# Developing object-oriented parallel iterative methods

**Chakib Ouarraui and David Kaeli***
Department of Electrical and Computer Engineering,
Northeastern University,
Boston, MA 02115
E-mail: ochakib@ece.neu.edu E-mail: kaeli@ece.neu.edu
*Corresponding author

**Abstract:** In this paper, we describe our work developing an object-oriented parallel toolkit on OO-MPI. We are developing a parallelised implementation of the multi-view tomography toolbox, an iterative solver for a range of tomography problems. The code is developed in object-oriented C++. The MVT toolbox is presently used by researchers in the field of tomography to solve linear and non-linear forward modelling problems. The performance of the toolbox is heavily dependent on the performance of a small number of classes present in the IML++ library. This paper describes our experience parallelising a sparse matrix algorithm provided in the IML++ object-oriented numerical library. This library comprises a number of iterative algorithms. In this work, we present a parallel version of BiCGSTAB algorithm and the block-ILU preconditioner. These two algorithms are implemented in C++ using OOMPI (object-oriented MPI), and run on a 32-node Beowulf cluster. These two routines are also fundamental to obtaining good performance when using the parallelised version of the MVT toolbox. We also demonstrate the importance of using threads to overlap communication and computation, as an effective path to obtain improved speedup.

**Keywords:** parallel computing; OOMPI; iterative methods; IML++.

**Reference** to this paper should be made as follows: Ouarraui, C. and Kaeli, D. (2004) 'Developing object-oriented parallel iterative methods', *Int. J. High Performance Computing and Networking*, Vol. 1, Nos. 1/2/3, pp.85–90.

**Biographical notes:** Chakib Ouarraoui received a BS in computer engineering from North Eastern University, Boston, MA, in 2001. He is presently an MS student in the Department of Electrical and Computer Engineering in Northeastern University. He is presently working as a Research Assistant in the Northeastern University Computer Architecture Research Laboratory (NUCAR). His research interests include cluster computing and object-oriented design for clusters.

David Kaeli received a BS and PhD in electrical engineering from the Rutgers University, and an MS in computer engineering from the Syracuse University. He is presently an Associate Processor on the ECE faculty at Northeastern University, Boston, MA. Kaeli has published over 100 critically reviewed publications, three books, four patents, and is presently co-editing a book on speculative execution. He is an Associate Editor of *IEEE Transactions on Computers*, *IEEE Computer Architecture Letters*, the *Journal of Instruction-Level Parallelism.*

## 1 INTRODUCTION

Developing object-oriented programs on a parallel programming environment is an emerging area of research. As we begin to utilise an object-oriented programming paradigm, we will need to rethink how we develop libraries in order to maintain the encapsulation and polymorphism provided by the language. In this paper, we develop parallel versions of methods associated with the IML++ library (Dongarra et al., 1994). Our goal is to get an efficient implementation of the library that can be run on a Beowulf cluster. The IML++ library is C++ code that contains a number of iterative algorithms such as biconjugate gradient, generalised minimal residual, and quasi-minimal residual. In order to develop a parallel version of the IML++ library, each individual algorithm needs to be parallelised.

To develop a parallel implementation of an object-oriented library, we need a middleware that works

seamlessly with an object-oriented programming paradigm. OOMPI is an object-oriented message passing interface (Squyres et al., 1996). The code provides a class library that encapsulates the functionality of MPI into a functional class hierarchy, providing a simple and flexible interface. We will use OOMPI as we attempt to parallelise the MVT toolbox on a Beowulf cluster.

This paper is organised as follows. In Section 2, we describe the multi-view tomography toolbox. In Section 3, we discuss the design of the IML++ library. In Section 4, we go into the details of the parallel BiCGSTAB algorithm and the ILU preconditioner. In Section 5, we discuss the issue of scalability as a function of the amount of communication inherent in the code. Finally in Section 6, we summarise the paper and discuss directions for future work.

