Optimized Parallel Sparse Matrix Solver for Large-Scale Linear Circuit Analysis

Sushirdeep Narayana, Texas A&M University, College Station
Amith Pai, Texas A&M University, College Station

***Abstract*—In the recent years, there has been a dramatic shift towards multicore computing in search for better performance and scalability. There is also a huge demand for high performance and accuracy in parallel circuit simulation tools. Specific multicore optimizations are required for solving large scale sparse matrices through efficient algorithms. There exist many large scale linear circuits, for example on-chip power nets, which must be analyzed accurately. In this project, we have developed a parallel sparse matrix solver for large scale linear circuit analysis based on iterative approach. This project deals with speeding up the DC/ transient simulation process of these type of circuits (> 1 million nodes). OpenMP parallel computing platform is used to achieve the goal of creating such a parallel sparse matrix solver. The developed solver has been tested with matrices of various types and sizes. The performance metrics of the parallel solver has been analyzed and optimized to deliver good speedup.**

***Index Terms*—Circuit Simulation, Linear Circuits, Symmetric Sparse Matrices, Sparse Matrix Vector Mulitpication, Loop Parallelism**

I. Introduction

Circuit simulators form a significant tool in the IC design industry. Simulators are generally used to prototype a design, predict the circuit performance and verify the design of the electronic circuit before it goes for tape-out of the chip. This has necessitated the use of circuit simulators to model and simulate the circuit behavior at the tool level in order to achieve highly efficient and accurate circuits in terms of both functionality as well as cost. The circuit designers today have access to cheaper and more powerful computing facilities due to the growth of the semiconductor industry from Moore's Law. It should be noted that there is an increased demand for simulation tools with higher accuracy, faster speed and new analysis capabilities as the designers move towards increasingly complex IC designs. Compute Intensive applications have necessitated the shift to multi-core processor technology. Several highly parallel computing platforms are now available to support parallel computations. This has resulted in a natural move towards highly parallel applications which can take advantage of the inherent parallel properties of matrices and the corresponding algorithms to improve the speed and accuracy. However there are multiple challenges which needs to be addressed in order to develop a highly coherent application for sparse matrix solving. This has motivated new research activities in parallel circuit simulation.

Serial programs written for a conventional single core processor, usually cannot exploit the presence of multiple cores. Converting these into parallel programs, which can make use of multiple cores forms the significant part of EDA tool development industry.

Sparse matrices are at the heart of linear algebraic systems. Typical applications such as on-chip power grids and industrial power nets are highly sparse massive circuits and usually have greater than one million nodes. These circuits require highly parallel efficient algorithms to accurately analyze them in the design process. There exists both direct and iterative methods to solve the sparse matrices. Highly efficient parallel direct methods based on LU factorization [10] can be employed to solve the matrices. Iterative methods based on Conjugate Gradient (CG) method or General Minimal Residual Method (GMRES) can also be incorporated in a circuit simulator to improve its performance.

Direct methods are based on solving the equation, $Ax = b$ by matrix decomposition. Cholesky decomposition and LU decomposition algorithms are used for symmetric and non-symmetric matrices respectively. The following steps are used to obtain the complete solution:

1. A preordering step that re-orders the rows and columns such that the factors suffer little fill-ins.
2. Symbolic factorization that determines the non-zero structures of the factors and create suitable data structures for the factors.
3. Numerical factorization that computes the L and U factors.

A solve step that performs forward and backward substitution using the factors.

However, the time complexity of computing the inverse of coefficient matrix A can be prohibitively high. Besides this method also has the tendency to create more fill-ins and hence can severely affect the sparsity of the original matrix and lead to memory limitations. Iterative methods mentioned earlier use successive approximations to obtain accurate solutions at each step. They also are more effective in terms of memory. But these methods are usually limited by accuracy and convergence of the iterative solution.

The main objective of this project is to develop a Conjugate gradient based iterative linear circuit solver for very large matrices using parallelization concepts. Section 2 of the paper presents a brief background of the different concepts used

while developing the parallel solver. The workflow outline is discussed in section 3. Section 4 describes the experimental setup used in the design. The various simulation results and the analysis of the results is provided in section 5. Section 6 concludes the paper with findings and discusses the scope for further improvement of the parallel solver.

II. Relevant Background

A linear circuit can be modeled by a linear differential equation

$$C\,\frac{d}{dt}\,x(t) + G\,x(t) = b\,u(t) \qquad (1)$$

where C is the capacitance matrix (it may have contributions from inductive elements), G is the conductance matrix, and b is the input vector that connects external input $u(t)$ to the system. Applying a standard numerical integration method, say Backward Euler, leads to a system of linear equations for each time point

$$C\,\frac{x(t_i) - x(t_{i-1})}{h_i} + G\,x(t_i) = b\,u(t_i) \qquad (2)$$

Let $A = \frac{C}{h_i} + G$ and $v = C\frac{x(t_{i-1})}{h_i} + b\,u(t_i)$, then equation (2) can be written as

$$A\,x = v \qquad (3)$$

where $x = x(t_i)$.

