to improve the performance. Hierarchical Schur method expolits the block structure of multilevel nested dissection reordered sparse linear system and decompose the direct matrix factorization scheme into concurrent subproblems. In each subproblems we properly applied different techniques for lower level parallelism. Moreover by analyzing the computation cost of each subproblems, it is able to distribute the computation load to different resources to improve overall scalability.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Solving sparse linear system $Ax = b$ is the core of many scientific computation problems, there problems were generated from various real world problems such as weather prediction, nano-structure optimization design, renewable energy, chemical kinetics, photoelectric materials and biomedicine. As the scale of problem increases, to solve the linear system efficiently becomes an important problem.

1.1 Sparse Linear Solvers

Most linear solvers can be classify to a iterative solver or a direct solver, each type of solver has its benefit and restriction.

Iterative Solver

Iterative solver is popular for sparse linear systems, iterative solvers are less memory consumptive and usually have less computational complexity compare to direct solvers. Conjugate gradient, biconjugate gradient stabilized method(BiCGStab) and generalized minimal residual method(GMRES) are widely used iterative solver algorithm. Iterative solver is not ensured to converge for ill-conditioned systems, in many case applying a properly designed preconditioner can fix this problem, however in many cases to design a preconditioner for specified problem is not easy.

Direct Solver

Direct sparse solver is ensured to produce accurate solutions. The drawbacks of direct solver is it need more memory and also have more computational complexity. Direct solver is more difficult to scale up in distributed memory system.

1.2 Related Works

Serveral excellent works has been done in the past, for iterative solvers, PETSc[2] and PARALUTION[6] are stable and high performance software, both are support GPU and distributed memory system. For sparse direct solvers, MUMPS[1] is a high performance multifrontal solver designed for multi-core CPU architecture and also have distributed memory system support by MPI. PARDISO[7] implements supernodal LU factorization for multi-core CPU architecture with many-core Intel Xeon Phi acceleration. Both MUMPS and PARDISO are capable to solve unsymmetric, symmetric indefinite and symmetric positive definite linear systems. Chenhan's solver[5] is a CPU-GPU hybrid multifrontal Cholesky solver optimize for CPU-GPU compututation load balancing. CHOLMOD[4] implements supernodal Cholesky algorithm, provides MATLAB interface and also support GPU acceleration. Another class of sparse solver combines direct factorization and iterative solvers, PDSLin[8] is based on Schur's complement method, the diagonal blocks are solved by direct method and Schur's complement is solved by iterative method.

1.3 Purpose

In this research we aim to develop a stable and high performance linear solver on a multi-node cluster with GPU accelerators. This work is focus on improving the scalability by studying the task scheduling and parallel implementation of hierarchical Schur method.

Chapter 2

Background

This chapter contains related algorithms applied in hierarchical Schur method, including nested dissection sparse matrix fill reducing ordering, block matrix decomposition and Schur's complement method.

2.1 Nested Dissection

The propose of nested dissection to reduce non zero fill in sparse matrix factorization. An input sparse matrix A is treated as an adjacency matrix of corresponding graph G. An example is shown in Figure 2.1. The index on each node of graph is the row and column index of A.

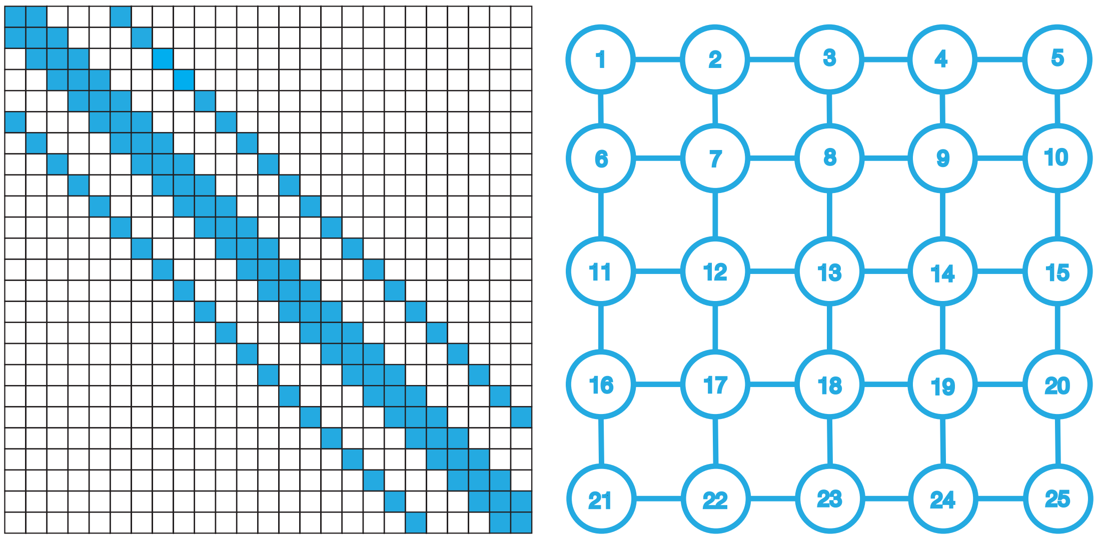


Figure 2.1: Sample sparse matrix and its corresponding graph.