## 2 MVT TOOLBOX

The MVT toolbox provides researchers in the many tomography fields with a collection of generic tools for solving linear and non-linear forward modelling problems (CenSSIS, 2003). The toolbox is designed for flexibility in establishing parameters to provide the ability to tailor the tool to specific tomography problem domains. This library provides a numerical solution to the partial differential equation of the form

$$\nabla\cdot\sigma(r)\nabla\phi(r) + k^2(r)\phi(r) = s(r), \quad (1)$$

where $r$ is a point in 3-space; $\sigma(r)$ is the conductivity and can be any real-valued function of space; $k^2(r)$ is the wave number and can be any complex-valued function of space; $\phi(r)$ is the field for which we are solving; s(r) is the source function; and the boundary conditions can be Dirichlet, Neumann, or mixed.

The MVT toolbox is designed as a numerical solver for problems of the general form defined by equation (1). The toolbox can solve this problem on a regular or semi-regular cartesian grid. It has the ability to model arbitrary sources as well as receivers. It also can use a finite-difference approach for discretisation of the equation defined before and associated boundary conditions in 3-D space. In this case, the problem of solving the differential equation for the field is reduced to solving a large sparse system of linear equations of the general form $Af = s$, where the matrix A represents the finite-difference discretisation of the differential operator, as well as the boundary conditions. The vector f contains the sample values of the field at the points on the grid. The vector s holds the discretised source function. IML++ package is used as the main solver for the equation $Ax = b$.

## 3 IML++ LIBRARY

The IML++ package (Dongarra et al., 1994) is a C++ version of set of iterative methods routines. IML++ was developed as part of the Templates Project (Barrett et al., 1994). This package relies on the user to supply matrix, vector, and preconditioner classes to implement specific operations. The IML++ library includes a number of iterative methods that support the most commonly used sparse data formats. The library is highly templatised such that it can be used on different sparse matrix formats, together with a range of data formats. The library is described in Barrett et al. (1994) and contains the following iterative methods:

- conjugate gradient (CG)
- generalised minimal residual (GMRES)
- minimum residual (MINRES)
- quasi-minimal residual (QMR)
- conjugate gradient squared (CGS)
- biconjugate gradient squared (BiCG)
- biconjugate gradient stabilised (Bi-CGSTAB)

## 4 PARALLEL MVT

To develop a parallel implementation of the MVT toolbox, we have utilised a *profile-guided* approach. The main advantage of this technique is that there is no need to delve into the details of the library (source code for the library may not be available). Profiling allows us to concentrate only on the hot portions of the code.

The first step in our work is to profile the serial code on a uni-processor machine. We will then develop a parallel version of the hot code.

### 4.1 Profiling the MVT toolbox

To profile the toolbox, we used the technique described in Ashouei et al. (2002) and El-Shenawee et al. (2002). A similar technique has been used to profile IO accesses (Wang and Kaeli, 2003). Using this technique, the serial code is profiled on single processor machine. The GNU utility *gprof* was used to profile the library. The results show that most of execution time is spent in the iterative method library (IML++). So we chose to focus our parallelisation work on the IML++ library.

### 4.2 Parallelisation of object-oriented code

The IML++ library code computes solutions for different sources and frequencies. Therefore, the easiest way to parallelise the toolbox is parallelise on a coarse grain and compute each source on a node of the cluster. However, in general, the number of modelled sources is less than the number of processors; hence, the available resources will be underutilised. Another option is to follow a hybrid approach where each source is solved on a group of processors. While this approach obtains good results, it is still necessary to parallelise the IML++ library.

### 4.3 Parallel IML++

To begin to parallelise the IML++ library, we need to consider the various iterative algorithms included with the

package. Each individual algorithm needs to be parallelised. In this section, we explore a parallel implementation of the biconjugate gradient stabilised algorithm (BiCGSTAB), as well as a parallel version of the incomplete LU factorisation code.

#### 4.3.1 BiCGSTAB algorithm

The BiCGSTAB algorithm is a Krylov subspace iterative method for solving a large non-symmetric linear system of the form $Ax = b$. The algorithm extracts approximate solutions from the Krylov subspace. BiCGSTAB is similar to the BiCG algorithm except that BiCGSTAB provides for faster convergence. The pseudocode for the BiCGSTAB algorithm is provided in Figure 1.

