Ghost Cell Pattern

Fredrik Berg Kjolstad
University of Illinois
Urbana-Champaign, USA
kjolsta1@illinois.edu

Marc Snir
University of Illinois
Urbana-Champaign, USA
snir@illinois.edu

January 26, 2010

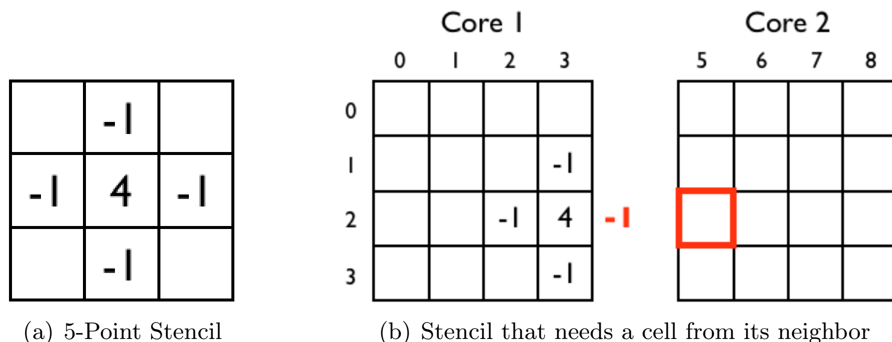
Problem

Many problems consist of a set of points in a grid that are updated in successive iterations based on the values of their neighbors in the same grid. These problems can be divided geometrically into chunks that are computed on different processors. However, since computing the value of each point requires the values of other points these computations are not embarrassingly parallel. Specifically, the points at the borders of a chunk require the values of points from the neighboring chunks. How can we communicate these values between processes in an efficient and structured manner?

Context

Many problems can be described as a structured grid of points in N dimensions where the location of each point in the grid defines its location in the problem domain. The values of the points are updated iteratively and for each update the values of a fixed set of neighboring points are required (see *Structured Grids* and *Iterative Refinement* [1]).

The set of neighboring points that influence the calculations of a point is often called a *stencil*. The stencil defines how the value of a point should be computed from its own and its neighbors' values. It can take many forms and can include points that are not directly adjacent to the current point. Figure 1(a) shows a five-point Laplace operator, which is a stencil that can be used to find edges in an image. It specifies that the value of a point in the current iteration shall be the value of its left, right, up and down neighbors from the previous iteration subtracted from its own value multiplied by four.



(a) 5-Point Stencil

(b) Stencil that needs a cell from its neighbor

Figure 1: Stencil computation in geometrically decomposed grids

Geometric Decomposition [1] is a common pattern for calculating the values of such grids in parallel using different processes or threads. In the rest of this pattern we will use the term process to encompass both processes and threads unless we are talking about issues directly related to shared memory in which case we will use the term thread. The basic idea is to divide the grid into chunks and have each process update one or more of these. As shown in figure 1(b) a common problem with this approach is how to calculate the values at the borders between chunks since these require values from one or more neighboring chunks. Retrieving the required points from the process processing the neighbor chunk as they are needed is not a good solution as it would introduce a lot of small communication operations in the middle of computation which would lead to high latency costs.

Solution

Allocate additional space for a series of *ghost cells* around the edges of each chunk. For every iteration have each pair of neighbors exchange their borders and place the received borders in the ghost cell regions as shown in figure 2. The ghost cells form a *halo* around each chunk containing replicates of the borders of all immediate neighbors. These ghost images are not updated locally, but provide stencil values when updating the borders of this chunk.

If you have a stencil that extends beyond immediate neighbors or if you want to trade computation for communication then use a *Wide Halo* (page 8). If your computation requires cells from diagonal neighbors then you must also exchange *Corner Cells* (page 10). If you need additional performance then *Avoid Synchronization* that is not required by the problem (page 12).

