Developing an Object-oriented Parallel Iterative-Methods Library

Chakib Ouarraoui
David Kaeli
Department of Electrical and Computer Engineering
Northeastern University
ochakib,kaeli@ece.neu.edu

Abstract

In this paper we describe our work developing an object-oriented parallel toolkit on OO-MPI. We are developing a parallelized 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 modeling problems. This paper describes our experience parallelizing an object-oriented numerical library, which 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), run on a 32-node Beowulf cluster. 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, multi-view tomography, iterative methods.

1 Introduction

Developing object-oriented programs on a parallel programming environment is an emerging area of research. As we begin to utilize 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 [3]. 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, Generalized Minimal Residual, and Quasi-Minimal Residual. In order to develop a parallel version of the IML++ library, each individual algorithm needs to be parallelized.

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 [7]. 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 parallelize the MVT Toolbox on a Beowulf cluster.

This paper is organized as follows. In Section 2 we will describe the Multi-view Tomography Toolbox. In Section 3 we will discuss the design of the IML++ library. In Section 4 we will 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 summarize 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 modeling problems [6]. 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

$$\nabla \cdot \sigma(r) \nabla \phi(r) + k^2(r) \phi(r) = s(r),$$  \hspace{1cm} (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 discretization 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 discretization 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 discretized source function. IML++ package is used as the main solver for the equation $Ax = b$.

3 IML++ Library

The IML++ package [3] is a C++ version of set of iterative methods routines. IML++ was developed as part of the Templates Project [2]. 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 templated such that it can be used on different sparse matrix formats, together with a range of data formats. The library is described in [2], and contains the following iterative methods:

- Conjugate Gradient (CG)
- Generalized Minimal Residual (GMRES)
- Minimum Residual (MINRES)
- Quasi-Minimal Residual (QMR)
- Conjugate Gradient Squared (CGS)
- Biconjugate Gradient Squared (BiCG)
- Biconjugate Gradient Stabilized (Bi-CGSTAB)
4 Parallel MVT

To develop a parallel implementation of the MVT Toolbox, we have utilized 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 [1, 4]. A similar technique has been used to profile IO accesses [8]. 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 parallelization work on the IML++ library.

4.2 Parallelization of Object-oriented Code

The IML++ library code computes solutions for different sources and frequencies. Therefore the easiest way to parallelize the toolbox is parallelize on a coarse grain, and compute each source on a node of the cluster. However, in general, the number of modeled sources is less than the number of processors, so the available resources will be underutilized. 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 parallelize the IML++ library.

4.3 Parallel IML++

To begin to parallelize the IML++ library, we need to consider the various iterative algorithms included with the package. Each individual algorithm needs to be parallelized. In this section we will explore a parallel implementation of the Biconjugate Gradient Stabilized algorithm (BiCGSTAB), as well as a parallel version of the Incomplete LU factorization 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.

4.3.2 Parallel BiCGSTAB Algorithm

Iterative algorithms can be challenging to parallelize. In the BiCGSTAB algorithm, there is a loop-carried dependence in the algorithm. If we attempt to parallelize the algorithm using this form of the code, a synchronization barrier is required after each iteration, so most of the time will be spent synchronizing on shared access to the data. There has been related research on minimizing the number of synchronizations. Following the approach described by Laurence [9], the BiCGSTAB algorithm can be restructured so that there is only one global synchronization by redefining and splitting the equations of the original algorithm. For example, $v_n$ can be written as
1: \[ r_0 = b - Ax_0 \]
2: \[ \rho_0 = \alpha_0 = \omega_0 = 1; \]
3: \[ v_0 = p_0 = 0; \]
4: for \( n = 1, 2, 3, \ldots \) do:
5: \[ \rho_n = r_0^T r_{n-1}; \]
6: \[ \beta = \frac{\rho_n}{\rho_{n-1} \omega_{n-1}}; \]
7: \[ p_n = r_n - 1 + \beta_n (p_{n-1} - \omega_n - 1 v_{n-1}); \]
8: \[ v_n = A p_n; \]
9: \[ \alpha_n = \frac{p_n}{r_n^T s_n}; \]
10: \[ s_n = r_{n-1} - \alpha_n v_n; \]
11: \[ t_n = A s_n; \]
12: \[ \omega_n = \frac{t_n^T s_n}{r_n^T s_n}; \]
13: \[ x_n = x_{n-1} + \alpha_n p_n + \omega_n s_n; \]
14: if \( x_n \) is accurate enough then
15: STOP
16: endif
17: \[ r_n = s_n - \omega_n t_n; \]
18: endfor

Figure 1: The BiCGSTAB Algorithm.

\[ v_n = A p_n = A(r_{n-1} + \beta_n (p_{n-1} - \omega_{n-1} v_{n-1})) \]
\[ = u_{n-1} + \beta_n v_{n-1} - \beta_n \omega_{n-1} q_{n-1} \]

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 parallelization 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 the vector updates in lines 21 and 22 are also independent.