$1 : r_0 = b - Ax_0$
$2 : \rho_0 = \alpha_0 = \omega_0 = 1;$
$3 : v_0 = p_0 = 0;$
$4 :$ for n $= 1, 2, 3,$...do5 :
$5 : \quad \rho_n = r_0^T r_{n-1};$
$6 : \quad \beta = \frac{\rho_n}{\rho_{n-1}} \frac{\alpha_n}{\omega_{n-1}};$
$7 : \quad p_n = r_n - 1 + \beta_n(p_{n-1} - \omega_n - 1 v_{n-1});$
$8 : \quad v_n = Ap_n;$
$9 : \quad \alpha_n = \frac{\rho_n}{r_0^T v_n};$
$10 : \quad s_n = r_{n-1} - \alpha_n v_n;$
$11 : \quad t_n = As_n;$
$12 : \quad \omega_n = \frac{t_n^T s_n}{t_n^T t_n};$
$13 : \quad x_n = x_{n-1} + \alpha_n p_n + \omega_n s_n;$
$14 : \quad if\ x_n\ is\ accurate\ enough\ then$
$15 : \quad$ STOP
$16 : \quad$ endif
$17 : \quad r_n = s_n - \omega_n t_n;$
$18 :$ endfor

**Figure 1** *The BiCGSTAB algorithm*

#### 4.3.2 Parallel BiCGSTAB algorithm

Iterative algorithms can be challenging to parallelise. In the BiCGSTAB algorithm, there is a loop-carried dependence in the algorithm. If we attempt to parallelise the algorithm using this form of the code, a synchronisation barrier is required after each iteration; hence, most of the time will be spent synchronising on shared access to the data. There has been related research on minimising the number of synchronisations. Following the approach described by Yang and Brent (2002), the BiCGSTAB algorithm can be restructured so that there is only one global synchronisation by redefining and splitting the equations of the original algorithm. For example, vn can be written as

$$v_n = Ap_n = A(r_{n-1} + \beta_n(p_{n-1} - \omega_{n-1}v_{n-1}))$$

$$= u_{n-1} + \beta_n v_{n-1} - \beta_n \omega_{n-1} q_{n-1})$$

The same thing can be done for all vectors so that $\phi n = r_0 s_n$, and $\tau n = r_0 v_n$.

These code transformations make the task of parallelisation more straightforward. A modified version of the BiCGSTAB algorithm is presented in Figure 2. In the modified BiCGSTAB algorithm, we can clearly see that the inner products in steps 13–18 do not involve loop-carried dependencies. The vector updates in lines 10 and 11 are independent, and those in lines 21 and 22 are also independent.

$1 : r_0 = b - Ax_0, u0 = Ar_0, f_0 = A^T r_0, q_0 = v_0 = z_0 =$
$2 : \sigma_{-1} = \pi_0 = \phi_0 = \tau_0 = 0, \sigma_0 = r_0^T u_0, \rho_0 = \alpha_0 = \omega_0 =$
$3 : for n = 1, 2, 3, ...do$
$4 : \quad \rho_n = \phi_{n-1} - \omega_{n-1}\sigma_{n-2} + \omega_{n-1}\alpha_{n-1}\pi_{n-1};$
$5 : \quad \delta_n = \frac{\rho_n}{\rho_{n-1}}\alpha_{n-1}, \beta = \frac{\delta_n}{\omega_{n-1}};$
$6 : \quad \tau_n = \sigma_{n-1} + \beta_n \tau_{n-1} - \delta_n \pi_{n-1};$
$7 : \quad \alpha_n = \frac{\rho_n}{\tau_n};$
$8 : \quad v_n = u_{n-1} + \beta_n v_{n-1} - \delta_n q_{n-1};$
$9 : \quad q_n = Av_n;$
$10 : \quad s_n = r_{n-1} - \alpha_n v_n;$
$11 : \quad t_n = u_{n-1} - \alpha_n q_n;$
$12 : \quad z_n = \alpha_n r_{n-1} + \beta_n z_{n-1} - \alpha_n \delta_n v_{n-1};$
$13 : \quad \phi_n = r_0^T s_n;$
$14 : \quad \pi_n = r_0^T q_n;$
$15 : \quad \gamma_n = f_0^T s_n;$
$16 : \quad \eta_n = f_0^T t_n;$
$17 : \quad \theta_n = s_n^T t_n;$
$18 : \quad \kappa_n = t_n^T t_n;$
$19 : \quad \omega_n = \frac{\theta_n}{\kappa_n};$
$20 : \quad \sigma_n = \gamma_n - \omega_n \eta_n;$
$21 : \quad r_n = s_n - \omega_n t_n;$
$22 : \quad x_n = x_{n-1} + z_n + \omega_n s_n;$
$23 : \quad if\ x_n\ is\ accurate\ enough\ then$
$24 : \quad STOP$
$25 : \quad end\ if$
$26 : \quad u_n = Ar_n;$
$27 : end\ for$

