EFFICIENT ALGORITHMS FOR GHOST CELL UPDATES ON TWO CLASSES OF MPP ARCHITECTURES

Bruce Palmer and Jarek Nieplocha
Pacific Northwest National Laboratory
Richland, WA 99352
USA

Abstract: Several methods for updating the ghost cell regions of large, distributed arrays are described and results on their relative efficiencies are presented. It was found that no single algorithm provided optimal performance on all platforms and that the most efficient update depended on the details of the system architecture.

Keywords
ghost cells, distributed arrays, update algorithms

1. Introduction

Calculations on logically regular grids are common in many areas of numerical simulation and engineering. Increases in computer speed have led to corresponding increases in the accuracy and range of application of these calculations; however, the trend toward parallel computers has made realizing these gains in speed and problem size increasingly difficult. For large problems, efficient use of the computer requires communication between different processors, which complicates the programming model considerably. To simplify the creation of parallel programs, higher level languages and toolkits have been developed to provide tools that bypass the need for explicitly programming message-passing from one processor to another. Examples are the POOMA environment [1], the Overture toolkit [2], and KeLP [3]. Typically, the higher the level of abstraction, the easier it is to develop code, but performance may not be optimal for all, or even many, situations. Lower levels of abstraction provide the programmer with more flexibility and hence better performance, but at the cost of increased consideration of how data is distributed across processors and what patterns of data access will lead to optimal performance.

This paper describes a new methodology that integrates ghost cell capabilities with a library-based shared-memory programming model implemented in the context of distributed dense arrays. This hybrid approach combines the benefits of a shared memory programming model, high-level abstractions, ease-of-use, implicit communication, and a global view of arrays, with the flexibility and performance of distributed array libraries that offer ghost cell capabilities. This work has been done using the Global Array toolkit [4] designed to support the use of ghost cells for regular grids. In addition, we will discuss the performance issues associated with updating the ghosts cells with data from other processors. The Global Array (GA) toolkit allows the programmer to create regular arrays that are distributed across processors and supplies a complete suite of operations to access and manipulate the data without the need for programming explicit communication calls. These interfaces can be called from programs written in Fortran 77, Fortran 90, C, and C++. The ghost cell implementation described here is aimed specifically at supporting calculations on distributed logically regular grids. Because of this focus, the Global Arrays have several advantages over other existing systems for problems defined on regular grids. The most important is that the update operation to fill in the ghosts cells can be treated as a collective operation, enabling a multitude of optimization techniques. The toolkit also allows ghost cell widths to be set to arbitrary values in each dimension, thereby allowing programmers to improve performance by combining multiple fields into one global array and using multiple timesteps between ghost cell updates. Other packages have also implemented ghost cell capabilities. KeLP builds up the total array from regions associated with each processor and communication is handled by creating a schedule based on region overlaps. Although this allows for more flexible simulation domains, the communication between processors is not optimal for large, regular arrays. Overture is also based on building up logically rectangular regions into complex geometries. Each region is associated with a single processor and the toolkit provides routines for automatically interpolating and communicating between grids in the regions where they are matched or overlap. Again, for large regular arrays, communication may not be optimal. POOMA is a fairly high level abstraction and is designed primarily to support object-oriented programming. None of these packages currently have the availability, portability, language independence, and support of Global Arrays.

In the following sections, we discuss the technical approach, including algorithmic and architectural issues influencing the selection of communication protocols, and present performance results for two classes of MPP systems – one optimized for one-sided communication and the second for message-passing. The first is the Cray T3E representing the global address space architecture with h/w optimization for low-latency and high bandwidth one-sided communication. The other is the IBM SP representing a cluster architecture based on the commodity SMP nodes and the interconnect optimized for message-passing (IBM in recent years has added support for one-sided communication through the LAPI library; however, that capability is implemented on top of the low level message-passing and Pthreads). Although results are not reported

here, the ghost cell algorithms have also been tested on a shared memory platform (SGI) and a Linux cluster. Performance results for an actual application that was developed based on the described ghost cell interfaces are also given.