Nested dissection scheme divide the graph into three parts, first two parts are called **subdomains**, the two subdomains has no edge connection. The third part is call a **separator**, which connected the two subdomains. Then re-indexing the original graph with subdomains first, separator last, the effect is shown in Figure 2.2.

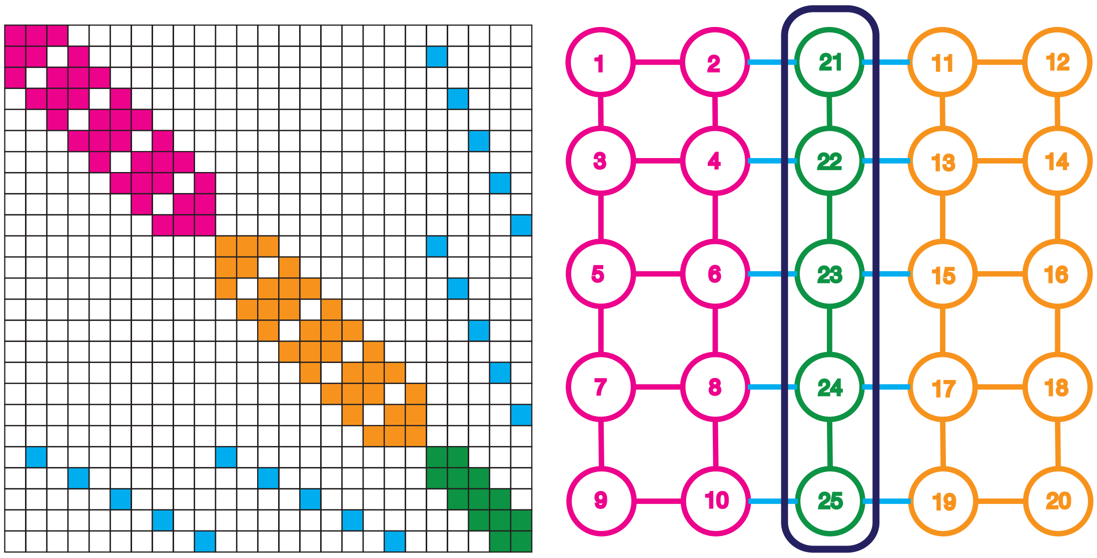


Figure 2.2: Matrix and graph after Nested Dissection.

Nested dissection can be applied on each subdomains, then new subdomains and separators will be generated. While applying k times recursively on each subdomains, the reordered matrix A_k can be represented as

$$A_k = \begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,2^{k+1}-1} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,2^{k+1}-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{2^{k+1}-1,1} & A_{2^{k+1}-1,2} & \cdots & A_{2^{k+1}-1,2^{k+1}-1} \end{bmatrix} \tag{2.1}$$

Definition 2.1. For $i = 1, 2, \ldots, 2^{k+1} - 2$, $k \geq 0$, a **parent** of i is $2^k + \lceil \frac{i}{2} \rceil$.

Definition 2.2. For $i, j \in 1, 2, \ldots, 2^{k+1} - 1$, j is a **ancestor** of i if j is reachable by from i by proceeding to its parent.

Definition 2.3. For $i, j \in 1, 2, \ldots, 2^{k+1} - 1$, j is a **descendant** of i if i is an ancestor of j.

Definition 2.4. For $i = 2^k + 1, 2^k + 2, \ldots, 2^{k+1} - 1$, $k \geq 0$, a **left child** is $2i - 2^{k+1}$, a **right child** is $2i - 2^{k+1} + 1$.

Theorem 2.5. *A block $A_{i,j}$ in A_k contains a non zero element if*

- $i = j$
- *i is an ancestor of j*
- *i is a descendant of j*

While $k = 3$, the sparse block pattern of nested dissection reordered matrix is in 2.3

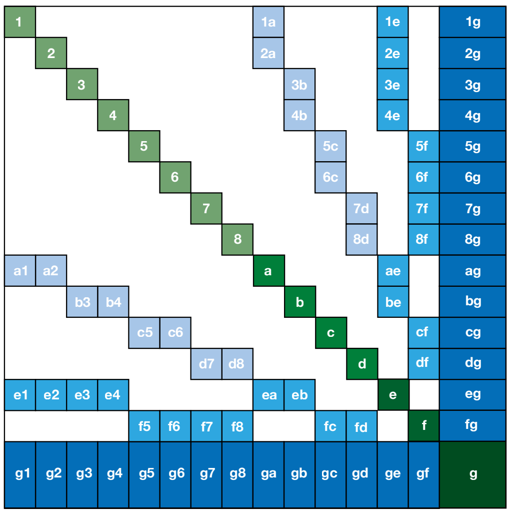


Figure 2.3: Sparse pattern of A_3 matrix

2.2 Schur's Complement Method

Schur's complement method is a technique to parallelize the computation of LU factorization for matrix B which has following block structure

$$B = \begin{bmatrix} B_{1,1} & 0 & B_{1,3} \\ 0 & B_{2,2} & B_{2,3} \\ B_{3,1} & B_{3,2} & B_{3,3} \end{bmatrix} \tag{2.2}$$

Assuming both $B_{1,1}$ and $B_{2,2}$ are full rank, the Schur's complement S_B is defined as

$$S_B = B_{3,3} - B_{3,1}B_{1,1}^{-1}B_{1,3} - B_{3,2}B_{2,2}^{-1}B_{2,3} \tag{2.3}$$

