Structure-adaptive parallel implementation of solution for sparse triangular system

Satish Kumar,[*] M. Raviteja[†]

SuperComputer Education and Research Centre, Indian Institute of science, Bengaluru, India.

Abstarct

Solving Sparse triangular systems holds importance in many of the numerical linear algebra packages. Direct methods pose a problem as these are highly resistant to parallelism. Hence a scalable technique to solve them is always sought. In this project, we have implemented novel parallel algorithms, developed by E. Totoni et al.[1], which focus on exploiting parallelism through sparsity. We have developed a method for hybrid (MPI-CUDA) implementation of solution of sparse triangular linear system. The report discusses the overview of the individual algorithms implemented. In the end the scalability of the algorithm is analyzed. Also our implementation is compared with the 'csrsv' routine available in the cuSparse library in terms of execution time.

I. Introduction

Sparse triangular systems are encountered in numerous science and engineering applications. Typically, the need to solve these are encountered in many iterative methods (such as Gauss Seidel method).Other application areas are in using the pre-conditioners for other iterative methods. (The incomplete cholesky preconditioner).
Triangular systems, in general are highly resistant to parallelism. Solving for each unknown can only progress sequentially, given the structural dependencies among the unknowns have not been analyzed. This leads to poor scaling, in a parallel implementation of solving a triangular system. Added to this, is the cost of communication which also can progress only sequentially. This can result in a higher solution time when compared with a single process execution time. In spite of this, the system is often solved in parallel because of the memory scalability that is inherently available. Also, the matrix is generally present distributed across the processors, as a result of a previous computational step. Thus, it is desirable to analyze the structure of the matrix in order to exploit parallelism through the available sparsity of the triangular matrix.

II. Related Work

Most of the prior work, laid focus on two different approaches. Generating a directed acyclic graph, is one among them. The parallelism is realized in identifying the independent rows to solve in levels. Each level progresses when the solution of the prior level is known. The computations within a level can be performed in parallel, given the previous level information is available to each of the new row in the next level. This methodology has been implemented in CUDA - CuSparse library (*cusparse<t>csrsv*)[2]. However this is a shared memory implementation. Replicating this in a distributed machine, would require data distribution and global synchronizations (the same level has to be processed across each process at any given time). As described by Totoni et. al. [1], other techniques involve partitioning the matrix to sparse factors and computing inversions of the off-diagonal elements. These procedure depends on the numerical accuracy of the inversions limiting their usage.

[*]MTech SERC: 11052 p.kumarsatish@gmail.com
[†]MTech SERC: 11051 raviteja@gmail.com

Totoni et.al.[1] discuss novel parallelism strategies that can be adopted to a distributed architecture. Their implementation has been done in charm++, a higher level abstraction of MPI. In the following, we discuss these strategies in general, also highlighting the complexities of the MPI-CUDA (CPU-GPU) hybrid implementation.

III. Methodolgy

Three main strategies have been recognized to exploit parallelism.
(1) Identifying independent rows
(2) Early communication of the computed data
(3) Parallel processing of off-diagonal rows

The implementation starts with distributing the obtained matrix in blocks, which are a set of continuous columns, to each process. These form the basic unit of parallelism. For implementing the first strategy, the initial phase involves generating a dependency information that is communicated across the processes. Initially each block scans through its local (off diagonal) rows and columns to identify the external dependency and communicate across each other. This enables each block to identify the rows (independent) that can be processed without any information external to the block. A simple heuristic adopted here was to reorder the independent rows in the reverse order to enable faster evaluation of the near diagonal dependencies. These two algorithms were directly adopted from [1].The algorithm proceeds next to two important phases: the analysis phase and the solve phase. In order to achieve the early communication strategy, each diagonal row maintains a list of off diagonal rows that can be computed and sent when that particular unknown is computed (refer algorithm 1). This is required as all unknowns are computed in the permuted order, and each off diagonal row depend on different diagonal rows.

Terminology:
{S1}: Set of independent diagonal rows
{S2}: Set of dependent diagonal rows
{S3}: Set of off-diagonal rows
{S4}: {S2} U {S3}

Algorithm 1: Constructing Lists

```
input : Block-Dependancy Information
        Dsend[] and DRecv[],
        Reordering Information Perm[]
output: List List_Ind[] and List_Dep[],
        Value Num[];
Initialize all list to NULL and all Num to zero;
for each row r in set S4 do
    if r.depend = true then
        for each non-zero e in r do
            add r to list_Dep[Xe];
            Num[r] ← Num[r]+1;
        end
    else
        Xl← last Xi needed for row r;
        add r to list_Ind[Xl];
    end
end
```

The solve phase starts with the triangular solve routine, as described in algorithm 2.