2. Technical Approach

Currently, the decomposition of data representing a large array among different processors in the Global Array toolkit is non-overlapping. To allow easy access to boundary data on neighboring processors, the Global Arrays have been extended to include ghost cell regions on each processor. The extended Global Arrays now have allocated memory for a boundary region that can be filled in with data from the adjoining processors. This is illustrated schematically in Figure 1. The data in the Global Array can be accessed either locally or through the global index space. The Global Array toolkit also has been augmented by an update operation that can be used to automatically fill in the ghost cells with data. Several different implementations of the update operation were investigated during this work, and performances were compared on a number of different platforms.

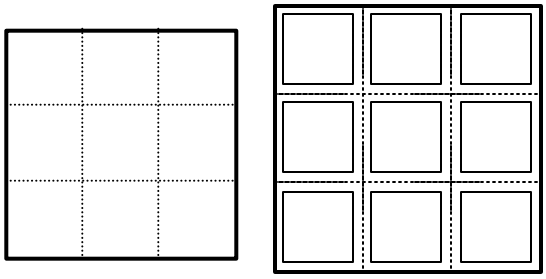

Figure 1. Schematic illustration of extension of ordinary global array (left) to global array with ghost cells (right). Heavy solid lines are global array boundaries, light solid lines are boundaries of visible data on each processor, dotted lines are boundaries of ghost cell data.

Ghost Cell Implementation and Operations

The bulk of the effort in implementing Global Arrays with ghost cells involved the ghost cell update operation. As described above, the update operation fills in the ghost cells with the visible data residing on neighboring processors. Once the update operation is complete, the local data on each processor contains the locally held "visible" data plus data from the neighboring elements of the Global Array, which has been used to fill in the ghost cells. Thus, the local data on each processor looks like a chunk of the Global Array that is slightly bigger than the chunk of locally held visible data.

There are multiple ways of implementing the update operation. The most straightforward method is for each processor to decide where its visible data needs to go and to place that data on each of the neighboring processors using a set of "PUT" operations. For a D-dimensional system, this requires 3^D-1 messages. The implementation of the simple PUT based algorithm is illustrated schematically in Figure 2.

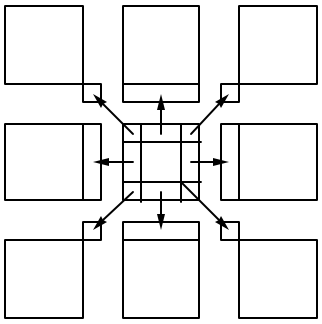

Figure 2. Schematic of simple PUT implementation of update algorithm in two dimensions. Visible data from central processor is transferred in separate messages to neighboring processors.

Another way of implementing the update operation is the shift algorithm. A variant of this algorithm used in parallel implementations of Lagrangian simulations was first described by Plimpton [5]. Recently, Ding and He [6] have applied a similar algorithm to regular lattices, although they did not investigate the efficiency of the update operation. The shift algorithm makes use of the fact that after part of the update has been performed, some information from neighboring processors is held locally and can be included in messages to other processors. This cuts down on the total number of messages that need to be sent from 3^D-1 for the simple PUT algorithm to $2D$ messages for the shift algorithm, so latency costs for the shift algorithm should be much lower (the total message volume is the same for both algorithms). However, it is much more difficult to implement the shift algorithm so that it gives good performance over a broad range of platforms.

Our shift algorithm is illustrated schematically for a two-dimensional array in Figure 3. The data is updated sequentially for each of the D coordinate directions in a D-dimensional Global Array. As illustrated in Figure 3, the first coordinate update puts data into the ghost cell patches immediately adjacent to the east and west sides of the locally held visible data block. The second coordinate update then takes data from both the locally held visible block and the ghost cells to fill in the ghost cell patches to the north and south of the visible data block. This has the effect of filling in the corner ghost cell blocks with data without sending any additional messages. It is important to note, however, that for this algorithm to work correctly, each processor must verify that it has received all ghost cell data from the previous coordinate updates before starting the updates along a new coordinate direction