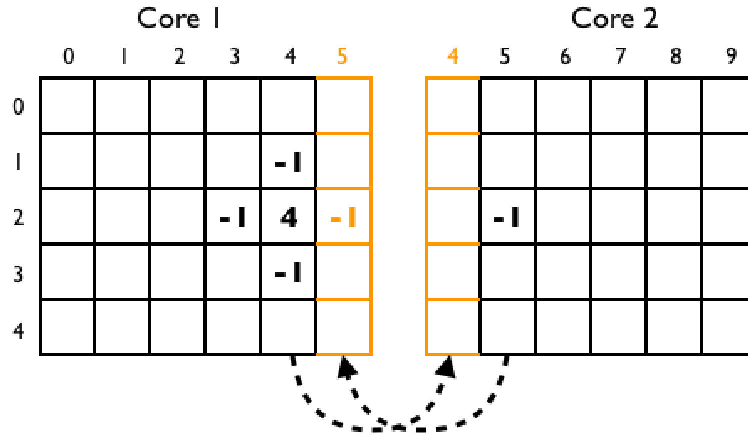


Figure 2: Each chunk receives a vector of ghost cells from neighboring chunks

Forces

Performance vs Complexity A tension exist between the need for performance and the complexity of the implementation. You could simply have stencils fetch cells as they are needed, but that would introduce a lot of small messages that hurts performance. Using ghost cells and performing border exchanges alleviates this problem. If border exchanges are used then you have to consider the optimizations to perform. Examples of optimizations are trading computation for communication (see the section *Wide Halo*), avoiding synchronization (see the section *Avoid Synchronization*) and overlapping communication and computation. These optimizations increase overall performance at the cost of more complexity.

Shared Memory Cost Of Copying vs Contention and Locality On shared memory machines it is possible to avoid the copying associated with ghost cells by having all threads read directly from the set of points from the previous iteration of neighbors. However, this increases the likelihood of cache contention and false sharing as the processor cores read and write to the same cache lines. Alternatively the chunks could be kept separate to avoid false sharing and the threads could have special code to access the value of neighboring chunks, but this reduces locality which means less utilization of separate caches.

Cost of Computation vs Overhead of Communication As mentioned above, it is possible to trade extra computation for fewer border exchanges by maintaining a wider ghost cell halo and redundantly computing results locally. This leads to some redundant work being done, but may reduce time to completion if the overhead of send operations is high. However, the cost of performing the extra computations must be carefully evaluated against the overhead cost of sending messages.

Performance vs Portability Optimizations such as reducing communication at the expense of increased computation and avoiding synchronization require you to make careful trade-offs and these trade-offs are usually based on experimental results. However, these results vary from machine to machine and often also from library to library meaning a fast application on one system may be a slow application on another thus reducing portability.

Size of Chunks As mentioned earlier, the Ghost Cell Pattern is often used with the *Geometric Decomposition* pattern. One of the considerations when using the latter is to select a chunk size small enough to expose sufficient parallelism. However, this affects the border exchanges as it causes the area/volume ratio to increase. The area/volume ratio is the ratio between the surface area of a chunk and its volume. As the size of a three-dimensional chunk increases its volume grows faster than its surface area, $O(n^3)$ vs $O(n^2)$, which decreases the ratio.[1] A high ratio means that more of the time is spent on communication per iteration so that there is less time left to spend on actual computations.

Implementation

Given an image we want to generate a new image containing the edges of the first one. This is called *edge detection* and one way to do it is to repeatedly apply the laplace operator from figure 1(a) to every pixel of the input image. The laplace operator intuitively measures how much each point differ from its neighbors and figure 3 shows it applied to Lena.

[1] For a two-dimensional problem it would be the ratio between the border size and the surface area, but the term area/volume ratio is used in general to describe the effect.