4.3.3 Implementation of Parallel BiCGSTAB

IML++ is a highly templatized 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
1: $r_0 = b - Ax_0, u_0 = Ar_0, f_0 = A^T r_0, q_0 = v_0 = z_0 = 0$;
2: $\sigma_{-1} = \pi_0 = \phi_0 = r_0 = 0, \sigma_0 = r_0^T u_0, \rho_0 = \alpha_0 = \omega_0 = 1$;
3: for $n = 1, 2, 3, \ldots$ do
4: $\rho_n = \phi_{n-1} - \omega_{n-1} \sigma_{n-2} + \omega_{n-1} \alpha_{n-1} \pi_{n-1}$;
5: $\delta_n = \frac{\rho_n}{\rho_{n-1}} \alpha_{n-1}; \beta = \frac{\delta_n}{\omega_{n-1}}$;
6: $\tau_n = \sigma_{n-1} + \beta_n \tau_{n-1} - \delta_n \pi_{n-1}$;
7: $\alpha_n = \frac{\phi_n}{\tau_n}$;
8: $v_n = u_{n-1} + \beta_n v_{n-1} - \delta_n q_{n-1}$;
9: $q_n = A v_n$;
10: $s_n = r_{n-1} - \alpha_n v_n$;
11: $t_n = u_{n-1} - \alpha_n q_n$;
12: $z_n = \alpha_n r_{n-1} + \beta_n z_{n-1} - \alpha_n \delta_n v_{n-1}$;
13: $\phi_n = r_0^T s_n$;
14: $\pi_n = r_0^T q_n$;
15: $\gamma_n = f_0^T s_n$;
16: $\eta_n = f_0^T t_n$;
17: $\theta_n = s_0^T t_n$;
18: $\kappa_n = t_0^T t_n$;
19: $\omega_n = \frac{\eta_n}{\kappa_n}$;
20: $\sigma_n = \gamma_n - \omega_n \eta_n$;
21: $r_n = s_n - \omega_n t_n$;
22: $x_n = x_{n-1} + z_n + \omega_n s_n$;
23: if $x_n$ is accurate enough then
24: STOP
25: end if
26: $u_n = Ar_n$;
27: end for

Figure 2: Parallel BiCGSTAB Algorithm.
class methods that define the range for the computation. Our selection was made based on the need to keep the BiCGSTAB algorithm highly reusable.

We developed an initial parallelized 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).

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 percent 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 parallelized the ILU preconditioner code.

4.4 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 factorization preconditioners are commonly used since they are suitable for any type of matrix. The pseudocode of the ILU algorithm is given in Figure 5.

4.5 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 parallelize on a cluster. In this paper we utilize the block ILU algorithm described in [5].
Figure 4: Profiling of the ILU Preconditioner Code

Algorithm 4.1: ILU($M, x, D$)

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

Figure 5: The ILU Algorithm
The block ILU chosen splits the matrix into blocks, assigning each block to a processor. This technique involves no synchronization 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 4 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.

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 parallelized version runs much faster and obtains better scalability. The speedup is close to 4 on a 8 processor machine (thought degrades when additional processors are used). This is not the result that were hoping for. We hoped to get linear scalability. So we continued to profile the code, looking for additional opportunities for tuning.

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 synchronization point. We obtained a profile of the total communication time and the communication time for each synchronization 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 7: *Speedup of the BiCGSTAB using Block-ILU*

Figure 8: *Communication profile for the code.*
4.6 Threaded Communication

In our work, we have found that there exist scalability issues which are caused by the communication overhead. A straight-forward 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 \texttt{Allreduce()} and \texttt{Allgather()} functions. Rewriting the \texttt{Allreduce()} call using \texttt{Isend} and \texttt{Irecv} is not a good solution since there is a need to synchronize 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 \texttt{Allreduce()} and \texttt{Allgather()}. Once the communication thread is started, it goes to sleep and waits for the main process to issue a wake-up signal. The main process spawns a wakeup communication thread, which runs in parallel to the main process which 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 synchronization with the main process. If the application involves a significant amount of computation, higher performance will result using our thread-based synchronization 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 parallelized code, up to 8 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 synchronization will result.
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 utilizes 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 obtained 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, $T_0$ is the latency, $M$ the message size in bytes and $r_\infty$ is the bandwidth.

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

From the algorithm, we know the size of each message and the frequency of each MPI call, so we can estimate the total time due to communication overhead. In this estimate, we assume that \textit{Allgather()} and \textit{Allreduce()} are executed in 4 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 10 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 parallelization 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. Some next steps are to develop parallel versions of the CG, BiCG, GMRES and QMR algorithms, and then integrate these into the parallelized 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 appears to be a promising direction for our work. We also plan to improve the design, targeting increased flexibility and reusability.

7 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).
References


