Vol. 23(2) : 173~179 | Ocean and Polar Research | June 2001

Application of a Parallel Asynchronous Algorithm to Some Grid Problems on Workstation Clusters

Pil Seong Park*

Department of Computer Science, University of Suwon
Suwon 445-743, Korea

Abstract : Parallel supercomputing is now a must for oceanographic numerical modelers. Most of today's parallel numerical schemes use synchronous algorithms, where some processors that have finished their tasks earlier than others must wait at synchronization points for correct computation. Hence, the load balancing is a crucial factor, however, it is, in general, difficult to achieve on heterogeneous workstation clusters. We devise an asynchronous algorithm that reduces the idle times of faster processors, and discuss application of the algorithm to some grid problems and implementation on a workstation cluster using Message Passing Interface (MPI).

Key words : parallel asynchronous iterations, load balance, message passing, MPI, workstation clusters.

1. Introduction

Grand Challenge (e.g. see http://www.nhse.org/grand_challenge.html) applications are fundamental problems in science and engineering whose solutions can be advanced by applying high performance computing and communications (HPCC) technologies. A common feature of many of these is that they involve simulation. Due to limitations in today's computing power, many simulations cannot be completed with sufficient accuracy and timeliness to be of interest.

Global scale oceanographic numerical modeling projects are included in such cases. For example, NASA's Earth and Space Sciences (ESS) Projects have 9 Grand Challenge teams (http://sdcd.gsfc.nasa.gov/ESS/grand.st2.html), including "Four Dimensional Data Assimilation" and "An Earth System Model: Atmosphere/Ocean Dynamics and Tracers Chemistry". However, such problems are not solvable without the help of HPCC scientists who identify and develop parallel techniques and algorithms that minimize communications. Repeated experiences have proven that taking existing serial computer programs and trying to port them to parallel computer systems is not only an arduous exercise but also results in poor performance. For this reason, the emphasis is on totally rewriting important computational kernels from scratch with the considerations of the parallel computer architectures in mind.

HPCC is not just a better option but a must for today's oceanographic numerical modelers, even if he/she does not deal with global scale phenomena. This is because a much finer model grid is needed than before to identify detailed oceanographic processes, and this in turn requires tremendous amount of computational time and use of parallel supercomputers.

Since the early 1990s, there has been an increasing trend to move away from the expensive and specialized proprietary parallel supercomputers towards low-cost network of workstations (e.g. Baker and Buyya 1999). That is, with the advent of high speed networks and low cost commodity processors, it is now easy to build a workstation cluster at moderate cost, since all the necessary technical documentation can be found on the

* Corresponding author. E-mail : pspark@mail.suwon.ac.kr

Internet (e.g. http://www.beowulf.org/). Due to its good performance compared to the price, it is expected that most of commercial supercomputers will be replaced by self-made workstation clusters in the future.

In the parallel programming paradigm, the main stream is still synchronous iterative algorithms, where some processors that have finished their tasks earlier than others must wait at synchronization points for correct computation. Hence, the load balancing is a crucial factor, and each subproblem must be approximately the same size (e.g. Foster 1995).

Since the pioneering work by Chazan and Miranker (1969), asynchronous iterative methods have been developed by many authors as a way to avoid or reduce processor idle time by eliminating synchronization points as much as possible. In asynchronous algorithms, each processor executes its own instructions, reading data produced by other processors, but not waiting for new data if the latter is not yet available. There is no synchronization barrier to overcome, and thus all processors compute their tasks with no interruption. The methods usually have a slower asymptotic convergence rate than ordinary synchronous ones, but in many instances, they produce an approximate solution in much less computational time.

As one of preparatory steps towards parallel computation of oceanographic numerical models on workstation clusters, we consider the asynchronous iteration technique for better performance. However, most authors used asynchronous iterative methods on shared memory systems, which is, in some sense, easy to implement. We consider implementation of the algorithm on distributed memory machines like workstation clusters, confining ourselves to some grid problems in numerical modeling.