III. Workflow Outline

In this section the design and development of our optimized parallel sparse matrix solver is described. The Workflow Section is organized as follows: a brief overview of the Conjugate Gradient will be provided in Section 3(a) and Section 3(b), effective storage of the sparse matrices will be discussed in Section 3(c), aspects of parallelism exploited in iterative methods are described in Section 3(d).

A. Overview of Iterative Methods

Iterative methods refer to the wide range of techniques that use successive approximations to obtain more accurate solutions to a linear system. The rate at which an iterative method converges depends on the spectrum of the coefficient matrix. Hence, some iterative methods usually involve a second matrix that transforms the coefficient matrix into one with a more favorable spectrum. The transformation matrix is called preconditioner.

The Conjugate Gradient methods fall under the category of Krylov subspace methods. The Krylov subspace methods deal with solving $Ax = b$ by finding a sequence $x_1, x_2, x_3, \ldots, x_k$ that minimizes some measure of error over the corresponding spaces

$$x_0 + K_i(A, r_0), \quad i = 1,2,3,\ldots,k$$

These sequences are defined by two conditions:

1) Subspace condition: $x_k \in x_0 + K_k(A, r_0)$

2) Petrov- Galerkin condition: $r_k \;\; orthogonal\; to\; L_K$

$$\Leftrightarrow\; (r_k)^t\, y = 0,\; \forall\, \mathbf{y} \in L_K$$

where x_0 is the initial iterate, r_0 is the initial residual, $K_k(A, r_0)$ is the Krylov subspace of dimension k and L_K is a well defined subspace of dimension k.

Conjugate Gradient method is the appropriate iterative Krylov method when dealing with symmetric positive definitive matrices as shown in Figure 1.

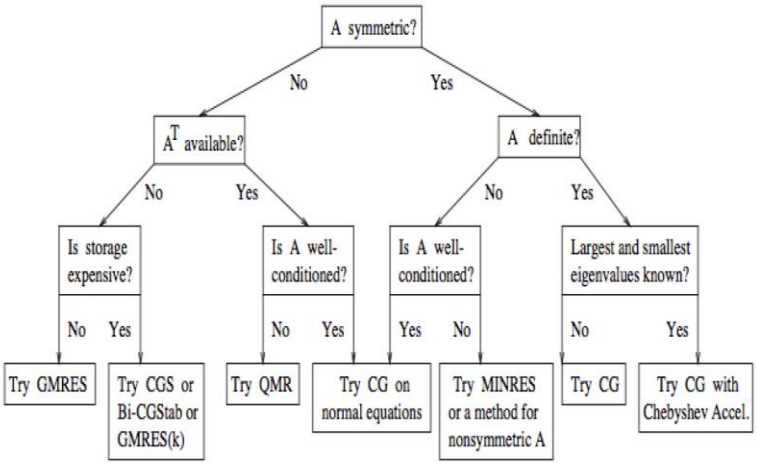

Figure 1: Overview of Iterative methods based on the properties of matrix A

B. Conjugate Gradient Algorithm

The Conjugate Gradient method derives its name from the fact that it generates a sequence of orthogonal vectors. These vectors are obtained from the computation of residuals during iterations. They can also be viewed as the gradients of a quadratic functional as shown by equations (1) and (2). The minimization of these gradients is equivalent to solving the linear system.

$$\phi(x) = \tfrac{1}{2}\,x^t\,A\,x\; - b^t\,x \qquad (1)$$

$$\nabla\phi(x) = Ax - b = 0 \qquad (2)$$

$$\nabla\phi(x) = Ax - b = 0 \quad \Leftrightarrow \quad x^* = A^{-1}\,b \qquad (3)$$

The CG algorithm is described below

Algorithm1

1) **Initializations**

$$r_0 = b - Ax_0\,, \rho_0 = ||r_0||_2^2\;,\; p_1 = r_0\;,\; k = 1$$

2) **Entering the iterative loop by checking the residual condition and k_max condition**

$$while\;\left(\sqrt{\rho_k}\;\; > \epsilon\left|\left|b\right|\right|_2\; and\;\; k < k_{max}\right)\; do$$

3) **Update the search direction p_k by computing the parameter β_k**

$$if\;(k\;\neq 1)\;then$$
$$\beta_k\;=\;(r_{k-1}, r_{k-1})/\;(r_{k-2}\,,\; r_{k-2})$$
$$p_k = r_{k-1} + \;\beta_k\,p_{k-1}$$
$$end\;if$$

