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

ABSTRACT

Many problems consist of a structured grid of points that are updated repeatedly based on the values of a fixed set of neighboring points in the same grid. To parallelize these problems we can geometrically divide the grid into chunks that are processed by different processors. One challenge with this approach is that the update of points at the periphery of a chunk requires values from neighboring chunks. These are often located in remote memory belonging to different processes. The naive implementation results in a lot of time spent on communication leaving less time for useful computation. By using the Ghost Cell Pattern communication overhead can be reduced. This results in faster time to completion.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Performance

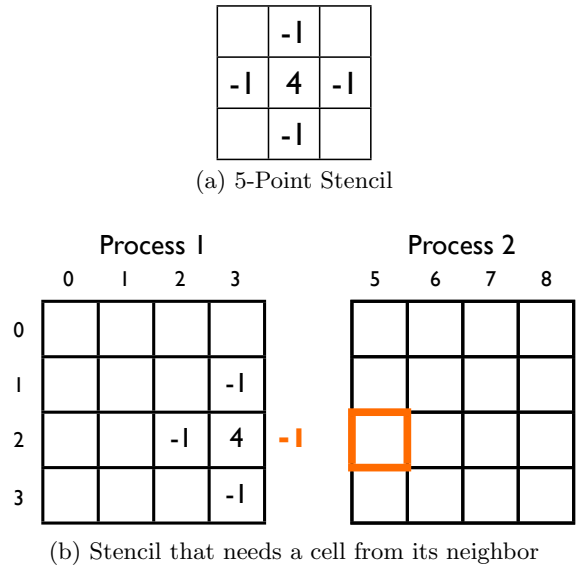
Keywords

pattern, ghost cells, border exchanges, structured grids

1. PROBLEM

Many problems can be modeled as a set of points in a structured grid that are updated in successive iterations based on the values of their neighbors from the previous iteration. These problems can be divided geometrically into chunks that are computed on different processors or cores. However, since updating a point requires the values of its neighbors, each chunk needs values from neighboring chunks to update the points at its borders. How can we communicate these values between processes in an efficient and structured manner?

ParaPLoP '10, March 30 - 31st, 2010, Carefree, AZ.



(a) 5-Point Stencil

(b) Stencil that needs a cell from its neighbor

Figure 1: Stencil computation in geometrically decomposed grids

2. CONTEXT

As mentioned above, many problems consist of 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 based on the values of their neighbors from the previous iteration (see *Structured Grids* and *Iterative Refinement* [6]). This pattern of computation is also often called a *convolution*.

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 we will use 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. In addition to detecting edges, the five-point Laplace operator can also be used to solve systems of partial differential equations iteratively.

Geometric Decomposition [6] 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 usually not a good solution as it introduces a lot of small communication operations in the middle of computation which leads to high latency costs on most current systems.

3. FORCES

Performance vs Complexity A tension exists 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 would severely hurt performance. You also have to consider which optimizations to implement. Examples of optimizations are trading computation for communication, avoiding unnecessary synchronization and overlapping communication and computation. These optimizations *may* increase overall performance, but do so at the cost of increased complexity and reduced maintainability.

Performance vs Portability Optimizations such as reducing communication at the expense of increased computation and avoiding unnecessary 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. Therefore, a fast application on one system may be a slow application on another thus reducing *performance portability*. In order to make it easier to move an application from system to system, parameters that are likely to differ should be made configurable through build or runtime options.

Cost Of Copying vs Contention and Locality On shared memory machines it is possible to avoid the copying associated with ghost cells by having a global set of points for the previous iteration that all the threads read from. However, this increases the likelihood of cache contention and false sharing as the processor cores read and write to the same cache lines. Additionally, on NUMA systems this results in some cores having to read data from remote memories.[1] Alternatively, the chunks could be kept separate in memory to avoid false sharing. However, this would reduce locality and cache utilization as the borders of the neighbors would not be located directly adjacent to the data of the chunk itself.

[1]NUMA, Non-Uniform Memory Access, means that there is one global memory address space, but that accessing different parts of it does not take the same amount of time. This is typically because there are several memories that are local to different cores.