2. A general asynchronous iterative method for linear systems

Asynchronous algorithms are being developed in many areas (e.g. Uresin 1990; Barán *et al.* or Cole and Ofer 1991 for other areas). However, they are not very common yet, since it is not easy to prove that they still give correct results in spite of asynchrony. Lu *et al.* (1993) gave a good but brief survey on general asynchronous parallel schemes. The following is a well established theory for the solution of some matrix problems (Szyld 1998).

We consider a linear system of equations $A\,x = b$, where $A \in R^{n\times n}$ and $b \in R^n$, in a block form as

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1L} \\ A_{21} & A_{22} & \cdots & A_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ A_{L1} & A_{L2} & \cdots & A_{LL} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_L \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_L \end{bmatrix}, \quad (1)$$

where L is the number of subproblems. We assume that the linear system is obtained by discretizing some partial differential equations on two- or three-dimensional grids.

Assuming that each subproblem has been already assigned to some process[1] (more than one subproblem can be assigned to a process) and an appropriate initial guess for each subvector x_k has been made, the general form of a parallel synchronous algorithm on shared memory machines is then,

Algorithm 1 (synchronous, for the process that updates x_l)

Repeat until satisfaction,

1. Read new values of x_k, $k \neq l$ from the shared memory.
2. Solve(or approximate) $A_{ll}\, x_l = b_l - \sum_{k\neq l} A_{lk}\, x_k$ for x_l.
3. Write x_l to the shared memory.
4. Compute the global residual and check for convergence.

Note that there are two implicit synchronization points for correct computation. That is, all the processes should finish updating the subvectors (step 3) before computation of the global residual, and all the processes should take part in step 4. Hence each process does the same number of iterations.

An asynchronous iterative algorithm looks very much the same as the synchronous Algorithm 1. However, each process executes every step independently without synchronization, and computation of the global residual for checking convergence has to be

1) In MPI programming, a working unit is a process, not a processor. In general, if the number of processes created is less than or equal to that of the processors, it is guaranteed that exactly one process runs per processor. Otherwise more than one process can reside on a processor. We describe algorithms based on processes hereafter.

replaced by the so called global *asynchronous residual* (to be defined and discussed in detail later) that is computable in an asynchronous way.

Hence the number of iterations each process computes may be different from process to process, and each process uses the values of the unknowns computed by other processes as is, regardless of whether they have been updated or not. Hence, faster processes need not wait for slower ones to update, and slower ones make use of only the most recent values, ignoring any older information.

Algorithm 2 (asynchronous, for the process that updates x_l)

Repeat until satisfaction without synchronization,

1. Read the values of x_k, $k \neq l$ as is, from the shared memory.
2. Solve(or approximate) $A_{ll}\ x_l = b_l - \sum_{k \neq l} A_{lk}\ x_k$ for x_l.
3. Write x_l to the shared memory.
4. Compute the lth partial *asynchronous residual* and check for convergence.

On a distributed memory systems, "read from the shared memory" (step 1) and "write to the shared memory" (step 3) should be replaced by "receive from other process(es)" and "send to other process(es)".

Asynchronous iteration can be easily implemented on shared memory systems, since communication between processes is done via common shared memory. Hence, if we just allow each process to write to shared memory whenever it updates unknowns at step 3, and read any value it needs at step 1, the resulting computation is exactly asynchronous.

However, it is not the case on distributed memory systems like workstation clusters, since data exchange between processes should be done via explicit message passing. Hence, we need to remove the two synchronization points, so that data exchange can be done asynchronously without any delay.

3. Domain decomposition and the resulting algorithm

For simplicity, we consider a two-dimensional grid problem on a rectangular domain, in which each grid point is connected to 4 adjacent ones. If we use lexicographical ordering, the resulting matrix is 5-diagonal, and only 5 one-dimensional arrays are enough to store nonzero entries of the matrix A.