4) **Update the step size by computing the parameter α_k**

$$\alpha_k = (r_{k-1}, r_{k-1})/\;(Ap_k\,, p_k)$$

5) **Update the solution guess x_k**

$$x_k = x_{k-1} + \;\alpha_k p_k$$

6) **Update the residual r_k**

$$r_k = r_{k-1} - \alpha_k A p_k$$

7) **Computing the norm of the residual for checking the loop condition**

$$\rho_k = ||r_k||^2$$

8) **Iteration update**

$k = k + 1$

end while

After k iterations of the CG algorithm, it can be shown that the error of the solution guess x_k is bound in terms of the spectral condition number κ(A) by equation (4)

$$||x^* - x_k||_A \leq 2||x - x_0||_A \left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^k \quad (4)$$

where, x_0 is the initial guess,

$||x||_A = \sqrt{x^T A x}$ *and* $\kappa(A) = \left|\frac{\lambda_{max}(A)}{\lambda_{min}(A)}\right|$, $\lambda_{max}(A)$ and $\lambda_{min}(A)$ are the smallest and the largest eigenvalues of a symmetric positive definite matrix A.

The implementation of CG algorithm involves one matrix-vector product, three vector updates and two inner products per iteration. Section 3(d) explores parallelizing these computations.

C. Sparse Matrices

Sparse matrices are two dimensional matrices primarily populated with zeros. They are routinely used in many iterative methods for solving large scale linear systems. There are many sparse matrix representations, each with different storage requirements, computational characteristics, and methods of accessing and manipulating entries of the matrix. In our project, the Compressed Sparse Row (CSR) format was chosen as it will be beneficial for us while storing the matrix non-zero entries and while performing Sparse Matrix Vector Multiplication.

The CSR uses three arrays for storing a sparse matrix: the **values** array stores the values of the non-zero elements of the sparse matrix in row-wise order, the **colind** stores the corresponding column indices and the **rowptr** array contains pointers to the start of each row. If ***values(k)*** = $\mathbf{a_{ij}}$, then ***rowptr(i)*** $\leq$ ***k*** $<$ ***rowptr(i+1)***. By definition of CSR format, ***rowptr(n+1) = nnz+1***, where nnz represent the number of non-zero entries of the sparse matrix A. An example of the CSR sparse matrix storage format is shown in Figure 2. Using CSR storage instead of storing $\mathbf{n^2}$ elements only **2nnz+n+1** elements are stored.

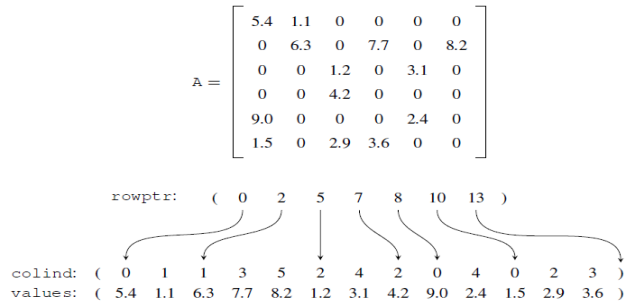

Figure 2: CSR Sparse Matrix Storage

Since the matrices we are dealing with are symmetric, we only need to store the upper (or lower) triangular portion and the diagonal elements of the matrix. As a result, the size of the **values** and **clind** arrays is reduced to (**nnz-n**)/ **2**. This procedure is shown in Figure 3.

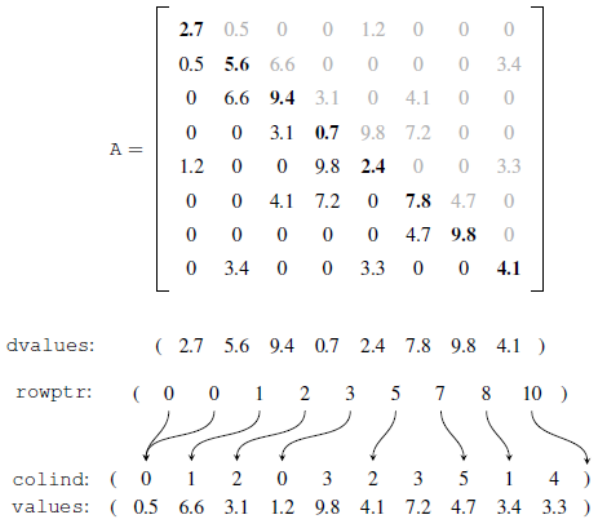

Figure 3: CSR for Symmetric Matrices

D. Parallelism

The iterative methods share most of their computational kernels. The basic time consuming kernels in our CG method are the following:

a. Inner products
b. Vector Updates
c. Sparse Matrix Vector Multiplication

Efficient parallelization of these computational kernels will lead to significant improvement in performance of the sparse matrix solver.