S_B is also full rank and there exist lower triangular matrix L_S and upper triangular matrix U_S such that $S_B = L_S U_S$. Then the LU Decomposition of B can be represented as

$$B = \begin{bmatrix} L_{1,1} & 0 & 0 \\ 0 & L_{2,2} & 0 \\ B_{3,1}U_{1,1}^{-1} & B_{3,2}U_{2,2}^{-1} & L_S \end{bmatrix} \begin{bmatrix} U_{1,1} & 0 & L_{1,1}^{-1}B_{1,3} \\ 0 & U_{2,2} & L_{2,2}^{-1}B_{2,3} \\ 0 & 0 & U_S \end{bmatrix} \tag{2.4}$$

$L_{1,1}, U_{1,1}$ and $L_{2,2}, U_{2,2}$ are LU decomposition of $B_{1,1}$ and $B_{2,2}$.

Schur's complement method makes LU decomposition scheme can be parallelized.

Chapter 3

Hierarchical Schur Algorithm

Hierarchical Schur algorithm is introduced in [3], this chapter provides a task based view point of both factorization, forward solving and backward solving.

3.1 HiS Algorithm

Hierarchical Schur method exploit the block structure of multilevel nested dissection re-ordered sparse matrix.

3.1.1 Factorization Algorithm

Algorithm 1 Hierarchical Schur Factorization Algorithm

```
Require: $A_k, k$
  for task $i = 1, 2, \ldots, 2^{k+1} - 1$ in parallel do
      if $i \leq 2^k$ then
          Compute $A_{i,i} = L_i U_i$ with sparse solver.
          Compute Schur update$A_{j,l} = A_{j,l} - A_{j,i} A_{i,i}^{-1} A_{i,l}$ for all ancestor $j, l$ of $i$
      else
          Wait for children to complete
          Compute $A_{i,i} = L_i U_i$ with dense solver
          if $i < 2^{k+1} - 1$ then
              Compute Schur update $A_{j,l} = A_{j,l} - A_{j,i} A_{i,i}^{-1} A_{i,l}$ for all ancestor $j, l$ of $i$
          end if
      end if
  end for
```

3.1.2 Solving Algorithm

After factorization, $L_{i,i}$ and $U_{i,i}$ are available. 2 show how to solve a right hand side b where

$$b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{2^{k+1}-1} \end{bmatrix} \tag{3.1}$$

Algorithm 2 Hierarchical Schur Solving Algorithm

```
Require: $L$ and $U$ such that $A_k = LU$, right hand side $b$
  Let $x = b$
  for $i = 2^{k+1} - 1, \cdots, 1$ do
      Compute $x_i = L_i^{-1} x_i$
      for all $j$, ancestor of $i$ do
          $x_j = x_j - A_{j,i} x_i$
      end for
  end for
  for $i = 1, 2, \ldots, 2^{k+1} - 1$ do
      for all $j$, ancestor of $i$ do
          $x_i = x_i - U_i^{-1} A_{i,j} x_j$
      end for
  end for
  return $x$
```

3.2 Example

Following is an example of factorization of 3 level nested dissection matrix A_3.

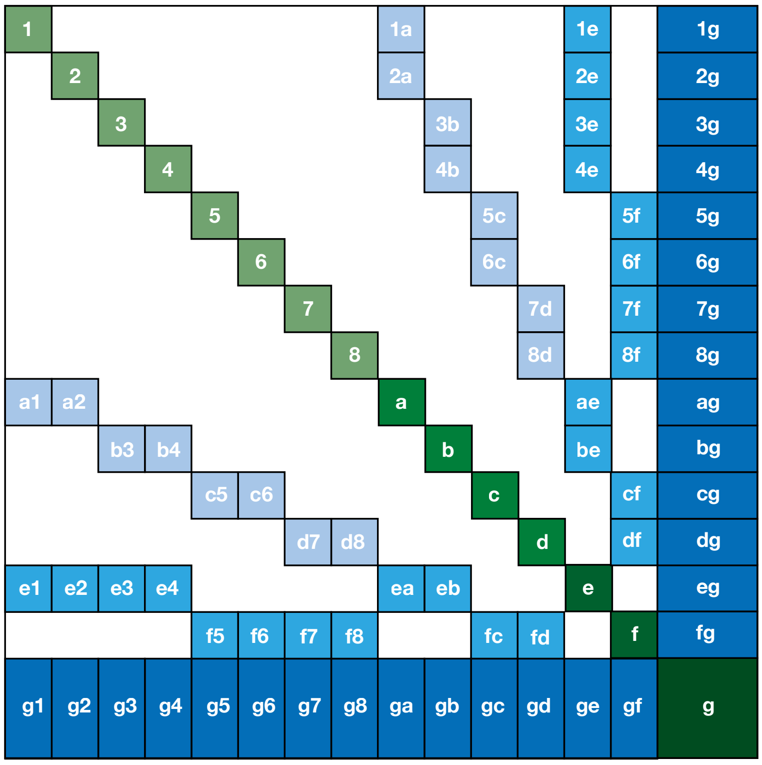


Figure 3.1: Sparse pattern of A_3 matrix

First, factorize the 8 independent diagonal blocks in parallel with a sparse direct solver.