We use domain decomposition in one direction (Fig. 1a). Let the subvector that consists of the unknowns on the ith column of the grid points be x_i. We use a line Gauss-Seidel method to update the values of the unknowns on each vertical line simultaneously, by solving each diagonal block (which is tridiagonal) A_{ii} for x_i by Gauss elimination (step 2 in Algorithm 2). We assume that each subdomain (that contains several vertical lines of grid points) is assigned to one process, and each process solves the subproblems one by one. For simplicity, we number all the processes from 0 to n-1(n is the total number of processes), and we call that the ith process lies of the (i+1)-th process, etc.

An example of decomposition of a 15 $\times$10 grid into 3 subregions, each having 5 columns of grid points (or 50 grid points), is shown in Fig. 1b. In terms of the parameters used in (1), L is 15 (i.e., the number of vertical grid lines), and we assign 5 subproblems (or 5 vertical lines of grid points) to each process.

Only the process responsible for update of the subvector x_i needs to have the non-zero entries of $A_{i,i-1}$, $A_{i,i}$, and $A_{i,i+1}$ of the partial matrix $[A_{i,1}\ A_{i,2} \cdots A_{i,L}]$ of A in (1), where $A_{i,i}$ is tridiagonal, $A_{i,i-1}$ and $A_{i,i+1}$ are diagonal, and the rest are zero matrices. In addition to the subvectors to update, the process should receive updated subvectors that correspond to the boundaries of the adjacent process(es) (drawn in thick lines in Fig. 1b).

Since each process does not need the subvectors at the internal grid points in adjacent subdomains, we let each process update the unknowns at the boundary points between subdomains first and send them immediately to adjacent processes, so that adjacent processes can make use of the new values as early as possible. In this respect, a line Gauss-Seidel method is preferable.

The global residual $r = b - A\,x$ for checking convergence can be computed by a certain process (say, the

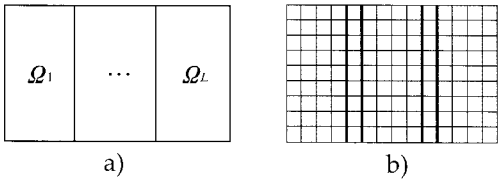

Fig. 1. a) Domain decomposition into L subregions in one dimension.
b) An example of decomposition of a 15 $\times$ 10 grid into 3 subregions.

root process, that can be any of them) after collecting the sizes of partial residuals computed by all the processes independently as follows:

Lemma 1. Let $r_i = b_i - \sum_k A_{ik} x_k$ be the i th partial residual. Then
$\| r \|_2^2 = \sum_i \| r_i \|_2^2$, and each r_i is computable by the process that updates x_i using only the information it has.

Proof: The relation between r and r_i is obvious. Only the nonzero submatrices $A_{i,i-1}$, $A_{i,i}$, $A_{i,i+1}$ and subvectors $\overline{x_{i-1}}$, x_i, and $\overline{x_{i+1}}$ are necessary for the computation of r_i, which the process that updates x_i already has.

Let us assume that the jth process is responsible for update of subvectors $\overline{x_{j1}}, \cdots, \overline{x_{jk}}$. Then the resulting synchronous algorithm (say, Algorithm S) can be written as

Algorithm S (synchronous, for the *j*th process)

1. Take appropriate initial guesses for $\overline{x_{j1-1}}$, $\overline{x_{j1}}$,..., $\overline{x_{jk}}$, $\overline{x_{jk+1}}$.
2. LU-decompose diagonal blocks $A_{j1,j1}$, ... , $A_{jk,jk}$.
3. Repeat until convergence,
   1) Update $\overline{x_{j1}}$ and $\overline{x_{jk}}$ on the boundary.
   2) Send $\overline{x_{j1}}$ to the process on the left, and $\overline{x_{jk}}$ to the process on the right.
   3) Receive $\overline{x_{j1-1}}$ and $\overline{x_{jk+1}}$ from the processes on the left and right.
   4) Update internal grid lines $\overline{x_{j1+1}}$,..., $\overline{x_{jk-1}}$ one by one.
   5) Compute $\sum_{l=1}^{k} \| \bar{r}_{jl} \|_2^2$ and send it to the root process.
   6) The root process collects all the partial residual information(including its own) and computes the size of global residual $\| \bar{r} \|_2^2$. Then send it to other processes.
   7) If $\| \bar{r} \|_2 <$ TOL (tolerance), stop running.