a. Parallelization of Inner Products

The computation of the dot product between two vectors can easily be parallelized. The input vectors are split into equal segments based on the number of threads. The dot product of each corresponding segments are computed by each thread as the local inner product. The local inner product are then combined to form the global inner product by the master thread.

b. Parallelization of Vector Updates

The computation of vector updates is easily parallelizable. The vector is split into segments based on the number of threads and each processor updates its own segment.

c. Sparse Matrix Vector Multiplication

Sparse Matrix Vector Product is a highly computational kernel in the CG algorithm. The process of sparse matrix vector multiplication is shown in Figure 4. The matrix vector product $y = Ax$ can be expressed by equation (5)

$$y_i = \sum_j a_{i,j} x_j \qquad (5)$$

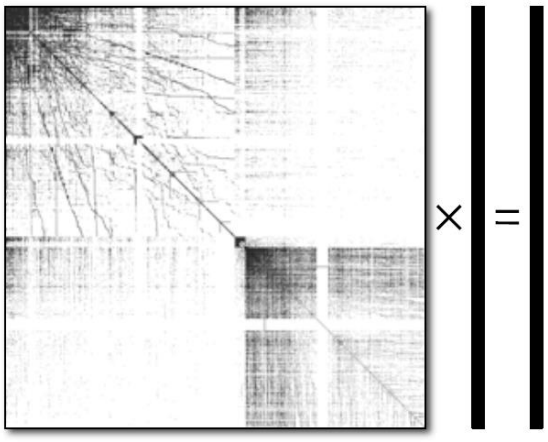

Figure 4: Sparse Matrix Vector Multiplication

Since, equation (5) traverses the rows of the matrix A, having stored the matrix in CSR format will be very beneficial. The pseudo-code for the matrix vector product obtained while multiplying a matrix A of size *nxn* stored in CSR format with an input vector x is given below

$$\begin{aligned}
&for\ i = 1\ to\ n \\
&\quad y(i) = 0 \\
&\quad for\ j = rowptr(i)\ to\ rowptr(i+1) - 1 \\
&\qquad y(i) = y(i) + values(j) * x\big(colind(j)\big) \\
&\qquad\qquad end; \\
&end;
\end{aligned}$$

As we have stored only the upper (or lower) triangular portion and the diagonal elements of the symmetric matrix, the above procedure to compute matrix vector product is modified as shown below

$$\begin{aligned}
&for\ r = 0\ to\ n-1 \\
&\quad y(r) = diagvalues(r) * x(r) \\
&\qquad for\ j = rowptr(r)\ to\ rowptr(r+1) - 1 \\
&\qquad\qquad c = colind(j) \\
&\qquad\qquad y(r) = y(r) + values(j) * x\ (c) \\
&\qquad\qquad y(c) = y(c) + values(j) * x(r) \\
&\qquad\qquad end; \\
&end;
\end{aligned}$$

In our project the Sparse Matrix Vector Multiplication is parallelized on a shared-memory machine. The procedure involves splitting the sparse matrix into strips corresponding to the vector segments. Each processor then computes the matrix vector product of a single strip. This procedure can be visualized from Figure 5. The matrix vector multiplication is parallelized by row blocks so as to avoid inter-thread dependencies and obtain random access to input vector *x*. The pseudo-code for the parallel program of sparse Matrix Vector Multiplication with N threads is given below

$$\begin{aligned}
&for\ i = 1\ toN\ do\ in\ parallel \\
&\quad for\ r = start[i]\ to\ end[i] \\
&\qquad y[r] = diagvalues[r] * x[r] \\
&\qquad for\ j = rowptr[r]\ to\ \ rowptr[r+1] - 1 \\
&\qquad\qquad c = colind[j] \\
&\qquad y[r] = y[r] + values[j] * x[c]\ (upper\ triangular) \\
&\qquad\qquad critical\ (only\ 1\ thread\ at\ a\ time) \\
&\qquad y_i[c] = y_i[c] + values[j] * x[r] \quad (lower\ triangular) \\
&\qquad\qquad end; \\
&\quad end; \\
&end;
\end{aligned}$$

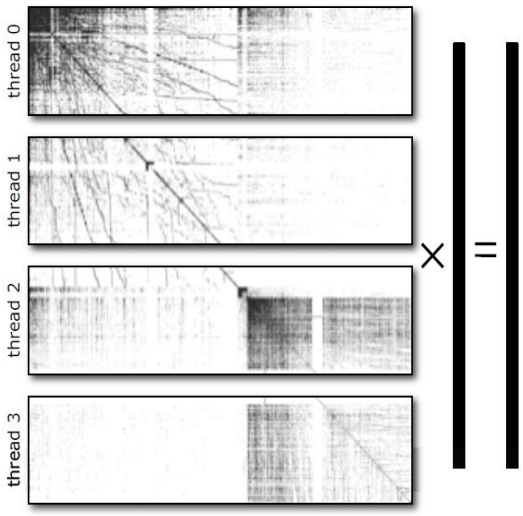

Figure 5: Parallelization of Sparse Matrix Vector Multiplication

IV. Experimental Setup

In this section, we present the details of our experimental setup. These include the details of the hardware platform used, descriptions of the test matrices and the specifics of our experimental approach. The parallel matrix solver designed was sufficiently tested for correctness, speedup and other design considerations.