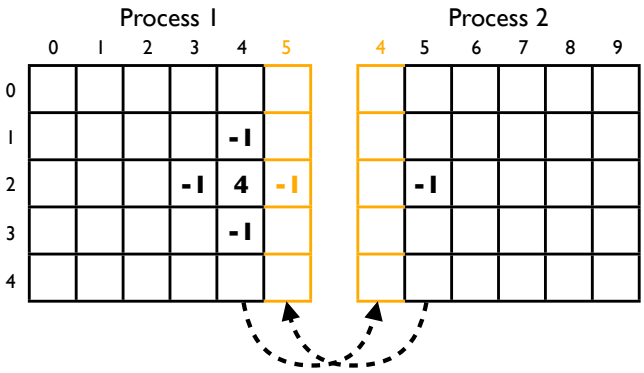


Figure 2: Each chunk receives a vector of ghost cells from neighboring chunks

Cost of Computation vs Communication Overhead On distributed memory systems it is possible to trade additional computation for less frequent communication by redundantly storing and updating cells that belong to a different process locally. This leads to some redundant computation, but may reduce time to completion if the overhead of send operations is high. However, the cost of performing the extra computation must be evaluated against the overhead cost of sending messages, both of which will differ between systems. This is discussed further in the context of this pattern in section 5.1.

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

Shape of Chunks The shape of the chunks also influence the surface to volume, or perimeter to area, ratio. In order to minimize it we want something close to a circle or a sphere, such as a hexagon. On the other hand it is often desirable to use a simpler shape, such as a square in order to decrease the implementation complexity. Another, common shape is a rectangle as it allows us to divide the grid along only one axis, but results in a worse perimeter to area ratio that division into squares.

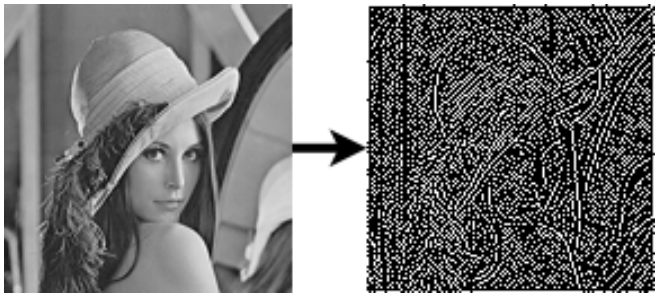


Figure 3: Edge Detection

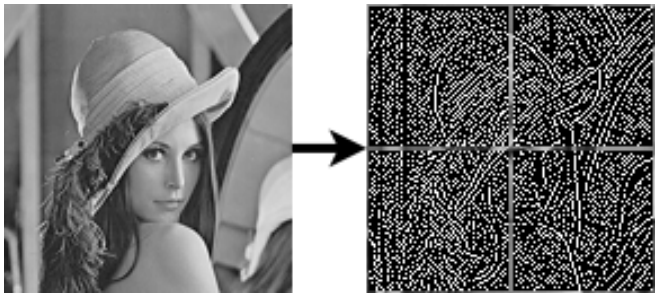


Figure 4: Edge detection *without* border exchanges

4. SOLUTION

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

If you have a stencil that extends beyond immediate neighbors or if you want to trade computation for communication then use a *Deep Halo* (section 5.1). If your stencil requires cells from diagonal neighbors then you must also exchange *Corner Cells* (section 5.2). If you need higher performance then *Avoid Unnecessary Synchronization* (section 5.3) or *Overlap Communication and Computation* (section 5.4).

5. 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 measures how much each point differ from its neighbors and figure 3 shows it applied many times to Lena. The sequential C code for computing the Laplacian on a gray-scale image is as follows:

```
1 void laplacian() {
2   for (int iter=0; iter < ITERATIONS; ++iter) {
3
4     // Compute the Laplacian
5     for (int y=1; y < (height-1); ++y) {
6       for (int x=1; x < (width-1); ++x) {
7         double pixel =
8               4 * GET_PIXEL(image, x  , y  )
9                 - GET_PIXEL(image, x-1, y  )
10                - GET_PIXEL(image, x+1, y  )
11                - GET_PIXEL(image, x  , y-1)
12                - GET_PIXEL(image, x  , y+1);
13        GET_PIXEL(buffer, x, y) =
14            BOUND(pixel, 0.0, 1.0);
15      }
16    }
17
18    // Swap buffers
19    POINTER_SWAP(buffer, image);
20  }
21 }
```

For each iteration of the outer loop the Laplacian is applied to every pixel of the image (line 5–16). GET_PIXEL on line 13 is a macro to retrieve the array location of the pixel at location (x,y) from the buffer and BOUND on line 14 caps the value of the given variable to the specified range.