In the next section, we modify Algorithm S so that it can run on workstation clusters asynchronously.

4. Implementation on workstation clusters in MPI environment

In the 1980s, it was believed that computer performance was best improved by creating faster and more efficient processors. However this idea was challenged by parallel processing, which means linking together two or more computers to jointly solve a computational problem. Since the early 1990s, a cluster/network of workstations (COWs/NOWs) is becoming an appealing vehicle for cost-effective parallel processing. Brief introductions to cluster computing, and its parallel programming models and paradigms are given in Baker and Buyya (1999) and Silva and Buyya (2000), respectively.

In 1994, the first private cluster computer consisting of 16 DX-4 processors was constructed for NASA's ESS project. They called the machine "Beowulf". It's a kind of high-performance massively parallel computer built primarily out of commodity hardware components, running a free-software operating system like Linux, interconnected by a private high-speed network.

The project quickly spread to other NASA sites, other R&D labs and to universities around the world. Clusters of PCs or workstations dedicated to running high-performance computing tasks are called Beowulf clusters.

For the implementation of the mentioned asynchronous algorithms, we use the cluster "Atom" (http://210.123.54.230/cluster/atom.htm) at Physics Department, University of Suwon. It is a Beowulf cluster that consists of 7 dual Pentium-2 nodes, equipped with MPI-1, and runs on Linux. We found that some essential features necessary for asynchronous iteration are not supported by MPI as is (e.g. MPI Forum 1995), at least by MPI-1. That is, even though we use unblocked send/receive functions or any others, the messages sent by faster processes could not be processed by slower processes and pile up in the system buffer, ending up with overflow.

Hence we created a new function that can receive messages asynchronously. This can be achieved by making each process probe for any message already arrived and perform appropriate jobs on it from time to time. The new function uses MPI function MPI_IProbe() and the structure is as follows:

```
while (MPI_IProbe(MPI_ANY_SOURCE, MPI_ ANY
      _TAG, comm, & flag, &status)) {
      if (flag ) {
      requester=status.MPI_SOURCE;
      jobkind=status.MPI_TAG;
      (Do an appropriate job according to the value of
      jobkind;)
        }
      }
```

MPI_IProbe() returns flag=1 if there are some mes-

sages of any kind from any source. If so, the next step identifies the sender and the kind of messages from the array "status" of size MPI_STATUS_SIZE. Then, it is supposed to perform appropriate work according to "jobkind".

To remove the synchronization point necessary for computing the global residual, we use an asynchronous residual instead, defined by

Definition 1. Let $\overline{r_i^{(A)}} = b_i - A_{ii}\,\overline{x_i^{(N)}} - \sum_{k \neq i} A_{ik}\,\overline{x_k^{(O)}}$ be the i-th asynchronous residual. Superscripts "(O)" and "(N)" denote "the value as is" and "the new value", respectively.

That is, the process responsible for computation of the partial residual $\overline{r_i^{(A)}}$ ("(A)" denotes asynchronous residual) requires new values of $\overline{x_i^{(N)}}$, but does not care whether the rest x_k's have been updated or not. Note that the asynchronous residual is not the same as the synchronous one, but it makes little difference in practice.

To implement all the idea mentioned so far, we may consider a few different asynchronous models. One of them is a client-server model (Let's call it Algorithm A.) that simulates a shared memory system. The role of a server process is to receive the updated values at boundary points between subregions from each process asynchronously, supply them whenever some client processes need them, and computes and sends the global asynchronous residual as soon as some clients send their partial residuals. The role of client processes is nearly the same as that described in Algorithm S, except that all message passing is done between clients and the server only. Note that the server's workload is quite smaller than clients' and it spends most of its time just by probing any incoming messages or requests. Hence, the server can quickly respond to clients' needs, but it wastes the computational power of the server.