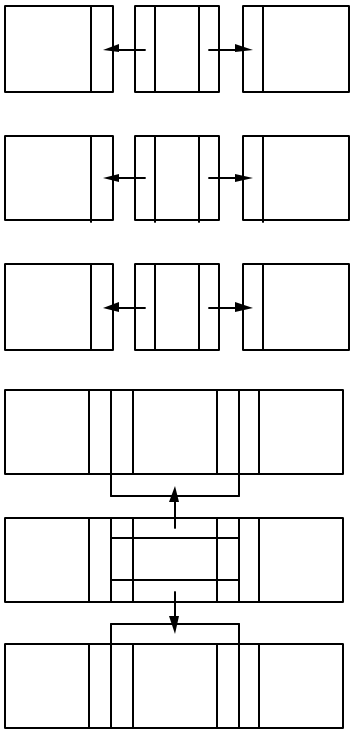

Figure 3. Schematic representation of the shift algorithm. (Top) First update of ghost cell data in horizontal direction uses only visible data on local processor to fill in ghost cell data to the left and right. (Bottom) Second update of ghost cell data uses both visible and ghost cell data (obtained in previous update) to update ghost cell data in processor above and below local processor.

To find the most effective way of performing the update operation, several different implementations were investigated. The first was the simple PUT algorithm, designated by S_P. Each processor needs to identify the blocks of visible data that must be sent to other processors. This data is then placed in the ghost boundary regions of neighboring processors using a sequence of PUT calls. Because this algorithm is sending only messages containing locally held visible data, no synchronization is needed between PUT calls and successive calls can be issued as soon as the previous PUT call has been initiated. Although the simple PUT algorithm requires more messages than the shift algorithm, the lack of synchronization during the update makes this algorithm quite effective over a range of platforms.

All the remaining implementations of the update operation are variations of the shift algorithm. As discussed above, the shift algorithm requires some kind of synchronization during the updates in each of the coordinate directions to guarantee that 1) data needed from other processors has arrived and 2) no data is being overwritten before it is used. However, making an explicit synchronization call between each coordinate update can result in significant communication overhead. To avoid this, the first implementation of the shift algorithm, M_P, attempted to eliminate global synchronization during the update by using message-passing. Message-passing enforces synchronization between sender and receiver and guarantees that no update along a coordinate direction occurs too early.

The second implementation of the shift algorithm, P_Flag, represented an effort to get around problems found for the message-passing algorithm on some platforms. Because the architecture of the Cray T3E was optimized for one-sided communication rather than message-passing, the M_P algorithm gave substantially poorer performance than S_P on the Cray T3E. A new algorithm using a modified PUT operation to perform each update, designated P_Flag, was designed to get around this problem. The standard PUT operation has no way of signaling the remote processor that the data has arrived. The modified PUT attempts to fix this by augmenting the message to include one additional pointer to a location on the remote processor and an extra integer. The extra integer is a flag that is used to tell the remote processor about data arrival. After completing PUTs in both the positive and negative directions, each processor then checks the flags corresponding to these two PUTs to see if the data coming from its neighboring processors has arrived. If not, the processor waits until the data shows up before starting the updates in the next coordinate direction.

The M_P and P_Flag algorithms have not given any consideration to actual configuration of the processors. However, most new parallel machines contain several processors per node. Within a node, processors can communicate optimally using shared memory (direct access to data); between nodes processors communicate using network protocols. To improve performance on these architectures, a hybrid shift algorithm, M_P/P_Flag, was created that attempts to optimize communication based on whether the target processor is on the same node or not. If the processor is on the same node, data is sent using the modified PUT operation with flag described above. If it is on a different node, message-passing is used.

As described above, none of these implementations has synchronization at the beginning or end of the update algorithm. Algorithm M_P does not need any additional synchronization because the send-receive paradigm guarantees that both the source and recipient of the data have completed other operations before the update can proceed. Using P_Flag and M_P/P_Flag without synchronization at the beginning of the update is a little riskier and it is possible that ghost cell data may be overwritten on a remote processor before that processor has had a chance to use it. Algorithm S_P is susceptible to writing over data before it is used and there is no guarantee for this algorithm that a processor has all of its ghost cell data updated before proceeding with the next set of operations. Explicit synchronization calls may not be necessary if the application itself enforces synchronization between updates, but S_P will, in general, require some kind of synchronization to guarantee that the ghost cell data

is available when needed. It is therefore important when assessing the relative merits of the different update implementations to consider the effect of synchronizations prior to and after the update because they can add significantly to the time required to fill in the ghost cell data.