Since the computations need the surrounding pixels from the *previous* iteration we cannot easily update the image in place and we will therefore use *double buffering*. This means that we have two sets of values; one for the current iteration and one for the previous. On line 19 we swap the buffers by using the POINTER_SWAP macro to swap the values of their pointers. Note that the code has been simplified for clarity by not computing the Laplacian at the image borders.

We will parallelize this code using the *Single Program Multiple Data* (SPMD) paradigm. We must first send roughly equal parts of the image to each process before calling the laplacian function and then merge these image parts again after it has completed. One call to the laplacian function will therefore only compute the Laplacian for a subset of the image.

However, this leads to the problem in figure 1(b), where each process needs pixels from the neighbors when computing its own border pixels. If we ignore the neighbor pixels we get the result shown in figure 4. This figure shows Laplacian edge detection applied to the image using four processors in a Cartesian grid 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 a border exchange for every iteration of the outer loop. The resulting code, written in C using MPI[1], can look like the following:

```
1 void laplacian() {
2   for (int iter=0; iter < ITERATIONS; ++iter) {
3
4     // Exchange borders with all four neighbors
5     exchange_horizontal_borders(); // MPI calls
6     exchange_vertical_borders();   // inside
7
8     // Compute the Laplacian
9     for (int y=1; y < (height+1); ++y) {
10      for (int x=1; x < (width+1); ++x) {
11        double pixel =
12              4 * GET_PIXEL(image_chunk, x  , y  )
13                - GET_PIXEL(image_chunk, x-1, y  )
14                - GET_PIXEL(image_chunk, x+1, y  )
15                - GET_PIXEL(image_chunk, x  , y-1)
16                - GET_PIXEL(image_chunk, x  , y+1);
17        GET_PIXEL(buffer, x, y) =
18            BOUND(pixel, 0.0, 1.0);
19      }
20    }
21
22    // Swap buffers
23    POINTER_SWAP(buffer, image_chunk);
24  }
25 }
```

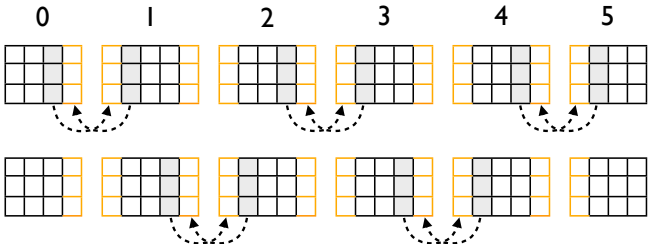


Figure 5: Deadlock-free border exchanges

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 at index 1 to the last cell before the right halo starts. Apart from that the biggest difference from the previous code listing is on line 5–6 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 code. Note that there is no need for a barrier here as all the necessary synchronization is implicit in the exchanges. When a process receives its data it is safe for it to proceed. The code listing below shows an implementation of the exchange_horizontal_borders() function:

```
1 void exchange_horizontal_borders() {
2   if (x_coord % 2 == 0) {
3     exchange_east_border();
4     exchange_west_border();
5   }
6   else {
7     exchange_west_border();
8     exchange_east_border();
9   }
10 }
```

To prevent a deadlock every other process performs border exchanges in the reverse order. This prevents all the processes from waiting for the west border before any of them sends the east border. If a process' x coordinate is divisible by two (line 2) it exchanges its east border before the west. If it is not divisible by two, then it exchanges the west border first. Figure 5 illustrates an example exchange with 6 processes in the x direction. The pairs <0, 1>, <2, 3> and <4, 5> exchange borders first by executing line 3 and 7 respectively. Then the pairs <1, 2> and <3, 4> exchange borders by executing line 4 and 8. Thus border exchanges always have a matching receiver and no deadlocks occur. The function exchange_vertical_borders() is implemented similarly.