**Figure 2** *Parallel BiCGSTAB algorithm*

#### 4.3.3 Implementation of parallel BiCGSTAB:

IML++ is a highly templatised library that provides the user with the ability to reuse the code for different problem classes and data formats. To maintain this polymorphism when moving to a parallel environment, some modifications needed to be made to the structure of the MVT toolbox. The parallel implementation of the BiCGSTAB algorithm requires the addition of new methods to the vector and matrix classes. These methods basically set the range for the computations. Making these changes will help to ensure the efficiency and reusability of the classes. We have other options available that still maintain the object-oriented nature of that code and these options would not impact main library. One solution would be to cast the input vector and matrix into a newly defined class that contains the needed methods. However, this would introduce additional memory overhead. Another option would be to extract the data from the input vectors and matrices, but this would add additional processing overhead on each access to the data.

So the option we arrived at requires that the input vector and matrix classes provide a new class method that defines

the *range* for the computation. Our selection was made based on the need to keep the BiCGSTAB algorithm highly reusable.

We developed an initial parallelised version of the code using OO-MPI and ran it on our 32-node Beowulf cluster (using only 24 nodes). Our initial performance results were somewhat surprising, only obtaining a 2X speedup on a 16-node configuration (see Figure 3).

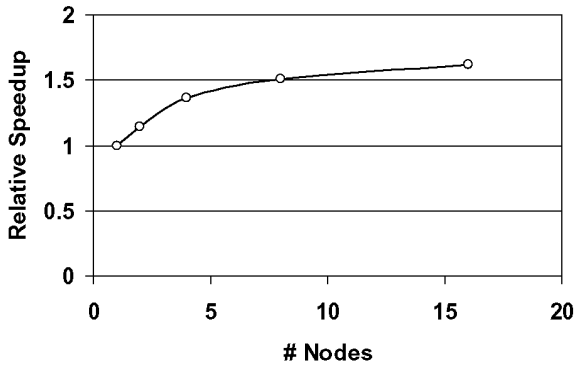


**Figure 3** *Speedup of BiCGSTAB*

To better understand the performance bottlenecks in this code, we used *method-level profiling*. The goal was to identify both the hot portions of the program (based on the number of method calls) and the methods where the program spends most of its execution. Figure 4 shows the percentage of time spent in ILU while varying the number of compute nodes. We found that the preconditioner consumes a significant amount of the execution, so we next parallelised the ILU preconditioner code.

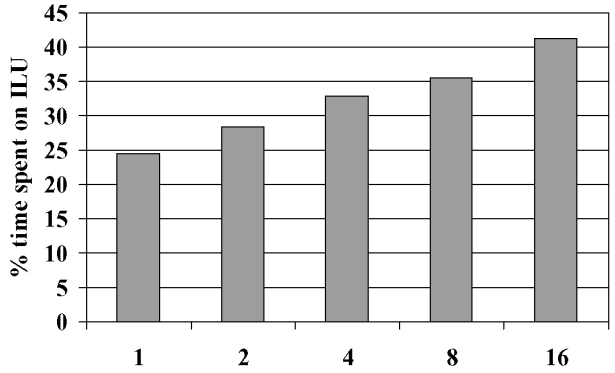


**Figure 4** *Profiling of the ILU preconditioner code*

#### 4.3.4 ILU preconditioner

Preconditioners are used to accelerate convergence of an algorithm. The basic idea is to find a preconditioner for a linear system $Ax = b$ that is a good approximation to A so that the same system is much easier to solve than the original system. Mathematically, the system can be solved using: $M^{-1}Ax = M^{-1}b$ by any iterative method. The preconditioner is not always easy to find. There are many techniques used to arrive at the best preconditioner. Incomplete factorisation preconditioners are commonly used since they are suitable for any type of matrix. The pseudocode of the ILU algorithm is given in Figure 5.