A. Hardware Platform Specifics

The experiments were conducted on Intel® Core™ i5-4210U CPU with 2 cores and 4 logical processors operating at a peak frequency of 1.70 GHz. All the code was developed and compiled using 64-bit mode with the –O3 optimization enabled. The code was designed and executed on a Linux based operating system. Our project was built for shared parallel machines using the OpenMP API interface (GNU, GCC Compilers).

B. Test Matrices

This study is aimed to develop and test a parallel matrix solver based on iterative method specifically for symmetric

positive definite (SPD) matrices. Matrices are chosen from The University of Florida Sparse Matrix Collection [6]. The matrices were chosen from a variety of sizes and types in order to systematically study the convergence of conjugate gradient based solver and also optimize it suitably for parallel processing. A detailed list of the matrices which have been used in this paper for simulation are mentioned in Table 1.

C. Experimental Methodology

The overall methodology used for conducting the various simulations and collecting performance data for different matrices is described here.

- True solution of *x* is assumed to be 1.0 and is used to generate the corresponding RHS vector b.
- The initial guess of the solution for the iterative process is the zero vector.
- The conjugate gradient iterations are terminated when the relative residual norm drops to 10^{-8} or the number of iterations exceeds 100000.
- The time reported as the run time of the computation is the execution time for the conjugate gradient calculations. It is calculated as the average of five simulations run on the same matrix with similar configurations.

Matrix Name	Dimension	Non Zero Elements
apache1	80800	311492
BenElechi1	245874	6698185
cant	62451	2034917
cvxbqp1	50000	199984
denormal	89400	622812
ecology2	999999	2997995
G2_circuit	150102	438388
G3_circuit	1585478	4623152
s3rmt3m3	5357	106526
thermomech_TK	102158	406858
tmt_sym	726713	2903837

Table 1: SPD Test Matrices with their dimension and Number of Non-zero elements

V. Experimental Results

In this section the performance metrics of the developed parallel sparse matrix solver is described. Our performance evaluation is organized into four parts. In the first part, we evaluate the performance of our developed parallel solver with the time taken to complete the execution of the CG iterative solver. Second part discusses the speed of execution of the parallel solver is with respect to the amount of parallelism (number of threads) applied. In the third part, the performance of the serial and parallel versions of our solver is compared. The average memory consumed at different levels of parallelization is analyzed in the fourth part.

A. Time Taken for Execution

The time taken for execution of the CG solver with respect to different test matrices are shown in Table 2 and Table 3. The time taken for execution against the number of threads in the parallel implementation of the solver is also shown in Table 2 and Table 3.

Matrix	Dimension	Sequential (sec)	1 thread (sec)	2 thread (sec)
s3rmt3m3	5357	48.805	38.584	37.317
cvxbqp1	50000	2.672	2.475	2.206
cant	62451	115.870	109.541	101.256
apache1	80800	3.363	3.455	2.951
denormal	89400	122.213	83.986	65.874
thermomech_TK	102158	146.765	120.072	112.124
G2_circuit	150102	22.283	17.491	17.592
BenElechi1	245874	1541.22	1326.312	1256.066
tmt_sym	726713	208.830	180.028	164.639
ecology2	999999	206.241	188.438	177.674
G3_circuit	1585478	452.752	329.395	266.783

Table 2: Time taken for execution in seconds with respect to different test matrices and serial and parallel versions (against number of threads) of the solver

Matrix	Dimension	3 thread (sec)	4 thread (sec)	8 thread (sec)
s3rmt3m3	5357	47.983	59.385	57.485
cvxbqp1	50000	2.5185	2.935	2.559
cant	62451	120.811	123.309	106.146
apache1	80800	3.196	3.691	3.308
denormal	89400	104.890	126.571	99.280
thermomech_TK	102158	84.153	68.617	116.856
G2_circuit	150102	19.299	19.180	18.166
BenElechi1	245874	1310.979	1279.892	1236.209
tmt_sym	726713	171.930	165.823	161.146
ecology2	999999	188.209	178.076	169.491
G3_circuit	1585478	318.011	218.690	343.256

Table 3: Time taken for execution in seconds with respect to different Test matrices and number of threads of the solver

From Tables 2 and 3 we can infer that there is a considerable amount of performance benefit in the parallel version of the CG solver developed. The solver developed takes less amount of time to execute in the parallelized version for almost all test matrices.