Algorithm A-1 (for client processes)

Repeat until the global asynchronous residual is satisfactory.

1. Update the subvectors at left and/or right boundary points.
2. Send the updated boundary values to the server.
3. Update the values of unknowns at internal grid points.
4. Request and receive boundary values, that are updated by adjacent processes, from the server.
5. Compute its own partial residual and send it to the server.
6. Receive the size of the global asynchronous residual $\| r \|_2$ from the server.
7. If $\| r \|_2 <$ TOL (tolerance), stop running.

Algorithm A-2 (for the server process)

Repeat until all servers stop running,

1. Probe any incoming messages or requests from other processes.

Case 1. If it contains updated boundary values,

1) Store it in its local memory.
2) Send the boundary values the client will need (i.e., the boundary values that are computed by the processes adjacent to the client)

Case 2. If it contains information on partial residual,

1) Store the value.
2) Compute the size of the global asynchronous residual using the newly arrived partial residual.
3) Send the value to the client.

The asynchronous residual is recalculated by the server whenever some clients report their partial residuals. The server then sends it to the client (but not to others) and knows whether the client will continue or stop iteration. The server stops running only if all clients stop iteration.

The second model (say, Algorithm H for simplicity) does not have a separate server process but one of the clients (say, the root process) is supposed to perform the server's job, too. In this case, the algorithm for the root process is a mixture of the above two, and we do not include it here.

In both cases, the algorithms are, in fact, synchronous between clients and a server (or the root process), but asynchronous on the whole.

5. Numerical experiments and discussion

As a sample problem, we choose a two-queue overflow queuing network model discussed in Kaufman (1983), whose graph is a two-dimensional grid of size 1,000 ×1,000 (hence there are 1,000 vertical grid lines), resulting in a matrix of size 1,000,000 ×1,000,000. As mentioned before, the matrix is sparse, and only 5 diag-

onals are nonzero. We use only 5 vectors to store the nonzero entries.

Table 1 shows the elapsed wall clock time to get the residual of size 10^{-3}, 10^{-6}, and 10^{-9}, respectively, by asynchronous Algorithms A and H, compared with that of the synchronous Algorithm S. Job assignment shows how many vertical grid lines have been assigned to each process. For example, job assignment for B07 means that 143 grid lines have been assigned to processes from 0 to 5, and 142 to process 6.

The results which begins with B is the case when workload is well-distributed. Results that begins with U is the case when a large job is given to a certain process. In all results, Algorithm A uses one more process (i.e., the server process) than the other two. Since there are 14 processors (i.e., CPUs) on Atom cluster, Algorithm A for B07, U07 uses 7 client processes and a server process, each running on different CPUs. But Algorithm A for B14 uses 15 processes running on 14 CPUs.

When the workload is well-balanced (e.g. see B07), there is little difference in the performance of the three algorithms. But in the result of B14, Algorithm A is slower than the other two, since two processes are running on a particular CPU. When the workload is unbalanced (e.g., see U07), asynchronous algorithms are faster than synchronous counter part by 7 or 8 %.

The result L07 is the case when the computational workloads are well-balanced, but some additional workload is given to process 3. Additional workload is given by inserting a dummy loop that just consumes CPU time. This simulates the case when some unexpected workload (e.g. workload by other users, etc.) is given to some processors. To get the residual of size 10^{-9}, the wall clock time of Algorithm S increased to 221.83, but that of Algorithm H increased to only 167.47.

Algorithm A may be thought of as the upper limit of the performance of Algorithm H, as long as exactly one process resides on each CPU. In practice, it seems waste of the computing power of the CPU on which the server process resides. But if we add a slower CPU like 486 to the cluster and make the server process run on it, it is meaningful.

6. Applicability of asynchronous algorithms

Of course, numerical modelling problems in oceanography are quite complicated and cannot be written as a single matrix problem. However, the technique can be applied to some inner solver of a numerical model.