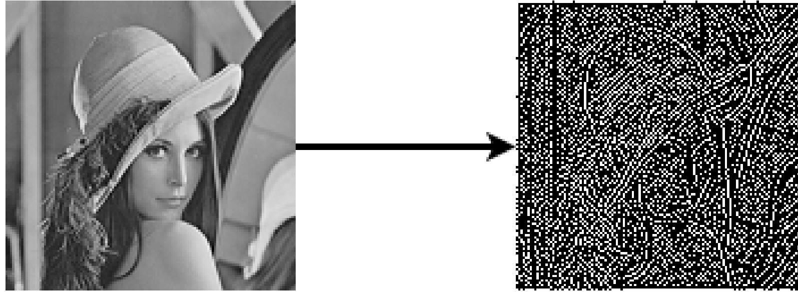


Figure 3: Edge Detection

The sequential algorithm for doing this on a gray-scale image is as follows:

```
1  void laplacian() {
2    for (int iter=0; iter<ITERATIONS; ++iter) {
3
4      // Loop to compute the laplacian
5      for (int y=1; y < (height-1); ++y) {
6        for (int x=1; x < (width-1); ++x) {
7          double pixel = 4 * GET_PIXEL(image, x  , y  )
8                           - GET_PIXEL(image, x-1, y  )
9                           - GET_PIXEL(image, x+1, y  )
10                          - GET_PIXEL(image, x  , y-1)
11                          - GET_PIXEL(image, x  , y+1);
12         GET_PIXEL(buffer, x, y) = BOUND(pixel, 0.0, 1.0);
13       }
14     }
15
16     // Swap buffers
17     SWAP(image, buffer);
18   }
19 }
```

The computation proceeds in iterations and for each iteration the laplacian operator is applied to every pixel of the input image (line 5-14). Since the computations need the surrounding pixels from the *previous* iteration we can't update the image in place and have to use *double buffering*. This means that we have two sets of values; one for the current iteration and one for the previous. On line 17 we swap the buffers. Note that the computation has been simplified for clarity by not computing the laplacian of the borders.

To parallelize this code using the *Single Program Multiple Data* (SPMD) paradigm we need to distribute roughly equal parts of the image to each process before calling `laplacian()` and then merge these image parts again after it has completed. One invocation of the `laplacian()` would thus compute the laplacian for only a fraction of the image.

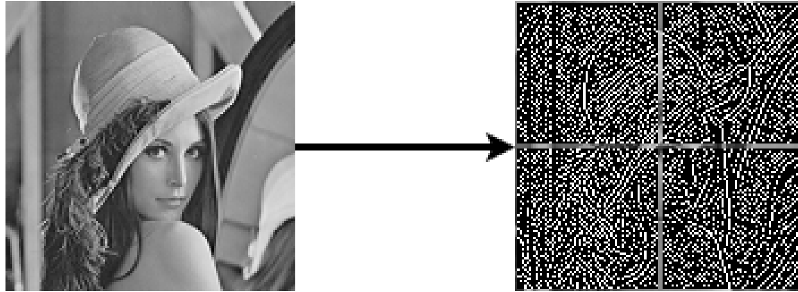


Figure 4: Multiprocess edge detection *without* border exchanges

However, this leads to the problem from figure 1(b) of needing pixels from the neighbors when computing the border pixels. If we ignore the neighbor pixels we get the result illustrated in figure 4. This figure shows laplacian edge detection applied to the image using four processors in a Cartesian grid, but without any border exchanges. Notice the noise at the inner borders between the image chunks.

In order to get rid of the noise we must perform border exchanges for each iteration of the outer loop. The resulting code can look like the following:

```
1  void laplacian() {
2    for (int iter=0; iter<ITERATIONS; ++iter) {
3      // Exchange borders with all four neighbors
4      exchange_west_border();
5      exchange_east_border();
6      exchange_north_border();
7      exchange_south_border();
8
9      // Loop to compute the laplacian
10     for (int y=1; y < (chunk_height+1); ++y) {
11       for (int x=1; x < (chunk_width+1); ++x) {
12         double pixel = 4 * GET_PIXEL(chunk, x  , y  )
13                          - GET_PIXEL(chunk, x-1, y  )
14                          - GET_PIXEL(chunk, x+1, y  )
15                          - GET_PIXEL(chunk, x  , y-1)
16                          - GET_PIXEL(chunk, x  , y+1);
17         GET_PIXEL(buffer, x, y) = BOUND(pixel, 0.0, 1.0);
18       }
19     }
20
21     // Swap buffers
22     SWAP(chunk, buffer);
23   }
24 }
```