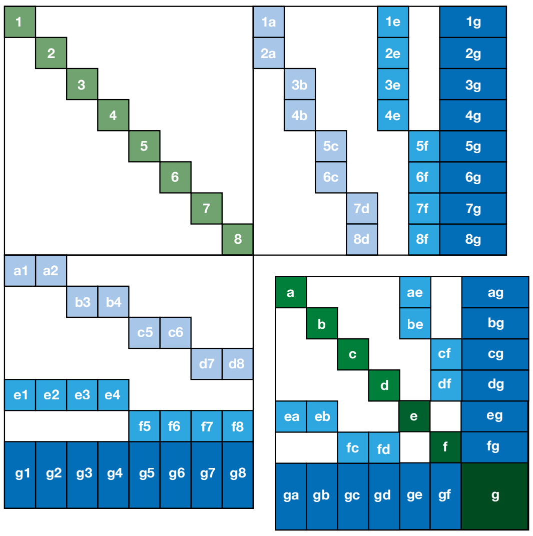


Figure 3.2: Blocks related to task 1-8

Compute Schur's update using previous factorization results and accumulate to corresponding position.

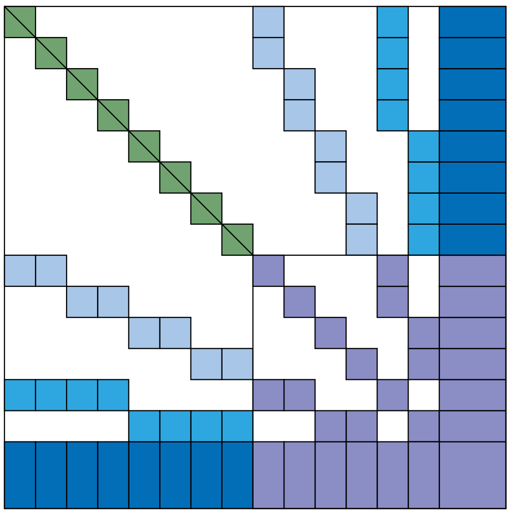
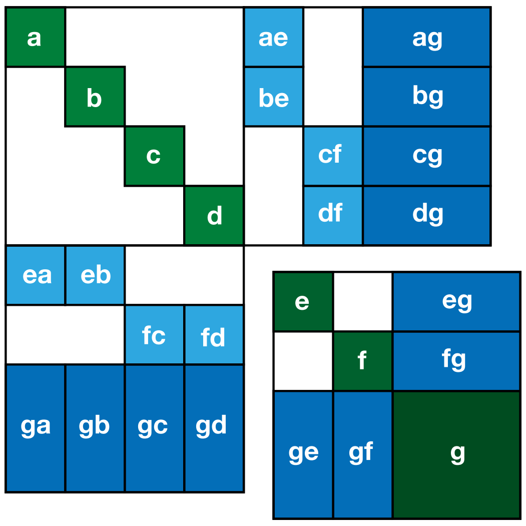
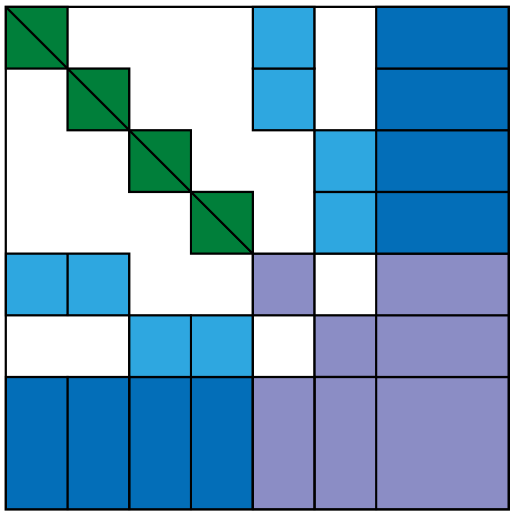


Figure 3.3: After task 1-8 done

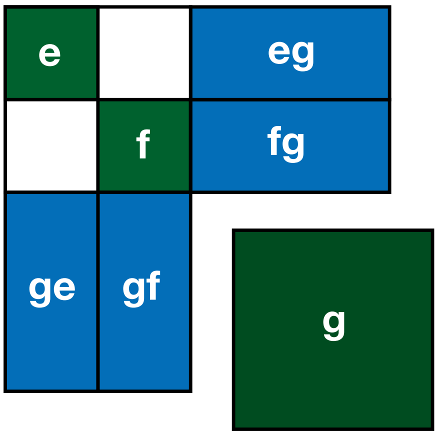
Factorize the 4 independent diagonal blocks A_9-A_{12} with a dense direct solver.



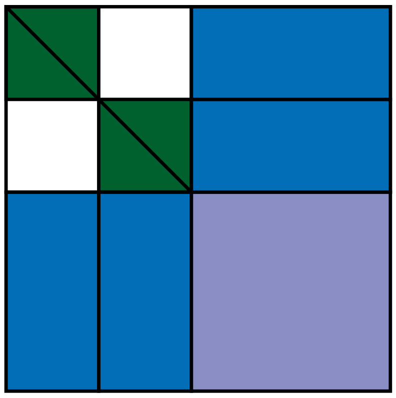
Compute Schur's update using result of factorized block $L_9, U_9, \cdots\ L_{12}, U_{12}$.