The remaining question is, whether the problem satisfies the necessary condition for convergence of the asynchronous algorithm as described below:

Theorem 1. (Szyld 1998) Let G be a covering (or a partition) of J_n. Let A and each of the matrices A_{ll} be monotone, $l \in J_L$. Let $(-A_{lk}) \geq O$, $l \neq k$, $l, k \in J_L$. Then, the asynchronous Algorithm 1 converges to the solution of (1) for any initial vector $\bar{x}^{(0)}$.

We should make sure that the resulting matrix satisfies the above condition. However, the above is a strict

Table 1. Comparison of synchronous and asynchronous algorithms.

Results	Algorithm	# of processes	wall clock time(sec)			Job assignment (grid lines)
			le-3	le-6	le-9	
B07	S	7	51.36	71.08	90.54	143,143, 143, 143, 143, 143, 142
	A		50.99	70.33	89.66	
	H		51.69	71.20	90.70	
U07	S	7	114.78	158.93	202.51	110, 110, 110, 340, 110, 110, 110
	A		104.08	144.75	184.85	
	H		105.62	147.20	188.77	
B14	S	14	31.43	43.38	55.26	72,72,72,72,72,72, 71,71,71,71,71,71,71,
	A		36.47	50.21	63.73	
	H		51.69	71.20	90.70	
L07	S	7	125.48	173.97	221.83	143,143,143,143,143,143,142
	H		101.77	136.81	167.49	

mathematically derived condition, and there may be enough cases for convergence even though the above condition is not satisfied.

In fact, a great number of scientific problems satisfies such conditions or the like, and we could easily use the technique in other problems.

Acknowledgements

The author wishes to thank Professor Seung Jong Lee at Physics Department, University of Suwon, for allowing numerical experiments on the Atom cluster. This work was supported by one of KORDI's research projects, Operational Ocean Prediction System (BSPM 00050-00-1291-1), funded by the Ministry of Maritime Affairs and Fisheries (MOMAF). The auther thanks Prof. J.H. Yun and an anonymous reviewers for their careful review and helpful suggestions.

References

Barán, B., E. Kaszkurewicz, and A. Bhaya. 1996. Parallel asynchronous team algorithms: Convergence and performance analysis. *IEEE Transactions on Parallel & Distributed Systems*, 7, 677-688.

Baker, M. and R. Buyya. 1999. Cluster computing at a glance. In: *High Performance Cluster Computing: Architecture and systems, Vol. 1*, ed. by R. Buyya. Prentice Hall, New York.

Chazan, D. and W. Miranker. 1969. Chaotic relaxation. *Linear Algebra and Its Applications*, 2, 199-222.

Cole, R. and Z. Ofer. 1991. *An Asynchronous Parallel Algorithm for Undirected Graph Connectivity.* TR-546, Dept. Computer Science, New York Univ.

Foster, I.T. 1995. *Designing and Building Parallel Programs.* Addison-Wesley Pub. Co., Reading, MA.

Kaufman, L. 1983. Matrix methods for queuing problems. *SIAM J. on Scientific & Statistical Computing*, 4, 525-552.

Lu, E.J., M.G. Hilgers, and B. McMillin. 1993. *Asynchronous Parallel Schemes: A Survey.* Technical Report CSC 93-19, Dept. of Computer Science, Univ. Missouri-Rolla.

MPI Forum. 1995. *MPI: A Message-Passing Interface Standard.*

Silva, L.M. and R. Buyya. 2000. Parallel programming models and paradigms. In: *High Performance Cluster Computing: Programming and Application Issues, Vol. 2*, ed. by R. Buyya. Prentice Hall, New York.

Szyld, D. B. 1998. Different models of parallel asynchronous iterations with overlapping blocks. *Computational and Applied Mathematics*, 17, 101-115.

Uresin, A. and M. Dubois. 1990. Parallel asynchronous algorithms for discrete data. *J. ACM*, 37, 588-606.

Received May 14, 2001
Accepted Jun. 30, 2001