Since we are operating under the SPMD paradigm this code is executed on every processor and operates on one chunk of the image instead of all of it. Our chunks have ghost cell halos extending one row/column in each direction so we iterate from the second cell (index 1) to the last cell before the right halo which is located at $width + 1$. Apart from that the biggest difference to the previous code listing is on line 4-7 inside the outer loop where the process perform border exchanges with each of its neighbors. As such it switches between performing computations in the nested loops and communication in the border exchange section. Note that there is no need for a barrier here as all the necessary synchronization is implicit in the exchanges. When a process has received its data it is always safe for it to proceed. The content of the border exchange functions depends on the programming model we are using. If we for instance use MPI then the following code is one reasonable implementation of the exchange_west_border() function:

```
1 // Exchange western colums with western neighbor's eastern columns
2 void exchange_west_border() {
3   if (west_neighbor != -1)
4     MPI_Sendrecv(&GET_PIXEL(chunk, 1, 1),
5                  1, vertical_border_t, west_neighbor, TAG,
6                  &GET_PIXEL(chunk, 0, 1),
7                  1, vertical_border_t, west_neighbor, TAG,
8                  cartesian, &status);
9 }
```

On line 2 we specify that we will only perform a border exchange with the western neighbor if there is one. The MPI_Sendrecv function performs a send and a receive in a deadlock-free manner and is therefore perhaps the most obvious candidate to use for the exchange. However, as we shall see later in the *Avoid Synchronization* section it is not the most efficient one. The parameters on line 4-5 specify the data being sent. Since we are exchanging borders with the western neighbor in this function we send the first column of the chunk that is not a part of the ghost cell region. This column is located at coordinate (1,1). vertical_border_t is a custom type describing one column of data. TAG is a just a name identifying the transmission. Line 6-7 specifies the same for the column we are receiving from the western neighbor and placing in the ghost region at coordinate (0,1). The other border exchanges are defined similarly. Running the new algorithm with border exchanges gives us noise-free edge detection as we saw in figure 3.

Wide Halo

The previous section discussed the use of a halo of ghost cells and disciplined border exchanges to get correct results at the inner borders between chunks being computed in different processes. That solution used a border with a thickness of one since that is sufficient to correctly implement the five-point laplace operator. However, there are a two situations that either require or benefit from a wider halo.

The first situation is the case where the stencil we want to apply to the grid reaches further than the immediate neighboring cells. In that case we *must* use a wide halo for correctness and it can be implemented by extending the previous code to reserve space for more columns and to send more than one border row/column in each border exchange.

The second, perhaps more interesting, case is the use of a deep halo to reduce the number of send operations. There are two components to the cost of sending a message between processes. The first is the time it takes from the call to send a message is issued to the receiving processor starts receiving data called the *latency* of the send. This part depends on the local overhead involved in preparing the message as well as the network latency and is the overhead of sending the message. The second part is the time it actually takes to transmit the data which is dependent on the network bandwidth and the size of the message. It is not uncommon for the message overhead/latency to far exceed the actual transmission time for short messages. Since the network latency is constant for all messages an effective strategy to maximize overall performance is to reduce the number of messages by merging them.

One way this can be done for border exchanges is by increasing the depth of the halo by some factor n beyond the cells actually needed for correctness. By doing this you can limit the border exchanges to every *nth* iteration and although you have to transmit a larger halo you can do so less often and therefore save on messaging overhead. Figure 5 demonstrates how this would work for a halo width of three.