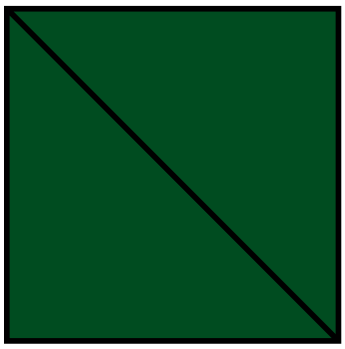
Factorize 2 independent blocks A_{13} and A_{14} with dense direct solver.



Update 15 by computing $A_{15} = A_{15} - A_{15,13}A_{13}^{-1}A_{13,15} - A_{15,14}A_{14}^{-1}A_{14,15}$.



Factorize the last block A_{15} then the scheme is done.



3.3 Task Distribution

This section discuss the distribution of factorization tasks on MPI processes in order to achieve higher scalability.

3.3.1 Communication Avoiding Distribution

The target of this distribution policy is to reduce the invocation of communication operations. In this distribution, all leaf tasks are assigned equally on each MPI process, for non-leaf tasks, it will be assigned to the same process of its left child.

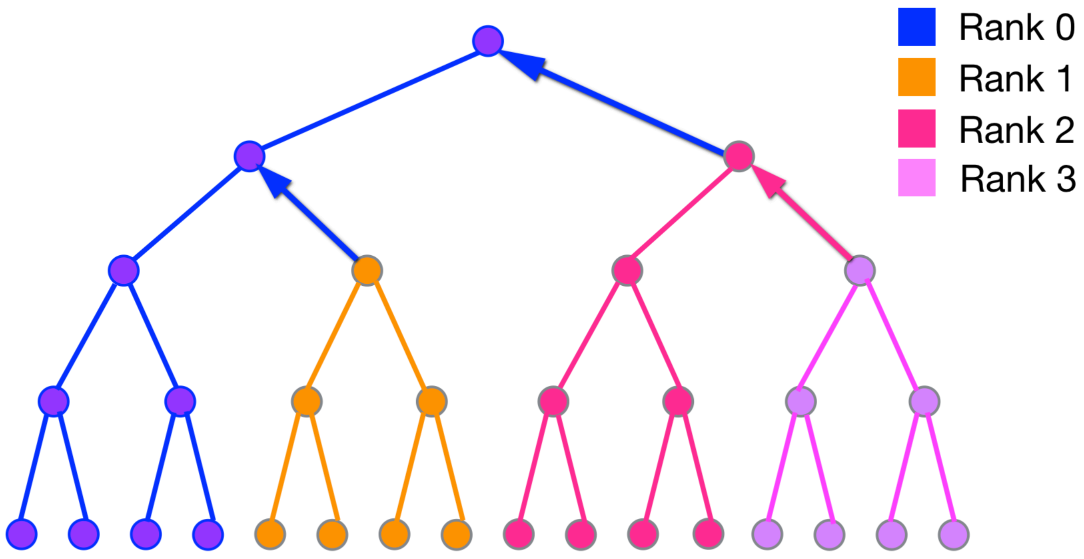


Figure 3.4: Communication Avoiding Distribution on 4 MPI Processes

3.3.2 Computation Load Balancing Distribution

Consider the following matrix S_i of task i in hierarchical Schur factorization

$$\begin{bmatrix} A_{i,i} & B \\ C & S \end{bmatrix} \tag{3.2}$$

Let the dimension of $A_{i,i}$ be $m-by-m$, B be $m-by-n$ and S be $n-by-n$, the total number of float point operations is about $O(\frac{2}{3}m^3+2m^2n+2mn^2)$ to factorize $A_{i,i}$ and compute $S = S - CA_{i,i}^{-1}B$. After the computation is done, the process have to send S to the process which will launch the parent task of i. The bytes to be sent is $O(n^2)$. Assume the computation performance is a constant C_p in GFLOP per second, the interprocess communication bandwidth is also a constant C_c in bytes per second and the communication latency is small enough. Then the ratio $\frac{CommunicationTime}{ComputationTime}$ is about $O(\frac{n^2}{\frac{2}{3}m^3+2m^2n+2mn^2})$. Which is expected to be small when m are n large. In this distribution policy we aim to distribute the computation load equally to all computation resources. By calculating the number of floating point operations needed by each task, it is possible to estimate how long will a task take. With the assumption of low communication time, we assign a tasks to a specific process with the following algorithm.

Definition 3.1. The **Minimal Start Time(MST)** of a task i is

- 0 if task i is a leaf.
- task compute cost + max(MST of left child, MST of right child) otherwise

The task selection criteria in load balanced distribution is

1. Select tasks with all leaf assigned to a process.
2. If number of tasks > 1 after last step, select tasks with maximum MST
3. If number of tasks > 1 after last step, select tasks with maximum cost
4. If number of tasks > 1 after last step, select the one with smallest ID

Chapter 4

Implementation

This chapter includes our implementation details of proposed Hierarchical Schur algorithm, or the HiS package in short, currently this software is targeted to symmetric positive definite linear systems, the developing environment is CentOS 6.5 with GNU gcc 5.2.0 compiler toolkit.

4.1 C++ Implementation

4.1.1 Object Oriented Design

The HiS package is carefully designed and implemented in C++ in objective oriented programming model, with properly applied design patterns, The implementation achieves high maintainability, extensibility and usability.