B. Speedup

The speedup of a parallel program is computed by equation (6).

$$Speedup = \frac{T_{serial}}{T_{parallel}} \qquad (6)$$

The speedup obtained due to parallelism was analyzed against the number of threads for all the test matrices. The plots showing this analysis are depicted in Figures 6 and 7.

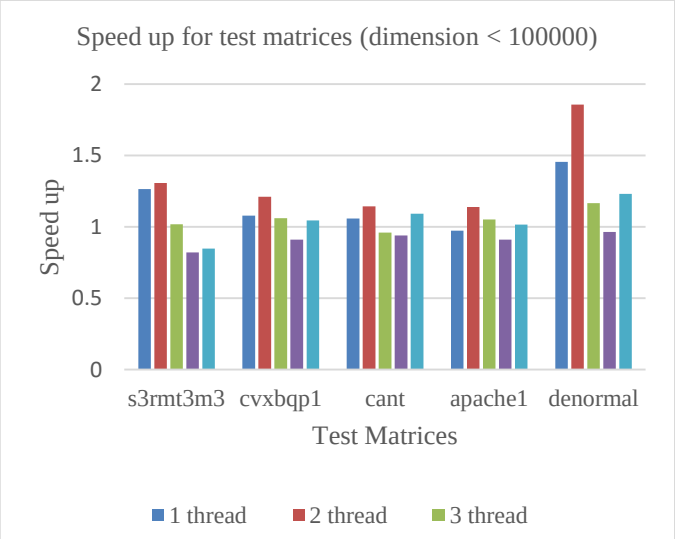

Figure 6: Graph depicting the speedup with increasing number of threads with respect to the test matrices whose dimensions are less than 100000.

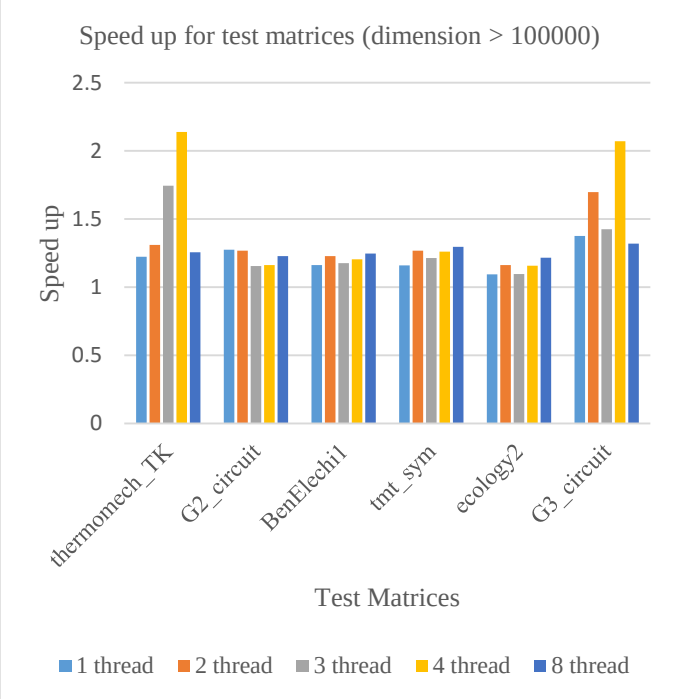

Figure 7: Graph depicting the speedup with increasing number of threads with respect to the test matrices whose dimensions are more than 100000.

From the plots it is observed that there is a significant speedup as the number of threads increase from 1 to 2 but as the threads increase to 3, 4 and 8, the speedup varies. This could be due to dominance of inter-thread communication and limited availability of the number of cores in the test machine.

A plot of maximum speedup obtained for each test matrices is shown in Figure 8.