The following code shows one possible implementation of the exchange_west_border() function that is called on line 4 and 7 in the previous listing:

```
1 void exchange_west_border() {
2   if (west_neighbor != -1)
3     MPI_Sendrecv(&GET_PIXEL(image_chunk, 1, 1),
4                  1, vertical_border_t,
5                  west_neighbor, TAG,
6                  &GET_PIXEL(image_chunk, 0, 1),
7                  1, vertical_border_t,
8                  west_neighbor, TAG,
9                  cartesian, &status);
10 }
```

On line 2 we specify that we will only perform a border exchange with the western neighbor if it exists. The MPI_Sendrecv function performs a send and a receive in a deadlock-free manner and is therefore the most obvious candidate to use for the exchange. However, as we shall see later in the *Avoid Unnecessary Synchronization* section it is not the most efficient one. The parameters on line 3–5 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 6–8 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 the noise-free edge detection we saw in figure 3.

5.1 Deep 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 that are computed by different processes. That solution used a border with a thickness of one since that is sufficient to correctly implement the five-point Laplacian. However, there are a two situations that either require or benefit from a deeper halo.

The first situation is when the stencil we want to apply to the grid reaches further than the immediate neighboring cells. In that case we *must* use a deep 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 and 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 depends 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 of 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. This way, if we started out with a halo of size one, we can limit the border exchanges to every *nth* iteration and then exchange n rows and columns. While the amount of data transfered is essentially the same, the number of messages decreases. Fewer messages means less messaging overhead, which reduces the cost of communication. Figure 6 demonstrates how this would work for a halo width of three.

The cost of increasing the halo's width is that we have to keep it updated locally, which adds to the computational work and storage requirements. This is shown as arrows in the figure. Still, *communication is expensive* and it is often

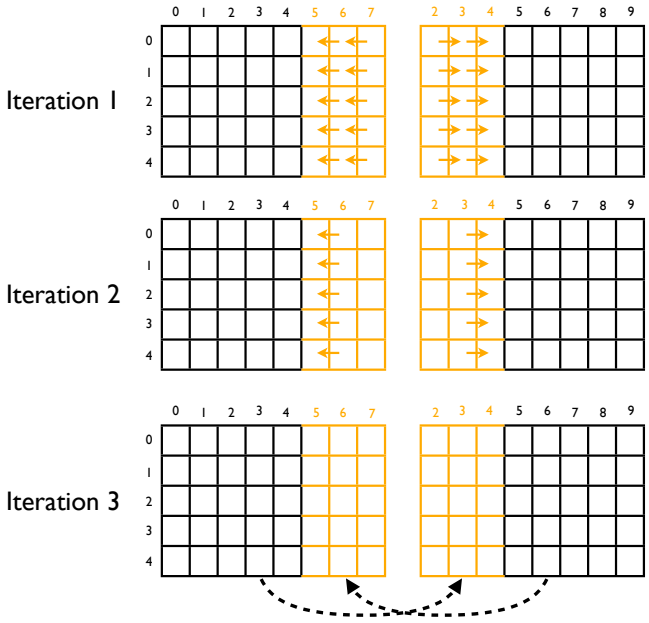


Figure 6: Deep Halo with a Border Exchange every *nth* iteration

beneficial to trade less frequent communication for 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 performance portability of the application.