3. Performance Evaluation

Two different evaluations of the ghost update operation were performed. The first was a timing study of the update operation on large integer arrays across two different platforms, the second was a parallel implementation of a two-dimensional lattice Boltzmann simulation. The two platforms used in the first study were an IBM SP (configured with 16 processors per node) and the Cray T3E. The benchmark code was designed so that the amount of data on each processor remained constant as the number of processors was increased. Calculations were performed in both two and three dimensions on arrays filled with arbitrary integers.

Timings for the IBM-SP in two dimensions are shown in Figure 4. For two dimensions, the simple PUT algorithm is about 4 times faster than any of the shift algorithms (the shift algorithms all behaved roughly the same). Note that for this particular test, good scaling behavior is exhibited by a constant time for the update operation as the number of processors is increased, and all the algorithms behave well in this regard. For three dimensions, the message-passing (M_P) and hybrid algorithm (M_P/P_Flag) were roughly comparable and faster than the other two algorithms by a factor of two.

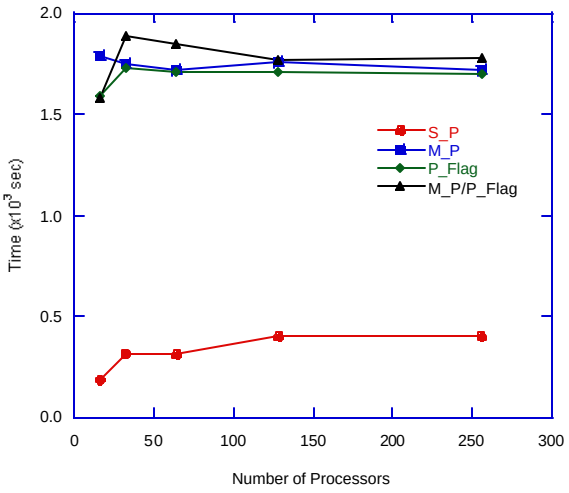

Figure 4. Update timings for two-dimensional test case on the IBM SP

The picture changes significantly if the probable cost of synchronization is added to these updates. The timings for the two dimensional system, including a synchronization call between updates, are shown in Figure 5. It is immediately apparent that all algorithms except M_P have significant synchronization costs for larger numbers of processors (even more so when one considers that M_P can probably be safely used in most cases without any additional synchronization whatsoever). These results suggest that the M_P implementation is more effective on the IBM SP for most cases (the only exceptions seems to be the single node case with 16 processors). The results in three dimensions (not shown) indicate that M_P is more efficient for all numbers of processors.

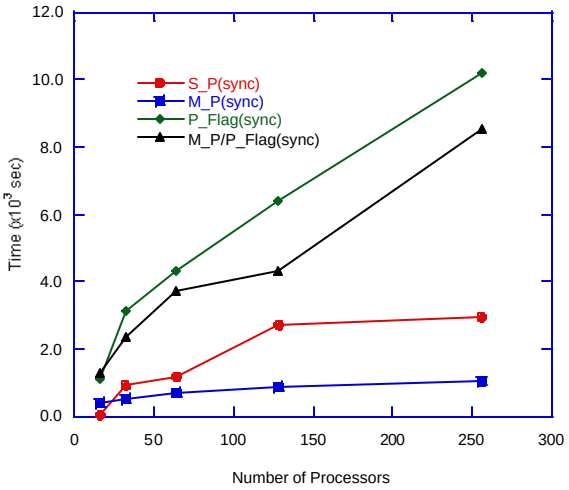

Figure 5. Timings for the update operation with embedded synchronization for the two-dimensional test case on the IBM SP.