**Algorithm IV.1:** ILU$(M, x, D)$

**for** $i \leftarrow 1$ **to** $n$
  **for** $j \leftarrow 1$ **to** $i$
    $v = v + A(i,j) * r(i)$
  $v = x(i) - v$
**for** $i \leftarrow n-1$ **to** $1$
  **for** $j \leftarrow n-1$ **to** $i$
    $v+ = A(i,j) * w(i)$
  $v = r(i) - v$
  $v = v * D(i)$
  $w(i) = v$

**Figure 5** *The ILU algorithm*

#### 4.3.5 Parallel ILU preconditioner

There are a number of parallel algorithms that implement the ILU method. However, it is hard to find a parallel ILU algorithm that is scalable. Most ILU methods are sequential in nature and thus they are difficult to parallelise on a cluster. In this paper, we utilise the *block-ILU algorithm* described in Gonzalez et al. (1999).

The block ILU chosen splits the matrix into blocks, assigning each block to a processor. This technique involves no synchronisation between processors, which makes it potentially scalable on clusters. The block parameter that defines the size of the overlap region is controllable. The matrix is divided into overlap blocks, and each block is stored in a single processor. Figure 6 describes this partitioning method. In this example, the original matrix is partitioned into four overlap blocks. Each block is assigned to a processor. The size of the overlap region is controllable, which helps the user to search for a better solution. The size of the overlap region is very important since a large overlap region will affect the performance of the code significantly. Using a small overlap will affect the quality and the convergence of the solution.

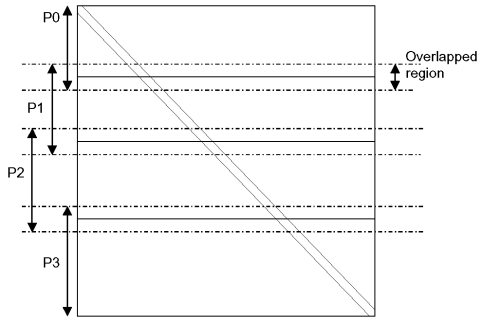


**Figure 6** *Block-ILU algorithm*

Now that we have described the parallel BiCGSTAB and parallel ILU preconditioner, we need to run our parallel implementation on our cluster. Runtime results are shown in Figure 7. The parallelised version runs much faster and

obtains better scalability. The speedup is close to four on an eight processor machine (though degrades when additional processors are used). This is not the result that we were hoping for. We hoped to get linear scalablity. So we continued to profile the code, looking for additional opportunities for tuning.

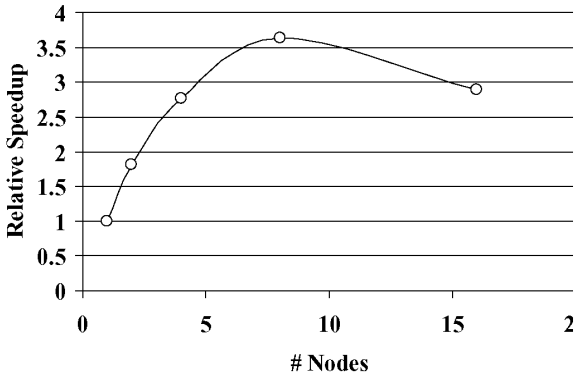


**Figure 7** *Speedup of the BiCGSTAB using block-ILU*

We next focus on the communication overhead present in the code. To obtain this profile, we inserted trace points into the code before and after each synchronisation point. We obtained a profile of the total communication time and the communication time for each synchronisation point.

Looking at the results in Figure 8, we can see that communication time for 2, 4, 8, 16 is about 20%, 30%, 35%, 44% of the total time. Obviously, we have scalability issues here. To address this problem, we need to hide some of this communication. One solution is to overlap communication with computation. However in our case, we are using a blocking *Allreduce()* method. A non-blocking version of the *Allreduce()* method is not provided. By using a non-blocking version, we expect the speedup to be closer to 8 on a 16 processor system.