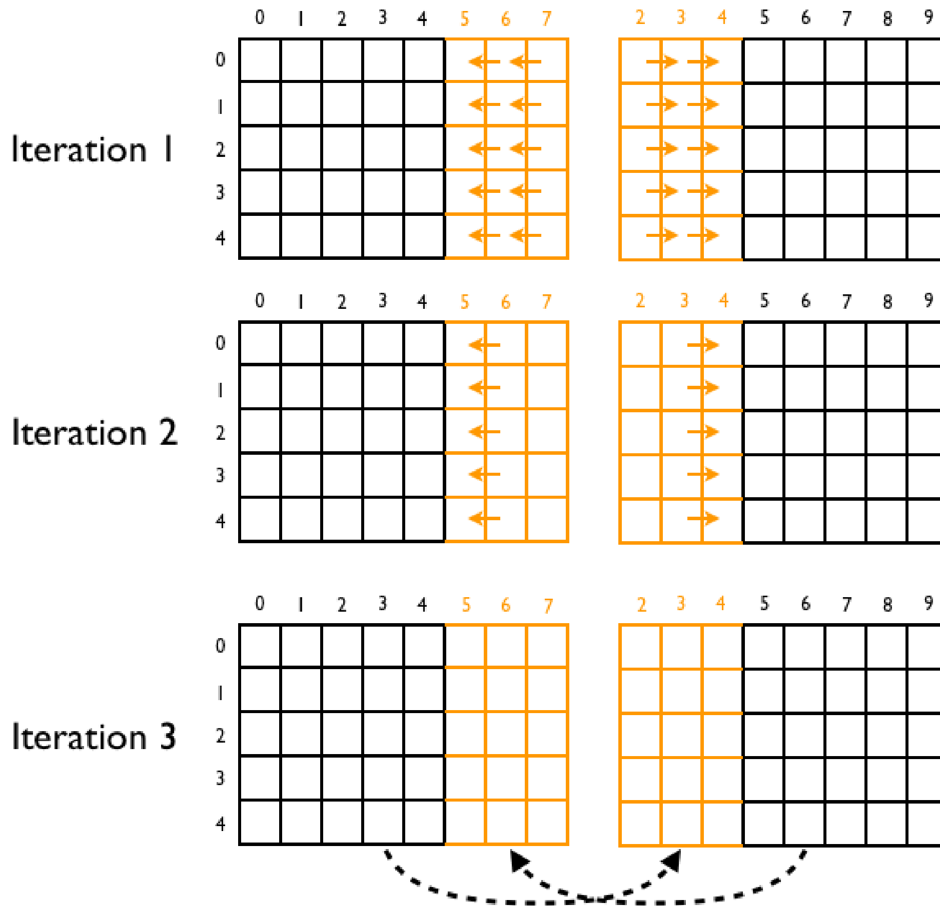


Figure 5: Wide Halo with a Border Exchange every *nth* iteration

The cost of increasing the halo's width is that we have to keep it updated locally, which adds to the computational work. Still, *communication is expensive* and it is often beneficial to trade less frequent communication for extra redundant work. Of course this trade-off must be made so that the cost of the extra work does not exceed the cost of the overhead we removed. Furthermore, the trade-off depends heavily on the target machine's communication capabilities which can vary greatly between different machines. Deep halos therefore reduce the portability of the application.

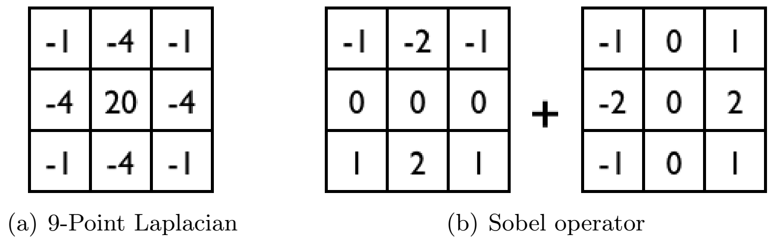
Building on the edge detection example the we will now expand it to use deep borders to trade extra computation for less frequent communication. The following code adds this capability to the `laplacian ()` function:

```
1  void laplacian() {
2    for (int iter=0; iter<ITERATIONS; ++iter) {
3      // Exchange borders with all four neighbors
4      if (border == 0 || iter % border == 0) {
5        exchange_west_border();
6        exchange_east_border();
7        exchange_north_border();
8        exchange_south_border();
9      }
10
11     // Loop to compute the laplacian
12     for (int y=1; y < (chunk_height + 2*border) - 1; ++y) {
13       for ( int x=1; x < (chunk_width + 2*border) - 1; ++x ) {
14         double pixel = 4 * GET_PIXEL(chunk, x  , y  )
15                          - GET_PIXEL(chunk, x-1, y  )
16                          - GET_PIXEL(chunk, x+1, y  )
17                          - GET_PIXEL(chunk, x  , y-1)
18                          - GET_PIXEL(chunk, x  , y+1);
19         GET_PIXEL(buffer, x, y) = BOUND(pixel, 0.0, 1.0);
20       }
21     }
22
23     // Swap buffers
24     SWAP(chunk, buffer);
25   }
26 }
```