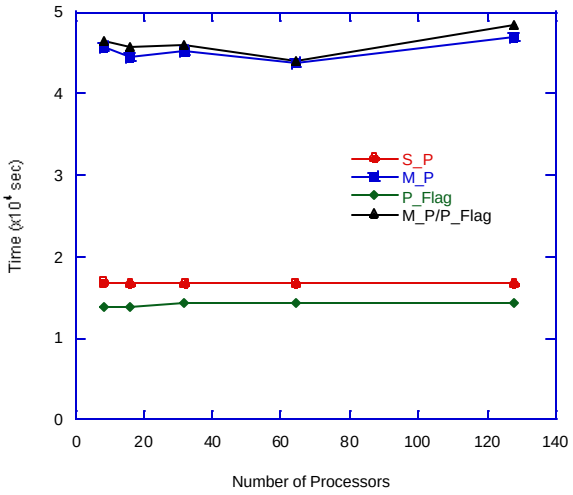

Figure 6. Comparison of update timings for two-dimensional test case on a Cray T3E.

The Cray T3E turns out to be much simpler to evaluate. Due to the h/w barrier support, the synchronizations are very fast in all cases compared to the time required for the update. For this platform, the cost of synchronization can be disregarded and only the times required for the update operations themselves need to be considered. The timings for the ghost cell update operations in two dimensions are shown in Figure 6. The update algorithms are all very well

behaved on the Cray and that there is only a minor amount of variation in the times with increasing number of processors. Furthermore, the P_Flag algorithm is the fastest algorithm in two dimensions and for all numbers of processors. Similar results are obtained in three dimensions. For the Cray T3E, the modified PUT algorithm P_Flag appears to be the optimal solution for all problems.

4. Application Experience

Finally, the Global Arrays with ghost cells were used to implement a two-dimensional lattice Boltzmann simulation of flow in a lid-driven square cavity [7,8]. The lattice Boltzmann method is an explicit technique for simulating fluid flow. It is therefore local and each node only needs information from nearest and next-nearest neighbor nodes in order to perform an update. Both the nearest and next-nearest neighbors can be accessed using a ghost cell region that is one node wide. This algorithm fits neatly into the Global Array with ghost cells model. Each node update is followed by a ghost cell update that communicates new values of the node across processor boundaries. The global address space of the Global Arrays makes it easy to set up the problem and apply boundary conditions by allowing straightforward conversion between local and global indices.

The lid-driven cavity is a square region completely filled with fluid that has "stick" boundary conditions on three sides of the cavity and a constant velocity parallel to the surface on the fourth side. On the IBM SP, the message-passing algorithm M_P was used for the ghost cell updates. Timings are shown in Figure 6 for a simulation on a 1024 x 1024 lattice over 5000 steps at a Reynolds number of about 850. The plot of total computation time shows reasonably linear behavior over the entire range of processors, although it begins to flatten out noticeably at the end. The update time shows a very large drop in going from 8 to 16 processors and then flattens out substantially, but it remains a relatively small fraction of the update time for all number of processors. Note that for this test the problem size is fixed and the amount of data per processor decreases as the number of processors is increased, so the decrease in update times is expected. A velocity magnitude plot of the system after a longer simulation of 100,000 steps is also shown in Figure 7.