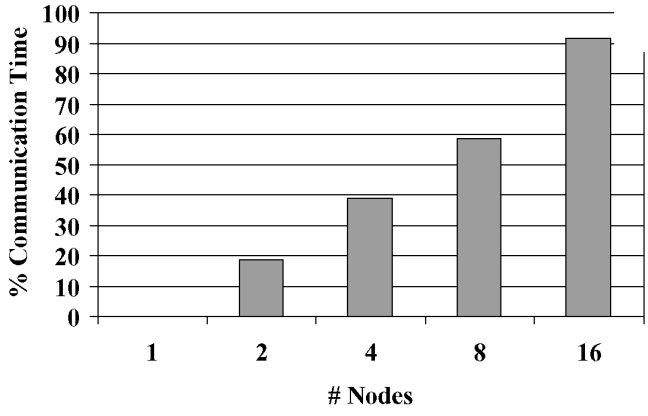


**Figure 8** *Communication profile for the code*

#### 4.3.6 Threaded communication

In our work, we have found that there exist scalability issues, which are caused by the communication overhead. A straightforward approach to address this problem is to allow overlap the communication and computation. In order to implement this solution, we needed to develop a number of non-blocking functions, specifically the *Allreduce()* and *Allgather()* functions. Rewriting the *Allreduce()* call using Isend and Irecv is not a good solution since there is a need to synchronise all processors at each processing step (i.e., too much communication). Next, we describe our solution to this problem using threads.

Our solution is to spawn a communication thread on each processor that only handles MPI calls such as *Allreduce()* and *Allgather()*. Once the communication thread is started, it goes to sleep and waits for the main process to issue a wakeup signal. The main process spawns a wakeup communication thread, which runs in parallel to the main process that is performing independent computation. Once the main process is done with the computation, it waits for the communication thread to finish communication with the other processors. This is an efficient solution since the communication thread will most likely be sleeping, waiting for a command from the main process, or waiting for an MPI call to complete. However, there is also the overhead associated with thread creation and synchronisation with the main process. If the application involves a significant amount of computation, higher performance will result using our thread-based synchronisation scheme. On the other hand, if the application is communication dominated, then the main process will be stalled waiting for communication to complete.

As shown in Figure 9, we obtained improved speedup for the threaded version of the parallelised code, up to eight nodes. However, when using 16 nodes, the execution is still dominated by communication. Since the problem is divided over more nodes, the problem size is smaller, so less time is spent on computation. Also, communication overhead will be higher as we add computational nodes, and additional overhead for thread synchronisation will result.

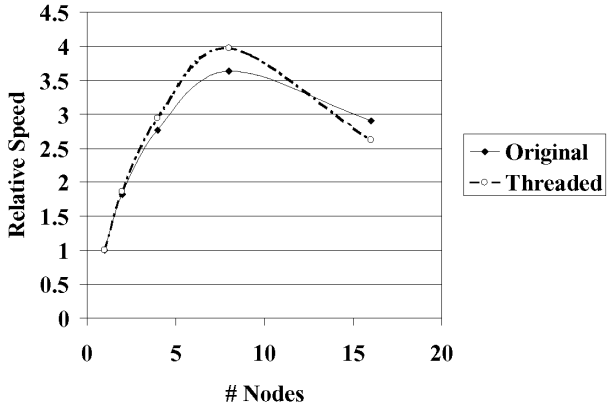


**Figure 9** *Original vs. threaded version speedup*

## 5 DISCUSSION

It should be clear from our results that this algorithm is not scalable on the system used in our performance study. This cluster utilises 100 megabit/second switched (i.e., fast) ethernet as a communication fabric. It is clear that

communications are the bottleneck for this algorithm. We are presently moving our work to a gigabit cluster system, and expect to obtain better speedup. The latency of gigabit ethernet is very similar to that of fast ethernet; they only differ in bandwidth (there would be little difference in performance if the application is only exchanging small messages in the absence of message collisions). In our case, the algorithm also needs to gather a vector on all processors, so we should expect a performance increase. To estimate the total time of the communication overhead, we developed a model 2, where T is the total time, T0 is the latency, M the message size in bytes, and r1 is the bandwidth.