Line 4 checks if the current iteration is a multiple of the border size and only performs border exchanges if it is. Furthermore, on line 12-13 the loop bounds are updated to include the halo, except from the outermost row/column, in the computations. This ensures that the values flowing from the halo into the chunk are also correct for iterations that don't include a border exchange. In addition to these changes we have to update the border exchange functions to exchange the whole halo, but that change is fairly straightforward and is not shown here. Finally, note that keeping the halo correctly updated using the above code requires access to the halo corners, which is discussed further in the following section.

Corners Cells

In some cases it is not sufficient to just exchange the immediate left, right, up and down borders. Consider for instance the operators shown in figure 6. These operators can be used for more precise edge detection that converges faster than when using the five-point laplacian operator. In addition the nine-point Laplacian operator, like the five-point version, can also be used to



(a) 9-Point Laplacian (b) Sobel operator

Figure 6: Stencil operators that use corner cells

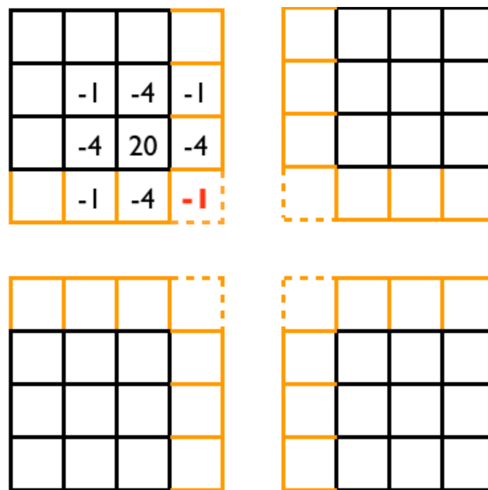


Figure 7: Two Dimensional Border Exchange with a nine-point stencil

solve systems of partial differential equations iteratively. Both the nine-point Laplacian and the Sobel operator require the value of the cells that are not a part of directly adjacent neighbors, but that comes from diagonal neighbors. Figure 7 shows how these cells are not exchanged with the schemes discussed so far.

Another case where we need to communicate cells from diagonal blocks is when we have wider borders than we need to in order to decrease the number of sends as explained in the previous section. In this case we need the corners of the ghost cell halo to (redundantly) update the values in the rest of the halo between the border exchange iterations.

The most common way to solve this problem is by performing the border exchanges along each dimensional axis as independent waves where each wave updates the halos in one direction. Consider the two dimensional border ex-