Building on the edge detection example the we will now expand it to use deep borders to trade redundant 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
4     // Exchange borders with all four neighbors
5     if (iter % border == 0) {
6       exchange_horizontal_borders();
7       exchange_vertical_borders();
8     }
9
10    // Compute the Laplacian
11    for (int y=1; y<(height+2*border)-1; ++y) {
12      for (int x=1; x<(width+2*border)-1; ++x) {
13        double pixel =
14              4 * GET_PIXEL(image_chunk, x  , y  )
15                - GET_PIXEL(image_chunk, x-1, y  )
16                - GET_PIXEL(image_chunk, x+1, y  )
17                - GET_PIXEL(image_chunk, x  , y-1)
18                - GET_PIXEL(image_chunk, x  , y+1);
19        GET_PIXEL(buffer, x, y) =
20              BOUND(pixel, 0.0, 1.0);
21      }
22    }
23
24    // Swap buffers
25    POINTER_SWAP(buffer, image_chunk);
26  }
27 }
```

-1	-4	-1
-4	20	-4
-1	-4	-1

(a) 9-Point Laplacian

-1	-2	-1
0	0	0
1	2	1

+

-1	0	1
-2	0	2
-1	0	1

(b) Sobel operator

Figure 7: Stencil operators that use corner cells

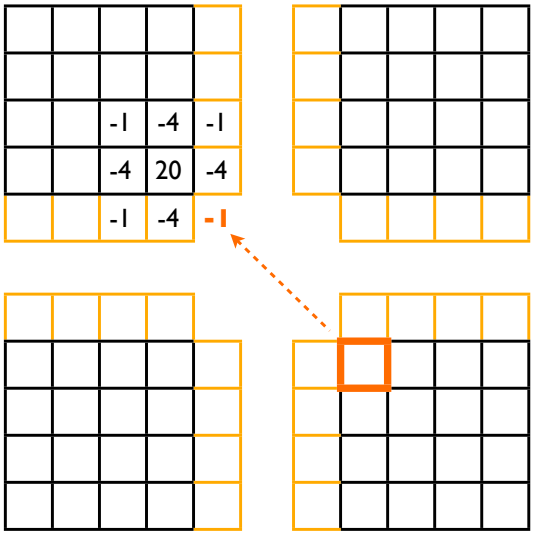


Figure 8: Two Dimensional Border Exchange with a Nine-Point Stencil

Line 5 checks if the current iteration is a multiple of the border size and only performs border exchanges when it is. Furthermore, on line 11–12 the loop bounds are updated to include the halo, except for 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 this we have to update the border exchange functions to exchange a larger 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. This is discussed further in the following section.

5.2 Corner 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 7. These operators can be used for more precise edge detection that converges faster than when using the five-point Laplace operator. Both the nine-point Laplacian and the Sobel operator require the value of the cells that that comes from diagonal neighbors. Figure 8 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 a deeper halo 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

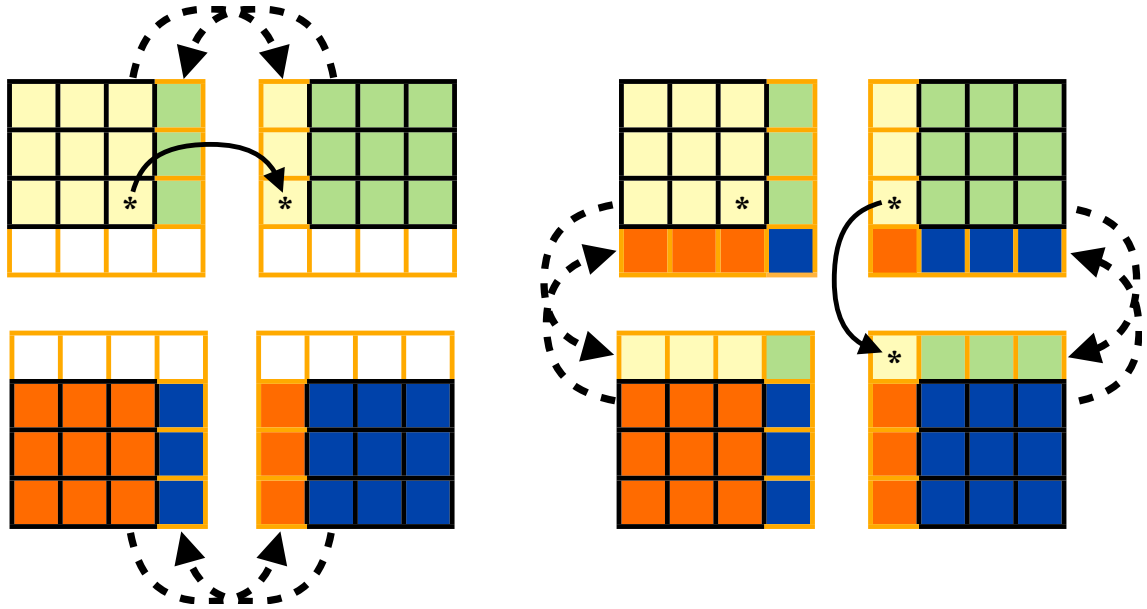


Figure 9: Border Exchange in two waves to copy corner cells across diagonals. One cell is marked to show its journey to the diagonal neighbor.

of the halo between the border exchange iterations.