HiS package allows user defined implementations of matrix containers, sparse matrix reordering algorithms, task distributing engines and linear solvers to work with HiS solver without modifying existing codes.

4.1.2 C++ 11 Standard

The benefit to adopt C++ 11 is huge in our implementation. This standard is a great improvement from C++ 03, the new features as lambda function support can make the design more flexible, right value reference helps developer manipulate temporary objects generated by some syntax and avoid performance traps, the new native threading libraries and parallel generic algorithm library also make developing to parallel programs easier.

4.1.3 Template Metaprogramming and Compile Time Polymorphism

In order to support as more numerical types as possible, most part of HiS package is implemented in C++ templates. Furthermore the underlying BLAS operations are encapsulated by type independent wrappers, which delegates the selection of correct type of function call to the compiler.

4.1.4 Multilevel Parallelism

HiS package is designed for clusters with multicore CPUs and powerful GPUs, to maximize the usage of the computation resources, carefully examination of each parallel level is necessary.

OpenMP 4.0 SIMD

OpenMP 4.0 provides a new directive `simd`, `simd` is to hint the compiler to generate vectorization code such as SSE or AVX instructions. This feature enables higher performance within a thread.

Multithreading with Modern C++

C++ 11 standard introduced a variety of parallel libraries, the `thread` library has similar functionality to POSIX thread with better language specific feature support like `lambda` expressions. The `future` library provides APIs to launch tasks asynchronously. Moreover, C++ 11 comes with a parallel `algorithm` library which can be enabled by specific compiler flags.

HiS package adopt `thread` and `future` to manage threading of tasks. The parallel `algorithm` is used to accelerate array sorting, transformation operations.

CUDA

The critical performance regions of HiS package is the factorization and building Schur's complement. Since these operations are in BLAS level 3, GPU will be good choice of accelerator. Both of sparse and dense factorization involve BLAS level 3 operations, while the dimension of matrix is large enough, performance of GPU is possibly better than CPU.

MPI

MPI support allows HiS to access more computing resources, in our current progress a factorization or solving task will map to a corresponding MPI process. Each MPI process can use one or more GPU device and multiple software threads.

4.2 Sparse Factorization

Sparse Cholesky solver is one kernel of HiS package, this section illustrate the selection of default sparse solver and benchmark results.

4.2.1 CPU Sparse Solver and GPU Sparse Solver

Various candidates of sparse solver are available, the sparse Cholesky factorization involves dense BLAS level 3 operations, however in hierarchical Schur method, sparse factorization is only applied on each subdomain blocks, in this case GPU solver is not necessary better than CPU solvers, to figure out the adequate solution, we have benchmark the performance of some sparse solvers.

4.2.2 Schur's Update Computation in Factorization

Building Schur's complement update is needed in hierarchical Schur method, in order to take advantage of sparsity of sparse multiple right hand side, we consider to solve the right hand during the factorization.

4.3 Dense Factorization

Dense Cholesky solver is another kernel of HiS package, since the dimension of dense blocks in hierarchical Schur method are usually large, GPU is solver is considered as default in HiS package. In this situation, we considered MAGMA and CUSOLVER as candidate of dense Cholesky solver.

4.3.1 GPU Dense Solver

Here is benchmark result of CUSOLVER Cholesky and MAGMA Cholesky on NVIDIA K40 GPU. The result shows the performance of MAGMA is much better than CUSOLVER between selected dimension of matrix. Our implementation adopt MAGMA Cholesky and other GPU BLAS routines in MAGMA.

4.4 Compressed Dense Block Matrix

4.4.1 Purpose and Introduction

Building Schur's update after dense factorization involves BLAS level 3 operations `trsm`, `syrk` and `gemm`, the computation cost is very high however for tasks closed to leaf level of binary tree, we found that in most cases many columns to be solved are all zero, thus, to reduce redundant computation and memory occupation, we introduce compressed dense block matrix to solve this issue. The idea of compressed dense block matrix is to remove empty columns are rows in order to reduce the size and computation cost of the matrix. In 4.1 a matrix is converted to compressed dense block format, two arrays is needed to store original row and column index.

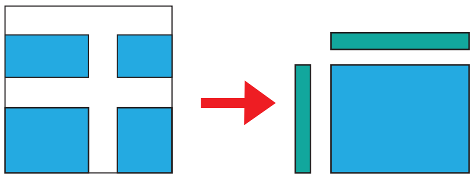


Figure 4.1: Example of CDB matrix

Chapter 5

Results

5.1 Test Environment

The performance results of HiS package is evaluated on IBM deep computing cluster(DCC). Each node of the cluster contains two Intel Xeon E5-2667 v2 octa-core CPU with 256 GB memory, two NVIDIA K40c GPU with 12GB device memory and infinitiband with upto 56Gb per second bandwidth. The code is compiled using GNU GCC version 5.2 with optimization options `-O3 -march=native -flto -fipa-icf -fipa-reference -fopenmp`. The optimization options is used for improving performance and reduce the code size. The CUDA toolkit version used is 7.0, MAGMA version 1.7.0, intel MKL version 2016.

5.2 Test Problems

Most test problems are from Florida matrix collection, the criteria of problem selection is the dimension of matrix larger than 100,000.