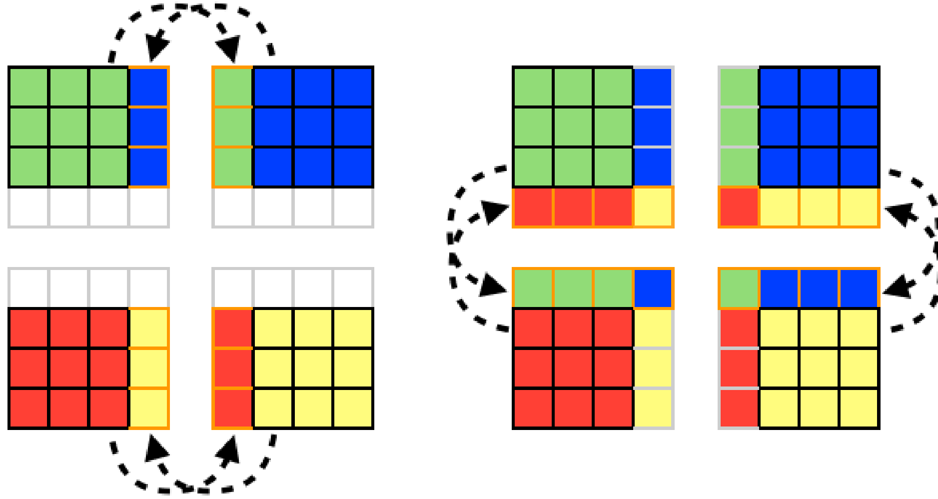


Figure 8: Two Dimensional Border Exchanges in two waves

change shown in figure 8. In the first wave the processes performs horizontal border exchanges and if the chunks are of size $n * n$ then they exchange n cells with their left and right neighbors. Therefore, when the second wave starts the processes have already received the borders from their horizontal neighbors and can include these in the vertical border exchanges, effectively folding the corner exchanges into the second wave. This wave will therefore exchange rows that are $n + 2$ wide (more if we have a *Wide Halo*) where the cells on each side of the border are the corners. For two dimensions this means that the halo corner will "automatically" be exchanged in the second wave, which saves us from having to perform four extra exchanges per chunk to exchange corners with diagonal neighbors.

Avoid Synchronization

When implementing a solution to an iterative grid problem using ghost cells *some* synchronization is required to ensure that the border exchanges complete before the next computation phase starts. That is, a certain amount of synchronization is inherent in the chosen solution. However, when implementing the solution it is desirable to avoid additional synchronization constraints that are not strictly necessary in order to get good performance. The implementation should perform as much synchronization as is necessary, but no more. This may sound obvious, but when using an API such as MPI many communication operations also serve as synchronization point, which

can slow down the implementation.

For example, the semantics of MPI_Send (and MPI_Sendrecv) is that it may block until the receiver starts to receive the message. The implementation is allowed to try to buffer the message and return before the matching receive is issued, but for portable programs one can not depend on this and must assume MPI_Send is a synchronization point. In addition even implementations that perform message buffering can run out of buffer space at which point they must revert to synchronous send operations.

Using synchronous send operations when performing border exchanges with n neighbors adds the constraint to the application that the sends have to be performed in the order in which they appeared in the code, even if no such ordering is required by the solution. Since most parallel machines don't have a direct communication path between every processor pair this means that the environment can't take advantage of the fact that the path to some of the neighbors might be free while the path to the receiver of the current send is busy [2].

The solution is to remove the unnecessary synchronization inherent in the synchronous sends and instead use asynchronous send operations. By doing this the environment is free to send those messages whose receiver lies at the end of a path that is not heavily loaded first and thus avoid bottlenecks in the communication network.

The border exchange function on page 7 performs deadlock-free communication with the western neighbor. This code is correct, but imposes an artificial ordering on the border exchanges that is not necessary. The following pseudo-code addresses this problem by replacing the synchronous MPI_Sendrecv with calls to the asynchronous and non-blocking functions MPI_Isend and MPI_Irecv [2]:

```
1  Do i=1, n_neighbors
2    MPI_Irecv(edge, len, MPI_REAL, nbr(i), tag,
3              comm, request(i), ierr);
4  Enddo
5
6  Do i=1, n_neighbors
7    MPI_Isend(edge, len, MPI_REAL, nbr(i), tag,
8              comm, request(n_neighbors+i), ierr);
9  Enddo
10
11 MPI_Waitall(2*n_neighbors, request, statuses, ierr);
```

The code starts by scheduling the receives of the borders from all the neighbors of the process. Since the receives are non-blocking the function calls returns immediately. Note that with non-blocking receives the user has

to supply the buffer to put the message in, which in this code is the edge argument. With MPI_Irecv this buffer will be filled with the message at some point in the future. One can query MPI for the status of the receive using MPI_Test or MPI_TestAny or ask MPI to block until the receive completes using MPI_Wait, MPI_Waitany or MPI_Waitall.

After posting all the receives the code schedules the matching sends to each neighbor using MPI_Isend. Like with MPI_Irecv MPI_Isend returns immediately and does not need to wait for the matching receive to be initiated. This means that the environment is free to send the messages in any order, which allows it to take advantage of lightly loaded network paths. Then, on line 11, a call to MPI_Waitall tells MPI to wait for the completion of all the message transmissions before continuing.

By restructuring our send operations in this way we have reduced the amount of synchronization to only what is actually required by the solution. That is, one synchronization point to ensure all the sends are completed before beginning the next computation phase as opposed to one synchronization point for every send.