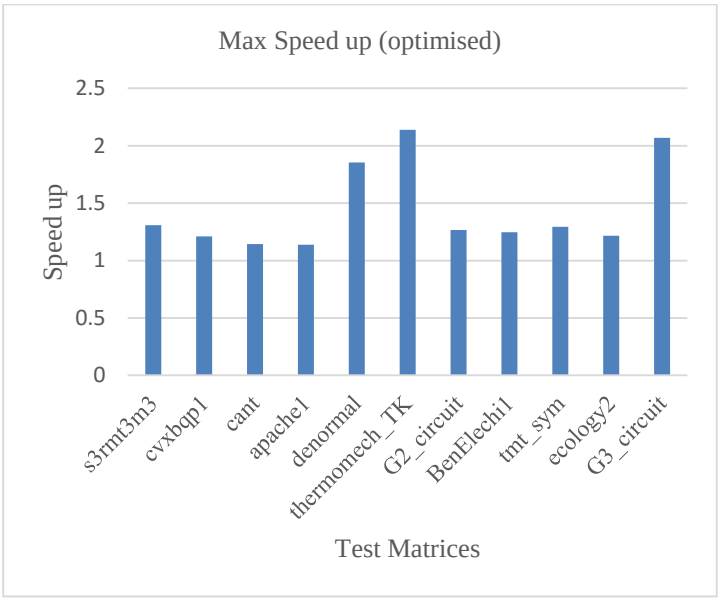

Figure 8: Maximum Speedup obtained by parallelization for different test matrices.

C. Comparison of Serial and Parallel Versions of the Solver

From Tables 2 and 3, it can be inferred that the parallel optimized version of the developed solver has a significant performance improvement when compared with the serial version of our developed solver. This can be visualized by the plot shown in Figure 9 which compares the time taken to execute the CG portion of the algorithm of the serial and optimized parallel versions of the developed solver.

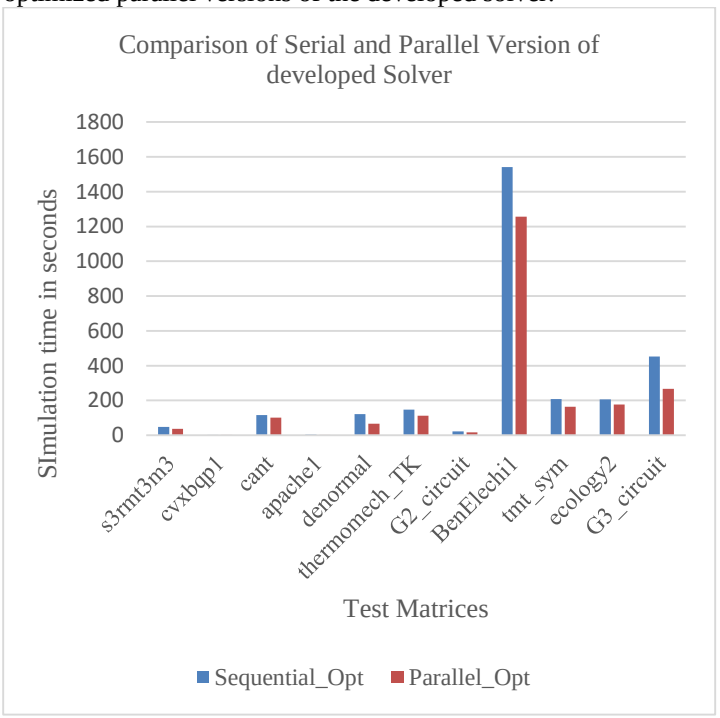

Figure 9: Graph showing performance improvement in parallel version of the developed solver with respect to the time taken for execution.

D. *Memory usage*

The memory usage of the developed parallel solver was analyzed and optimized for different stages of parallelism. The actual memory consumed by the processor for different number of threads is reported in Figure 10. We can observe that the speed up obtained through parallelization can easily overcome the increased memory requirements in multi-threaded programs running on computers with large memories.

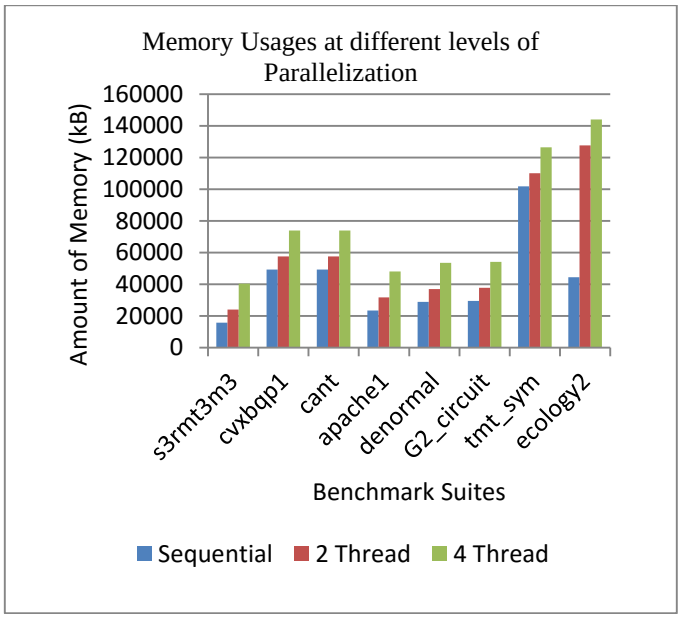

Figure 9: Graph showing the memory usages at different levels of parallelization.