A common way to solve this problem is to perform the border exchanges along each dimensional axis as independent *waves* where each wave updates the halos in one direction. Consider the two dimensional border exchange shown in figure 9. In the first wave the processes perform 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 corner cells from these in the vertical border exchanges. This effectively folds the corner exchanges into the second wave. This wave will therefore exchange rows that are $n + 2$ wide (more if we have a *Deep Halo*) where the cells on each side of the border are the corners. For two dimensional exchanges this saves us from having to perform four extra exchanges per chunk to exchange corners with diagonal neighbors.

5.3 Avoid Unnecessary 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 computations that need the ghost cells start. 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 implicitly serve as synchronization points. This 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. However, 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 cannot 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 [4]. Furthermore, it cannot take advantage of systems that can perform multiple sends in parallel.

The solution is to remove the unnecessary synchronization inherent in the synchronous sends by instead using asynchronous send operations. This leaves the environment free to send the messages in any order. It can, for example, send messages whose receiver lies at the end of a path that is not heavily loaded first and thereby avoid bottlenecks in the communication network. Additionally, since it is free to send messages in any order it can send multiple messages in parallel where this is supported. Finally, since avoiding unnecessary synchronization gives the environment more freedom to choose the best approach it increases performance portability as well.

In section 5 the functions exchange_horizontal_borders and exchange_vertical_borders are called to perform deadlock-free border exchange with all neighbors. Those functions are correct, but they impose an artificial ordering on the border exchanges. The following code addresses this problem by replacing the synchronous MPI_Sendrecv in the border exchange phase with calls to the asynchronous and non-blocking functions MPI_Isend and MPI_Irecv [4]:

```
1 void exchange_borders() {
2
3   // Start asynchronous ghost cell receives
4   for (int i=0; i < num_neighbors; ++i) {
5     MPI_Irecv(ghost_cells[i], len,
6               border_types[i],
7               neighbors[i], tag, comm,
8               &requests[i]);
9   }
10
11  // Start asynchronous border sends
12  for (int i=0; i < num_neighbors; ++i) {
13    MPI_Isend(borders[i],  len,
14              border_types[i],
15              neighbors[i], tag, comm,
16              &requests[num_neighbors+i]);
17  }
18
19  MPI_Waitall(2*num_neighbors, requests,
20              statuses);
21 }
```

The code first schedules the receives of borders from all the neighbors of the process into its ghost cell region. Since the receives are non-blocking the function calls returns immediately. Note that with non-blocking receives the user has to supply the memory that MPI will put the received data in. In this code the neighbor border data is placed directly into the ghost cell regions.

The semantics of MPI_Irecv only guarantees that this buffer will be filled with the message *at some point* in the future. Until that time the contents of the buffer is undefined. We can query MPI for the status of the receive using MPI_Test or MPI_TestAny. We can also 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 MPI_Irecv, it 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 and thus take advantage of lightly loaded network paths. Then, on line 19, 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.

5.4 Overlap Communication and Computation

In the previous sections we implemented edge detection by alternating between communication and computation.

We used synchronization calls, whether implicit such as MPI_Sendrecv or explicit like MPI_Waitall, to ensure that all processes had completed all communication before moving on to compute the next iteration. Thus the system would at any given point *either* be communicating or computing.

However, on many systems communication and computation can be done simultaneously. That is, a process can perform computation while waiting for a message to arrive at its destination. In MPI we can take advantage of this by using asynchronous functions, like MPI_Isend and MPI_Irecv.

In the previous section we used these functions to remove the need for sends and receives to happen in any particular order. We first registered all the sends and receives we needed with the MPI runtime and then we called MPI_Waitall to wait for their completion.

Since communication is expensive on most modern systems the processes are likely to spend significant time waiting at this synchronization point for all the border exchanges to complete. This leaves processor cores unused, wasting precious compute cycles. If we can find some useful work for the processor cores to do while they wait for the border exchanges to complete then we will avoid this waste. This means that the iterative algorithm will complete faster. In other words, we can hide some of the cost of communication by doing it in the background while we continue to make progress on the computation. This technique of hiding the communication cost is known as *latency hiding*.

In the ideal case we will have enough work to do to hide all of the communication. Furthermore, if the communication overhead is roughly equal to the time spent computing we can approach a 2x speedup using this technique.

The following code shows one approach to overlap communication with computation when using ghost cells and can be contrasted to the first parallel Laplacian implementation in the second code listing in section 5:

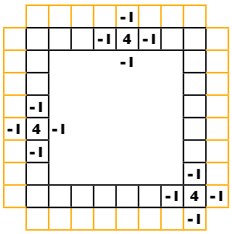
```
1 void laplacian() {
2
3   // Initial Border Exchange
4   exchange_borders();
5
6   for (int iter=0; iter < ITERATIONS; ++iter) {
7
8     // Compute the Laplacian for the borders
9     laplacian_borders();
10
11    // Start asynchronous ghost cell receives
12    // and border sends
13    exchange_borders_start();
14
15    // Compute the Laplacian for the interior
16    laplacian_interior();
17
18    // Block until all border exchanges completes
19    MPI_Waitall(2*num_neighbors, requests,
20                statuses);
21
22    // Swap buffers
23    POINTER_SWAP(buffer, image_chunk);
24  }
25 }
```

The laplacian function first performs an initial border exchange by calling the exchange_borders function from the previous section. This ensures that the ghost cells of the initial image_chunks are initialized so that the first iteration of the main loop can use them to compute the borders.

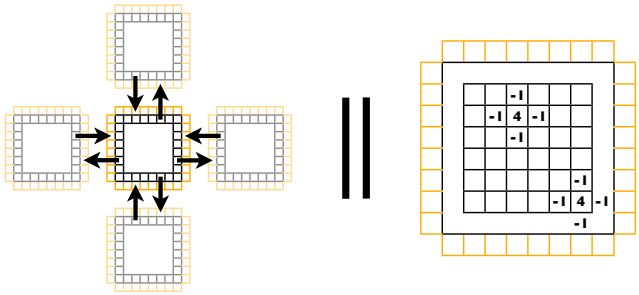
The main loop that starts at line 6 consists of five stages. The first stage computes the Laplacian for the borders of the image as shown in figure 10(a). The results are stored in the temporary buffer as before (recall that we are using double buffering).

After the borders of the image have been computed and stored in the temporary buffer the algorithm is finished with them and can start transferring them to the neighbors' ghost cell regions. This transfer is started in the second stage by calling the exchange_borders_start function. This function is similar to the exchange_borders function, except that it only initiates the sends and receives and does not call MPI_Waitall to wait for their completion. It is therefore an asynchronous function.

Now that the border exchanges have been initiated we



(a) Compute Borders



(b) Exchange Borders and Compute the Interior in Parallel[3]

Figure 10: Overlapping Communication and Computation

want to use the time until they complete to do something useful. Since the code has only computed the Laplacian for the borders we have the entire interior left to compute and we can do this while the borders are transfered across the interconnects. The third stage therefore computes the Laplacian for the interior. Thus, the borders exchanges and the interior computations are performed in parallel and they are both depicted in figure 10(b).