Known Uses

The Ghost Cell pattern is most commonly used in distributed memory systems where processors can't access each others memory, but is also applicable to NUMA (Non-Uniform Memory Access) systems to increase locality. It is widely used in image processing as well as in structured grid computations such as weather and atmospheric simulations and fluid dynamics where physical effects are simulated by repeatedly solving systems of differential equations. In particular these simulations have one equation per point in 3D space and each equation depends on a set of neighboring points.

PETSc is a very popular and widely used framework for solving scientific problems modeled by partial differential equations that uses border exchanges to communicate ghost nodes [3]. It is widely used for scientific computations in areas ranging from nano-simulations, imaging and surgery to fusion, geosciences and wave propagation.

Another use is in cellular automata[2] where new generations of cells are repeatedly created based on the previous generation and certain rules of interaction. The rules governing the updates of each cell are based on the states of a certain set of neighboring cells from the previous generation (a

[2] A famous example of cellular automata is Conway's Game of Life

stencil). These computations can therefore use the Ghost Cell pattern to perform the communication between each generation [4].

Related Patterns

Geometric Decomposition [1] Computations that have been structured using the Geometric Decomposition pattern are often applied in iterations. In those cases the Ghost Cell pattern is almost always used to handle the communication between each iteration.

Structured Grids [1] Structured Grid computations are usually stencil computations performed in iterations. When these computations are parallelized the Ghost Cell pattern is a good fit to perform the communication required to provide the values for the stencils.

Sparse Linear Algebra [1] Iterative methods for systems of linear equations such as Jacobi and Gauss-Seidel require global communication in general. However, when applied to certain sparse linear algebra problems where the equations have few terms, such as systems of partial differential equations, the communication switches from global to local. Examples of such differential equations are the laplacian and poisson equations. In these situations the Ghost Cell pattern is a good candidate for performing the necessary communication between each iteration.

Iterative Refinement [1] The iterative refinement pattern perform successive refinements until some exit condition is met. If these computations are based on a stencil of neighbors then the Ghost Cell pattern is a good candidate for the communication of these neighboring values.

Unstructured Grids [1] Unstructured Grid computation, as the name implies, are less regular than Structured Grids, but often require regular communication similar to the ones described in the Ghost Cell pattern. However, issues specific to Unstructured Grids are not covered by this pattern.

Collective Communication Patterns [5] Like the Ghost Cell pattern the patterns in the Collective Communication pattern language deal with structured communication between several processors. However, the Collective Communication patterns deal with *global* structured communication while the Ghost Cell pattern deal with *local* structured communication and they are therefore different.

Wavefront [6] The Wavefront pattern can be used to parallelize dynamic programming problems. In this pattern one has a set of values in N dimensions that must be computed where each value, due to memoization, depends on the values of the left and upper neighbors from the *same* iteration. The computations thus take the form of a diagonal sweep that resembles a wavefront. Although this pattern share some of the characteristics of the Ghost Cell pattern it is different because the neighboring values are taken from the same iteration, not the previous one. Another difference is that the communication is one-way while in border *exchanges* they are two-way.

References

[1] Tim Mattson and Kurt Keutzer. *Our Pattern Language*. Retrieved November 26, 2009.
`http://parlab.eecs.berkeley.edu/wiki/patterns/patterns`

[2] William Gropp & Ewing Lusk. *Tuning MPI programs for peak performance* (1997). Argonne National Laboratory.
`http://www.mcs.anl.gov/research/projects/mpi/tutorials/perf/`

[3] *PETSc – Portable, Extensible Toolkit for Scientific Computation.* Retrieved November 26, 2009.
`http://www.mcs.anl.gov/petsc/petsc-as/index.html`

[4] Barry Wilkinson and Michael Allen. *Parallel Programming, Techniques and Applications Using Networked Workstations and Parallel Computers* (Pearson Prentice Hall, 2005).

[5] Nicholas Chen, et. al. *Collective Communication Patterns* (2009).
`http://parlab.eecs.berkeley.edu/wiki/_media/patterns/paraplop_g1_4.pdf`

[6] Eun-Gyu Kim and Marc Snir. *Wavefront Pattern.*
`http://www.cs.illinois.edu/~snir/PPP/patterns/wavefront.pdf`