Table 5.1: Information of Tested Problems

Name	Dimension	Non-zeros	Cholesky flops
BenElechi1	$245,874$	$13,150,496$	4.15×10^{10}
Dubcova3	$146,689$	$3,636,643$	1.99×10^{9}
Emilia_923	$923,136$	$40,373,538$	2.10×10^{13}
Fault_639	$638,802$	$27,245,944$	1.41×10^{13}
Flan_1565	$1,564,794$	$114,165,372$	6.13×10^{12}
G2_circuit	$150,102$	$726,674$	2.63×10^{9}
G3_circuit	$1,585,478$	$7,660,826$	6.65×10^{10}
Geo_1438	$1,437,960$	$60,236,322$	2.64×10^{13}
Hook_1498	$1,498,023$	$59,374,451$	1.45×10^{13}
Serena	$1,391,349$	$64,131,971$	4.27×10^{13}
StocF-1465	$1,465,137$	$21,005,389$	4.77×10^{12}
af_0_k101	$503,625$	$17,550,675$	7.19×10^{10}
af_1_k101	$503,625$	$17,550,675$	7.19×10^{10}
af_2_k101	$503,625$	$17,550,675$	7.19×10^{10}
af_3_k101	$503,625$	$17,550,675$	7.19×10^{10}
af_4_k101	$503,625$	$17,550,675$	7.19×10^{10}
af_5_k101	$503,625$	$17,550,675$	7.19×10^{10}
af_shell3	$504,855$	$17,562,051$	6.32×10^{10}
af_shell4	$504,855$	$17,562,051$	6.32×10^{10}
af_shell7	$504,855$	$17,562,051$	6.32×10^{10}
af_shell8	$504,855$	$17,562,051$	6.32×10^{10}

apache2	$715,176$	$4,817,870$	2.12×10^{11}
audikw_1	$943,695$	$77,651,847$	8.90×10^{12}
bmw7st_1	$141,347$	$7,318,399$	1.68×10^{10}
bmwcra_1	$148,770$	$10,641,602$	9.29×10^{10}
bone010	$986,703$	$47,851,783$	7.19×10^{12}
boneS01	$127,224$	$5,516,602$	6.69×10^{10}
boneS10	$914,898$	$40,878,708$	3.13×10^{11}
cfd2	$123,440$	$3,085,406$	4.29×10^{10}
ecology2	$999,999$	$4,995,991$	1.63×10^{10}
hood	$220,542$	$9,895,422$	1.06×10^{10}
inline_1	$503,712$	$36,816,170$	1.78×10^{11}
ldoor	$952,203$	$42,493,817$	9.38×10^{10}
msdoor	$415,863$	$19,173,163$	2.34×10^{10}
offshore	$259,789$	$4,242,673$	1.05×10^{11}
parabolic_fem	$525,825$	$3,674,625$	8.81×10^{9}
pwtk	$217,918$	$11,524,432$	2.85×10^{10}
ship_003	$121,728$	$3,777,036$	1.37×10^{11}
shipsec1	$140,874$	$3,568,176$	5.34×10^{10}
shipsec5	$179,860$	$4,598,604$	9.12×10^{10}
shipsec8	$114,919$	$3,303,553$	5.61×10^{10}
thermal2	$1,228,045$	$8,580,313$	1.76×10^{10}
thermomech_TC	$102,158$	$711,558$	4.98×10^{8}
thermomech_TK	$102,158$	$711,558$	4.98×10^{8}
thermomech_dM	$204,316$	$1,423,116$	9.45×10^{8}

`tmt_sym`	$726,713$	$5,080,961$	1.31×10^{10}
`x104`	$108,384$	$8,713,602$	2.99×10^{10}

5.3 Performance Results

5.3.1 Sequential Performance

The following shows the performance of our implementation of hierarchical Schur algorithm on a single GPU multi-core CPU computer. HiS is running with one GPU card and one CPU core, Pardiso is running on one CPU core. This result shows single core Pardiso has higher performance for smaller matrices, and HiS can be much faster when Cholesky flops grown up. The reason should be that GPU is suitable accelerator for massively parallel application such as BLAS level 3 operations.

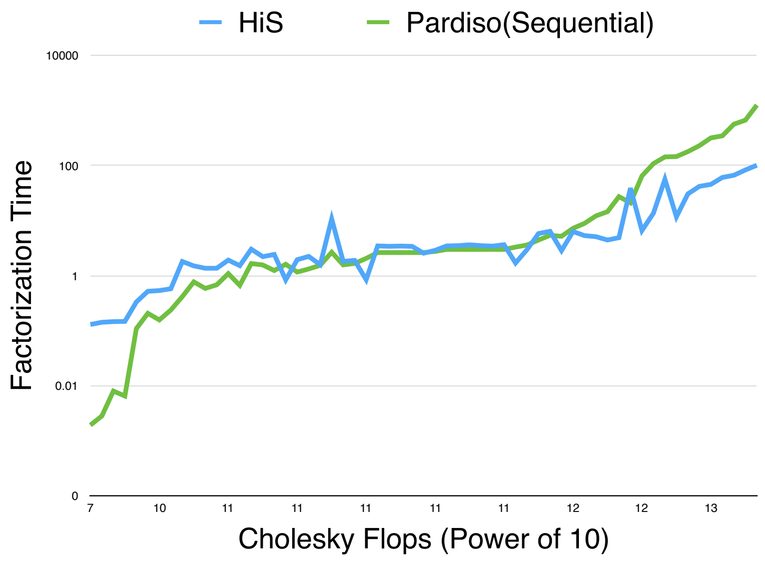


Figure 5.1: Sequential Performance Evaluation

5.3.2 Parallel Performance

The parallel scalability results is measure by running on 1, 2, 4, 8 MPI processes, each process used one GPU card and one CPU core. The x-axis is sorted by Cholesky flops of problems. The task distribution is communication avoiding. This result advices the scalability of hierarchical Schur method is independent from problem scale.