When the code is done computing the interior we have no more work to do so we call MPI_Waitall to wait for all the border exchanges to complete.

Once the border exchanges are done the temporary buffer contains a complete new version of the image_chunk with the Laplacian applied and the algorithm therefore swaps the buffers to set the stage for the next iteration.

6. KNOWN USES

The Ghost Cell pattern is most commonly used in distributed memory systems where processors cannot access each others memory. However, it is also applicable to shared memory and NUMA (Non-Uniform Memory Access) systems to increase locality. It is widely used in image processing as well as in various structured grid computations. Examples of the latter include weather and atmospheric simulations and fluid dynamics where physical effects are simulated by repeatedly solving systems of differential equations using a method such as the Jacobi method. Specifically, these simulations have one equation per point in 3D space and each equation depends on a set of neighboring points.

Although we have focused on the use of Ghost Cells in structured grid computations the pattern is also commonly used when parallelizing iterative algorithms involving unstructured grids and graphs.

[3]The || symbol is often used to denote that two things happen in parallel.

PETSc is a popular framework for solving scientific problems modeled by partial differential equations that uses border exchanges to communicate ghost nodes [2]. It is widely used for scientific computations in areas ranging from nanosimulations, imaging and surgery to fusion, geosciences and wave propagation. By using a framework like PETSc where possible you can avoid having to re-implement this pattern yourself.

Another use is in cellular automata where new generations of cells are repeatedly created based on the previous generation and certain rules of interaction.[4] The rules governing the updates of each cell are based on the states of a pre-determined set of neighboring cells from the previous generation (a stencil). These computations can therefore use the Ghost Cell pattern to perform the communication between each generation [7].

7. RELATED PATTERNS

Shared Data[6] In shared memory systems Shared Data is an alternative to the Ghost Cell pattern. Using this pattern a global grid can be shared between multiple threads through shared memory instead of distributing chunks of it to different processes.

Scratchpad This is a common alternative to the Ghost Cell pattern on GPUs. On some of these systems, the processing cores have local memories,[5] but data can not be moved directly between these. The Ghost Cell pattern can therefore not be used. In the Scratchpad pattern the whole grid is maintained in global memory. Chunks of this, including the ghost cell halos, are moved to local memories for computation, but are moved back to global memory before the start of the next iteration.

Geometric Decomposition[6] Data structures like grids that have been decomposed using the Geometric Decomposition pattern are often used for iterative stencil computations. In those cases the Ghost Cell pattern is almost always used to handle the communication between each iteration.

Structured Grids[6] 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.

Unstructured Grids[6] Unstructured Grids are, as the name implies, less regular than Structured Grids. The Ghost Cell pattern is nonetheless also useful when solving problems involving these. However, issues specific to Unstructured Grids are not covered here.

Sparse Linear Algebra[6] Iterative methods for systems of linear equations such as Jacobi and Gauss-Seidel require global communication in general. However, when applied to certain problems in sparse linear algebra where the equations have few terms, such as systems of

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

[5]These are often called shared memories, which is the term used in the CUDA programming model.

partial differential equations, the communication goes 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 necessary communication between each iteration.

Graph Algorithms[6] Graph Algorithms sometimes take the form of iterative stencil computations and are parallelized by partitioned the graph and distributing the subgraphs to different processes. In these cases the Ghost Cell pattern is a strong candidate for performing the necessary communication between each iteration. However, issues specific to Graph Algorithms are not covered here.

Iterative Refinement[6] 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.

Collective Communication Patterns[3] 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[5] 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 the Wavefront 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 it is two-way in border *exchanges*.

8. REFERENCES

[1] The message passing interface (mpi) standard. `http://www.mcs.anl.gov/research/projects/mpi/`, Retrieved June 14 2010.

[2] Petsc – portable, extensible toolkit for scientific computation. `http://www.mcs.anl.gov/petsc/petsc-as/index.html`, Retrieved June 14 2010.

[3] N. Chen, R. K. Karmani, A. Shali, B.-Y. Su, and R. Johnson. Collective communication patterns. `http://parlab.eecs.berkeley.edu/wiki/_media/patterns/paraplop_g1_4.pdf`, April 30 2009.

[4] W. Gropp and E. Lusk. Tuning mpi programs for peak performance. `http://www.mcs.anl.gov/research/projects/mpi/tutorials/perf/`, 1997.

[5] E.-G. Kim and M. Snir. Wavefront pattern. `http://www.cs.illinois.edu/~snir/PPP/patterns/wavefront.pdf`, Retrieved June 14 2010.

[6] T. Mattson and K. Keutzer. Our pattern language. `http://parlab.eecs.berkeley.edu/wiki/patterns/patterns`, Retrieved June 14 2010.

[7] B. Wilkinson and M. Allen. *Parallel Programming, Techniques and Applications Using Networked Workstations and Parallel Computers*. Pearson Prentice Hall, 2005.