VI. Conclusion and Future Scope

The objective of our project was to explore parallel sparse matrix solvers for large-scale linear circuit analysis. Sparse Matrix Solver using the method of Conjugate Gradient was developed. The iterative solver was later parallelized using the OpenMP API parallel computing platform for a shared memory model. Parallelizing the computation kernels involving inner products, vector updates and sparse matrix vector multiplication led to significant performance improvement. The symmetric properties of the sparse matrix were leveraged by storing the matrix in modified CSR format, which stored only half the non-zero entries (upper or lower triangular portion) and the diagonal entries of the matrix. The developed Conjugate Gradient Parallel Matrix Sparse Solver worked successfully with test matrices with dimensions > 1 million (ecology2 and G3_circuit). The performance metrics of the solver was evaluated with different types of test matrices from The University of Florida Sparse Matrix Collection [6]. Significant speedup and performance boost was obtained in the parallel version when compared with the serial version our solver.

As mentioned in Section 3, the convergence of our parallel solver depends on the spectral properties of the sparse matrix, in particular, the condition number of the matrix. The convergence of the solver can be further improved if an appropriate pre-conditioner is applied and the corresponding Preconditioned Conjugate Gradient Algorithm implemented. Also, the full potential of our parallel solver could not be realized as it was tested on personal laptops containing limited computational power when compared with Supercomputers. Further improvement in performance can be realized by exploring the techniques involving overlapping of computation with communication between threads when processing the sparse matrix in parallel.

References

[1] Peng Li, “Parallel Circuit Simulation: A Historical Perspective and Recent Developments,” Foundations and Trends in Electronic Design Automation: Vol. 5: No 4, pp 211-318, 2011 (invited).

[2] Saad, Yousef “Iterative methods for sparse linear systems” SIAM, 2003.

[3] Saleh, Resve A., "Parallel circuit simulation on supercomputers", Proceedings of the IEEE 77.12 (1989): 1915-1931.

[4] Z. Zeng, Z. Feng, P. Li, and V. Sarin, “Locality-driven parallel static analysis for power delivery networks,” ACM Transactions on Design Automation Electronic Systems, vol. 16, pp. 28:1–28:17, June 2011.

[5] Z. Zeng, P. Li, and Z. Feng, “Parallel partitioning based on-chip power distribution network analysis using locality acceleration,” in Quality of Electronic Design (ISQED 2009) Quality Electronic Design, pp. 776–781, March 2009.

[6] The University of Florida Sparse Matrix Collection, T. A. Davis and Y. Hu, ACM Transactions on Mathematical Software, Vol 38, Issue 1, 2011, pp 1:1 - 1:25

[7] Bolz Jeff, Farmer Ian, Grinspun "Sparse matrix solvers on the GPU: conjugate gradients and multigrid." *ACM Transactions on Graphics (TOG)*. Vol. 22. No. 3. ACM, 2003.

[8] Yang, Chao-Tung, Chih-Lin Huang, and Cheng-Fang Lin. "Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters." *Computer Physics Communications* 182.1 (2011): 266-269.

[9] W. Samuel, Oliker, James Demmel "Optimization of sparse matrix–vector multiplication on emerging multicore platforms." Parallel Computing 35.3 (2009): 178-194.

[10] D’Azevedo, Eduardo F., Mark R. Fahey, and Richard T. Mills. "Vectorized sparse matrix multiply for compressed row storage format." *Computational Science–ICCS 2005*. Springer Berlin Heidelberg, 2005. 99-106.

[11] Hestenes, Magnus Rudolph, and Eduard Stiefel. “*Methods of conjugate gradients for solving linear systems”*. Vol. 49. NBS, 1952.

[12] Freund, Roland W. "Conjugate gradient-type methods for linear systems with complex symmetric coefficient matrices." SIAM Journal on Scientific and Statistical Computing 13.1 (1992): 425-448.

[13] Mohiyuddin, Marghoob, et al. "Minimizing communication in sparse matrix solvers." Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. ACM, 2009.

[14] Gupta, A.,"WSMP: Watson sparse matrix package (Part-III: iterative solution of sparse systems)." SIAM Journal on Scientific Computing 32.1 (2010): 84-110.

[15] T. Gkountouvas, V. Karakasis, K. Kourtis, G. Goumas and N. Koziris, "Improving the Performance of the Symmetric Sparse Matrix-Vector Multiplication in Multicore," *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, Boston, MA, 2013, pp. 273-283.

[16] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix vector multiplication on emerging multicore platforms. In Proc. 2007 ACM/IEEE Conference on Supercomputing, 2007.

[17] Nathan Bell and Michael Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA", NVIDIA Technical Report NVR-2008-004, December 2008.

[18] Pacheco, Peter, “An Introduction to Parallel Programming”, Elsevier, 2011.