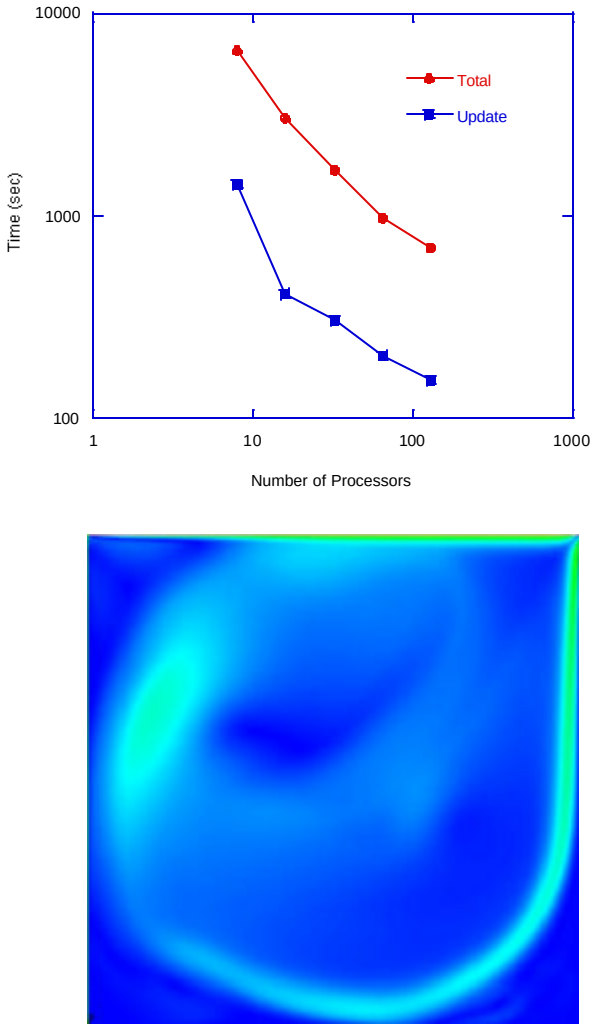

Figure 7. (Top) Timings for a 5000 step lattice Boltzmann simulation on 1024 x 1024 grid. Both the total time for the lattice Boltzmann calculation and the time required to update the ghost cells are shown. (Bottom) Velocity magnitude after 100,000 steps. The top surface is moving at constant velocity to the right.

5. Conclusions

Several algorithms for updating ghost cell regions were presented and analyzed. It was found that the optimal algorithm depends on the characteristics of the particular computer system being employed and, to a lesser extent, the size and dimensionality of the problem. Several of the update algorithms require synchronization either before or after the update and these synchronizations can significantly affect the overall performance of the update operation. The current implementation of the ghost cell update operation in the Global Array toolkit allows the user to control the synchronization semantics embedded in the update operation. Incorporation of the ghost cell routines

into a lattice Boltzmann code indicates that the ghost cells perform quite well in an actual application.

6. Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy (DOE) at Pacific Northwest National Laboratory (PNNL) operated for DOE by Battelle Memorial Institute. This work was supported by the Center for Programming Models for Scalable Parallel Computing, sponsored by the Mathematical, Information, and Computational Science Division of DOE's Office of Computational and Technology Research. The Molecular Science Computing Facility at PNNL, and National Energy Research Supercomputing Center provided computational resources for this work.

References

[1] J.A. Crotinger, J. Cummings, S. Haney, W. Humphrey, S. Karmesin, J. Reynders, S. Smith, and T.J. Williams. "Generic Programming in POOMA and PETE." Generic Programming Lect. Notes in Comput. Sci., **1766**, 218, 2000.

[2] D.L. Brown, W.D. Henshaw, and D.J. Quinlan. "Overture: An Object-Oriented Framework for Solving Partial Differential Equations on Overlappling Grids." SIAM Conference on Object-Oriented Methods for Scientific Computing, 1999.

[3] S. Baden, P. Collela, D. Shalit, and B. Van Straalen. "Abstract Kelp", Proc. 2001 International Conference on Computational Science, 2001.

[4] J. Nieplocha, R.J. Harrison, R. Littlefield. "Global Arrays: Shared Memory Programming Model for Distributed Memory Systems ", Supercomputing'94, 1994.

[5] S. Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. J. Comput. Phys.,**117**,1, 1995.

[6] C. Ding and Y. He. "A Ghost Cell Expansion Method for Reducing Communications in Solving PDE Problems", Supercomputing 2001, 2001.

[7] S. Hou, Q. Zou, S. Chen, G. Doolen, and A.C. Cogley. "Simulation of Cavity Flow by the Lattice Boltzmann Method." J. Comput. Phys., **118**, 329, 1995.

[8] B.J. Palmer and D.R. Rector. "Lattice Boltzmann Algorithm for Simulating Thermal Flow in Compressible Fluids." J. Comput. Phys., **161**, 1, 2000.