$$T = T_0 + \frac{M}{r_\infty} \tag{2}$$

From the algorithm, we know the size of each message and the frequency of each MPI call; hence, we can estimate the total time due to communication overhead. In this estimate, we assume that *Allgather()* and *Allreduce()* are executed in four steps on a 16-node cluster. Applying these values to our model (equation (2)) gives us 34 sec for fast ethernet and 3.34 sec for gigabit ethernet. We clearly can see that the experimental results for fast ethernet are very close to the results obtained from our model. This suggests that the communication overhead obtained when we move to a gigabit network will be one-tenth of the amount seen when using fast ethernet. This would provide us with a speedup of near ten on a 16-node cluster. Since gigabit-connected clusters will shortly become the de facto network fabric, our parallel library will provide users with effective pathway to obtain scalable performance.

## 6 SUMMARY

In this paper, we presented the parallelisation of an object-oriented library that implements the BiCGSTAB and ILU algorithms. This work is still in progress.

We have shown the importance of using threads to overlap communication and computation. Our initial results on fast ethernet showed that these algorithms are inherently dominated by communications. However, we have studied the amount of communication in these algorithms and predict that improved scalability will be obtained on a gigabit-connected cluster. We are presently working on parallel versions of the CG, BiCG, GMRES, and QMR algorithms, and how best to integrate these into a parallelised IML++ library. We have shown that the parallel iterative method library is both an object-oriented interface to speed code development and also will be able to obtain reasonable parallel speedup on evolving network fabrics. We plan to explore using blocked versions of some the algorithms, which initially appear to be a promisingdirection for our work. We also plan to improve the design, targeting increased flexibility and reusability.

## ACKNOWLEDGEMENTS

This work is funded by CenSSIS, the Center for Subsurface Sensing and Imaging Systems, under the Engineering Research Centers Program of the NSF (Award Number EEC-9986821). This work was also supported by a grant from the Center of Complex Scientific Software at the Northeastern University. The authors would also like to acknowledge the contributions of Kyle Guibert, Eric Miller, Dana Brooks, and Jennifer Black on the design and implementation of the MVT toolbox.

## REFERENCES

Ashouei, M., Jiang, D., Meleis, W., Kaeli, D., El-Shenawee, M., Mizan, E., Wang, Y., Rappaport, C. and Dimarzio, C. (2002) 'Profile-based characterization and tuning for subsurface sensing and imaging applications', *Journal of Systems, Science and Technology*, pp.40–55.

Barrett, R., Berry, M., Chan, T-F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. and van der Vorst, H. (1994) *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, PA.

CenSSIS (2003) 'The multi-view tomography toolbox', *Center for Subsurface Sensing and Imaging Systems*, URL: http://www.censsis.neu.edu/MVT/mvt.html.

Dongarra, J., Lumsdaine, A., Niu, X., Pozo, R. and Remington, K. (1994) 'A sparse matrix library in C++ for high performance architectures', *Proceedings of the 2nd Object Oriented Numerics Conference*, pp.214–218.

El-Shenawee, M., Rappaport, C., Jiang, D., Meleis, W. and Kaeli, D. (2002) 'Electromagnetics computations using the MPI parallel implementation of the steepest descent fast multipole method (SDFMM)', *ACES Journal*, Vol. 17, No. 2, pp.112–122.

Gonzalez, P., Cabaleiro, J.C. and Pena, T.F. (1999) 'Parallel incomplete LU factorization as a preconditioner for Krylov subspace methods', *Parallel Processing Letters*, Vol. 9, No. 4, pp.467–474.

Squyres, J., McCandless, B. and Lumsdaine, A. (1996) 'Object oriented MPI: a class library for the message passing interface', *Proceedings of the POOMA Conference.*

Wang, Y. and Kaeli, D. (2003) 'Profile-guided I/O partitioning', *Proceedings of the International Conference on Supercomputing*, pp.252–260.

Yang, L.T. and Brent, R.P. (2002) 'The improved BiCGStab method for large and sparse unsymmetric linear systems on parallel distributed memory architectures', *Proceedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP-02, Beijing, pp.324–328.