Algorithm 2: Triangular Solve

```
Algorithm: triangular_solve
input : Row myRows[], Value myRhs[]
Output: Value x[]

for each row $r \in \{S1\}$ do
    process_row_ind(r, myRhs);
end

while any DataMessage msg arrived do
    Recieve the msg
    row r ← tag of message
    if $Num[r] = -1$ then
        if $r \in S2$ then
            process_row_dep(r, myRhs,
            val);
        else
            send the partial sum;
        end
    else
        r.depend ← false;
    end
end
```

Each independent row (refer algorithm 3), after its computation, checks its list and launches a GPU kernel for processing all the off-diagonal rows together. The intermediate step involves copying the rows to a local CSR format before copying to the device. The GPU operation here is a matrix vector multiplication routine.

Algorithm 3: Processing Independent Rows

```
Algorithm: process_row_ind
input   : Row r, Value myRhs
Output: Value x

compute x ← (myRhs - val)/$l_{ii}$;
compute_gpu(List_Ind[r]);
for each row r1 in List_Ind[r] do
    if r1.depend = flase then
        send the partial sum;
    else
        Num[r1] ← -1;
    end
end
```

Algorithm 4: Processing Dependent Rows

```
Algorithm: process_row_dep
input   : Row r, Value myRhs, Value val
Output: Value x

compute x ← (myRhs - val)/$l_{ii}$
for each row r1 in List_Dep[r] do
    Num[r1] ← Num[r1]-1;
    if Num[r1] = 0 then
        Update val;
        if r1.depend = false then
            if r1 is diagonal row then
                process_row_dep(r1,
                myRhs, val);
            else
                send the partial sum;
            end
        else
            Num[r1] ← -1;
        end
    else

    end
end
```

The remaining rows (refer algorithm 4) are processed based on the arrival of the dependency messages from the neighboring blocks. This step needed an effective communicator in order to be able to randomly receive a message from any source. MPI_Probe conveys the information of the waiting messages for a given process. Since the total messages to be received are known (dependency information), this function is executed until all messages are received.

IV. Experiments and Results

A. Experimental Setup

All experiments were conducted on the Fermi cluster on a single nodelink for fermi system configuration. The number of processes used are 1, 2, 4, 8, 16 and 32. The implementation has been tested on 3 different matrices. The data was downloaded from The University of Florida Sparse Matrix Collection[3]. The nature of the data is summarized as in the table 1.

Name	Dimension	Indep. Rows	NonZeros
nlpkkt80	1,062,400	1,062,400	1,062,400
largebasis	440,020	200,100	3,000,062
hood	220,542	192,353	2,143,007

Table 1. Matrcies used for experiments

B. Results

The hybrid implementation here is compared with the Cusparse routine Csrsv, which is a CUDA implementation. Since both the implementations proceed in two steps, the evaluation time are also noted separately, enabling for better analysis.

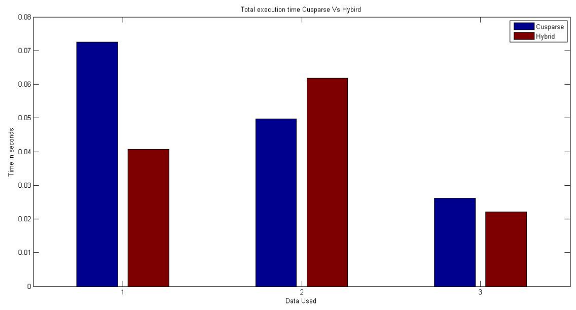

Figure 1. Total time taken

	np = 1	np = 2	np = 4	np = 8	np = 16	np = 32	cuSparse
nlpkkt80	0.0836	0.049304	0.029697	0.030979	0.03745	0.030172	0.0665
largebasis	0.096381	0.046978	0.025448	0.026277	0.031703	0.031799	0.0473
hood	0.030496	0.020086	0.014767	0.01714	0.012492	0.015158	0.0192

Table 2. Time in seconds, taken in analysis phase

	np = 1	np = 2	np = 4	np = 8	np = 16	np = 32	cuSparse
nlpkkt80	0.0334	0.01714	0.012049	0.010913	0.011366	0.011121	0.0060
largebasis	0.10459	0.045027	0.038597	0.038089	0.036292	0.038481	0.0025
hood	0.023584	0.013311	0.007355	0.009394	0.00753	0.00917	0.007