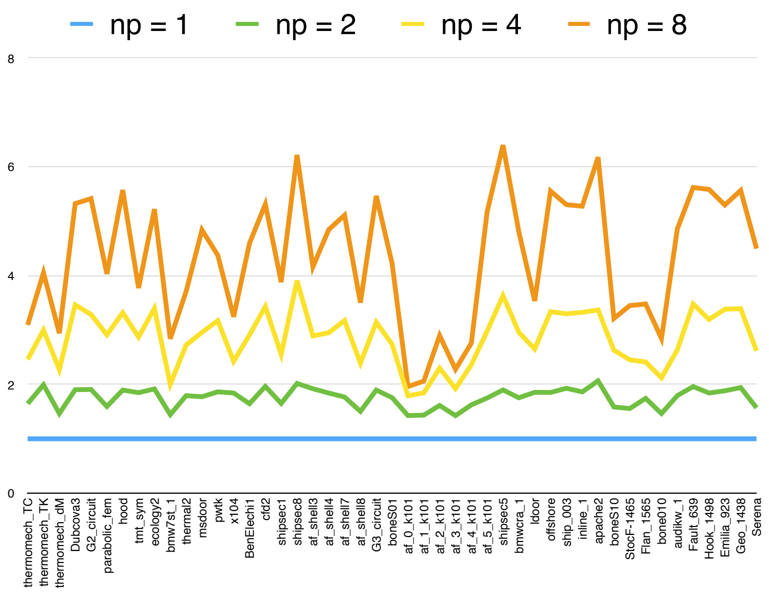


Figure 5.2: Parallel Result with Different MPI Size

In this research we test load balanced distribution with Fault_639 matrix. This result in the table proves load balanced distribution is possible to improve scalability of hierarchical Schur method, however, the size of data communication is grown very fast since we do not consider that in our policy.

Table 5.2: Compare result of different distribution of Fault_639 matrix

	Communication Avoid	Load Balance
P=1	86.975s	87.112s
P=4	36.500s	32.726s
Speedup	2.38 times	2.66 times
Data	3,138,682,568 bytes	17,512,362,456 bytes
Data Ratio	1	5.58

Chapter 6

Conclusion

6.1 Future Works

6.1.1 General Linear System Solver

Hierarchical Schur algorithm exploits the binary partition property of nested dissection to parallelize the factorization scheme. While the matrix is symmetric positive definite, no pivoting is need in the scheme. However when the matrix is not SPD, pivoting is necessary in many cases. Consider a matrix A

$$A = \begin{bmatrix} 1 & 0 & & & 0 \\ 0 & 1 & & & 0 \\ & & 1 & 0 & 0 \\ & & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{6.1}$$

In hierarchical Schur method, we have diagonal blocks

$$A_{1,1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{6.2}$$

$$A_{2,2} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \tag{6.3}$$

Apparently $A_{2,2}$ is singular but A^{-1} exists and can be written down

$$A^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 1 \\ 0 & 0 & -1 & 1 & 0 \end{bmatrix} \tag{6.4}$$

In this case, pivoting is necessary but this will break the structure of matrix which will void the application of hierarchical Schur method. Since a majority of problems are not symmetric positive definite, it is necessary to deal with the pivoting problem.

6.1.2 Interprocess Task Support

Currently in our implementation a factorization or solving task will only be launched on one MPI process, while the scheme reach the end, more and more computation resources are idling since there is not enough tasks. To utilize most resources, one possible method is to make tasks can be executed on multiple processes.

Bibliography

[1] P. R. Amestoy, I. S. Duff, J.-Y. L' Excellent, and J. Koster. Mumps: a general purpose distributed memory sparse solver. In *Applied Parallel Computing. New Paradigms for HPC in Industry and Academia*, pages 121–130. Springer, 2001.

[2] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.6, Argonne National Laboratory, 2015.

[3] C. Che-Ming. Hybrid hierarchical schur solvers for large sparse linear systems on cpu-cpu cluster. Master's thesis, Department of Mathematics, National Taiwan University, 2014.

[4] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam. Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate. *ACM Trans. Math. Softw.*, 35(3):22:1–22:14, Oct. 2008.

[5] D. Y. Chenhan, W. Wang, and D. Pierce. A cpu–gpu hybrid approach for the unsymmetric multifrontal method. *Parallel Computing*, 37(12):759–770, 2011.

[6] P. Labs. Paralution v1.0.0, 2015. `http://www.paralution.com/`.

[7] O. Schenk, K. Gärtner, W. Fichtner, and A. Stricker. Pardiso: a high-performance serial and parallel sparse linear solver in semiconductor device simulation. *Future Generation Computer Systems*, 18(1):69–78, 2001.

[8] I. Yamazaki and X. S. Li. On techniques to improve robustness and scalability of the schur complement method.