Table 3. Time in seconds, taken in solve phase

	np = 1	np = 2	np = 4	np = 8	np = 16	np = 32	cuSparse
nlpkkt80	0.117000	0.066444	0.041746	0.041892	0.048816	0.041293	0.072500
largebasis	0.200971	0.092005	0.064045	0.064366	0.067995	0.070280	0.049800
hood	0.054080	0.033397	0.022122	0.026534	0.020022	0.024328	0.026200

Table 4. Total time taken in seconds

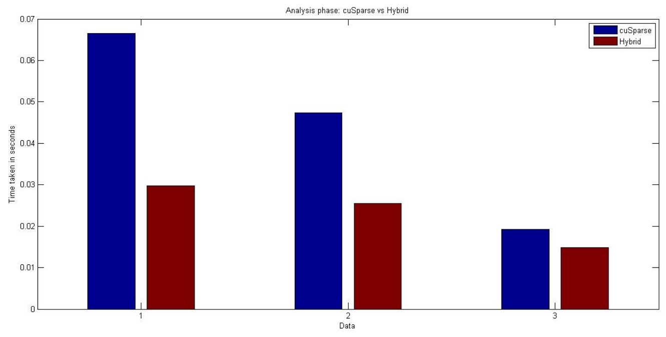

Figure 2. Time taken in analysis phase

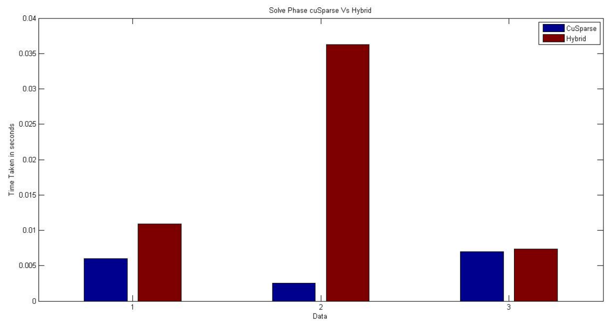

Figure 3. Time taken in solve phase

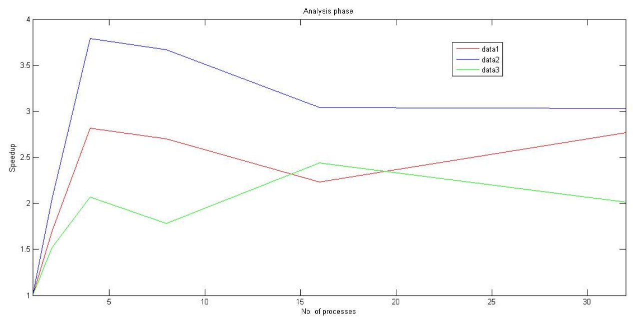

Figure 4. Speedup for analysis phase

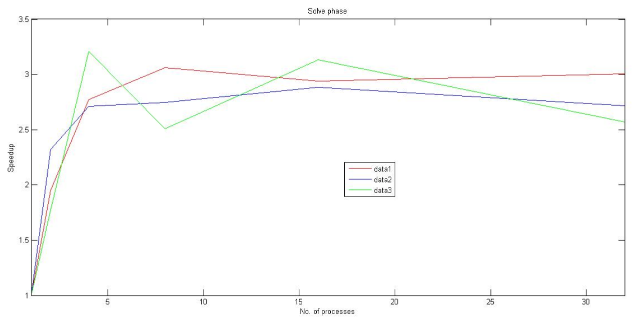

Figure 5. Speedup for solve phase

V. Conclusions

It is observed that both the phases are scalable, but this limits out after 4 processes. This is expected as the number of physical processors available in a single node are 4. The analysis phase seems to take considerably higher time than the solve phase. Also the scalability available in the analysis phase resulted in faster execution time when compared to the cusparse library function.
The difference in the time involved in the analysis phase and the solve phase can be offset when the need arises to solve a given triangular system for multiple RHS (the b in $Lx = b$). However, it could be observed that the Cusparse library has a faster solve phase. This is specifically notable when the sparseness of the matrix indicates a nearly embarrassingly parallel solve phase.
With memory scalability and with the availability of additional processors, it is expected that the hybrid implementation would result in a much faster analysis phase, with also notable improvements in the solve phase which would be entirely absent in the CUDA implementation.

References

[1] E. Totoni, M.T. Heath, L.V. Kale, *Structure-adaptive parallel solution of sparse triangular linear systems,* Parallel Computing, 40 (2014) 454–470. Science-Direct link

[2] M. Naumov, *Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU,* NVIDIA, 2701 San Tomas Expressway, Santa Clara, CA 95050. [PDF] from research.nvidia.com

[3] The University of Florida Sparse Matrix Collection

[4] cuSPARSE Documentation on docs.nvidia.com