MPI Multigrid Project

Final Report

Jason Greco & Marlena Fecho

Goals
This project is an MPI implementation of the NAS NPB Multigrid benchmark, starting from the FORTRAN OpenMP version, ported to C by Hallgeir Lien. Our goal is to demonstrate improved performance over the original OpenMP implementation by taking advantage of the finer level of control available in MPI.

Background

Multigrid Methods [1]

Multigrid methods are a class of algorithms which approximately solve linear systems by recursively solving over multiple resolutions. Solving at a coarse resolution provides a rough approximation, which is a good initial guess for solving on finer resolutions. This improves the convergence over single-resolution methods.

![Figure 1: Multigrid is a multi-resolution method which uses coarse resolution solutions to improve convergence of finer resolution problems.](image)

We are given a problem, $Au = f$, where $u$ is the vector of unknowns. For an initial guess $v = u + e$, we have $A(u + e) = f + r$, where $e$ is the error and $r$ is the residual. Because this is a linear system, solving $Ae = r$ is equivalent to solving $Au = f$. So we can approximately solve (or “smooth”) $Ae = r$ (with an initial guess $e = 0$) on a course resolution to improve our initial guess $v$.

A key observation leading to multigrid methods is that smoothing the residual causes the high frequency components to converge fairly quickly, while the low frequency components - namely, bias - converge much more slowly. However, when the residual is projected on a coarser resolution, these low frequency components appear to be higher frequency. So by smoothing at
multiple resolutions, multigrid methods attain broad-spectrum convergence faster.

The recursive multigrid algorithm - called “V-cycle” for the shape of the recursion - estimates the error $e$ for an initial guess $v$. In the “down-cycle”, the residual of $v$ is progressively smoothed and “restricted” where restricting refers to projecting onto a coarser resolution. In the “up-cycle”, the residual is progressively smoothed, “prolongated” and used to update the error estimate. “Prolongating”, or “interpolating” means projecting onto a finer resolution. Finally, the error estimate is used to correct the initial guess $v$. It’s important to note, that although this is a recursive algorithm, it is more efficient and more parallelizable to implement it iteratively, and because there is no branching, it is relatively straightforward to do so.

**Serial Implementation**

Our project is based on the NAS NPB Multigrid benchmark [2], which was ported to C and parallelized with OpenMP by Hallgeir Lien. The benchmark approximates the solution of a 3D discrete Poisson equation with a multigrid method. This implementation uses a slightly modified V-cycle which, on the down-cycle, restricts the residual without smoothing (calculating the pseudo-inverse).
Method

Parallelization
We chose a 1D blocked approach, in which we subdivide the grid into “slices” along the z-axis, leaving the x and y dimensions of each slice at their original sizes. While this approach requires slightly more data to be transferred overall than a 3D blocked approach, it reduces the total number of messages required to share all boundary data (each processor has at most 2 neighbors in the sliced approach, versus the 6 neighbors in the 3D blocked approach). We predict that interprocessor communication will be latency bound, so fewer larger messages are preferable to more smaller messages. Also, the 1D approach is simpler to implement, and with the time constraints of the project, this was favorable.

![Image](image.png)

Figure ##: left: 1D blocked approach; right: 3D blocked approach

Constraints
Due to some time constraints of our project and the limitations of our starter code there are a couple notable constraints for the multigrid benchmark to accurately run. The first constraint is that our problem size must be evenly divisible by either $2^{\text{depth}}$, or be some power of two smaller than $2^{\text{depth}}$. This constraint is present in the serial implementation because it is not set up to properly handle interpolating from a problem size of $N$ to a problem size of $2 \times N + 1$ and so logically this limitation is also in our MPI implementation.

The second constraint is that our problem size must be evenly divisible by the number of processors, and if we have more processors than our problem size at the coarsest resolution then our number of processors must be evenly divisible by 2 for every level of coarseness beyond the level where there is one processor per set of data on the z-axis.

Reducing the Number of Cores used on Coarse Grids
In the down-cycle, the problem is projected on progressively coarser grids. Once the z-dimension of the grid is smaller than the number of processors, there is not enough work to keep all the processors busy, nor to justify the communication overhead. Instead, we temporarily turn off unneeded processors, and have the remaining ones communicate with each other. In the up-cycle, the problem is interpolated back onto finer grids. As needed, we turn the idle processors back on during our interpolation. We accomplish this by reversing our process
of turning them off where each time our global problem size grows we double the number of active processors until we run out of processors to allocate. Each processor that begins the interpolation routine on calculates its own interpolated data, and the interpolated data for a processor that is left of it using the values in its data buffer and then transmits the interpolated data to the processor that is turning on through MPI messages.

```c
rprj3   //Projecting to a coarser problem size
{
    proc_step = max(1, proc.p / Nz_curr)
    if ((proc.my_rank-1) % proc_step != 0) then proc.on = false;
    return if processor is off
    Do the same data calculations here as the serial case here
    call comm3
}

void interp(Matrix z, int mlj, int m2j, int m3j, Matrix u_,int mlk,int m2k,int m3k)
{
    Matrix u = u_;
    if proc is on and other processors are being turned on:
        u = u_temp
    Do same computations here as serial implementation
    if processors are being turned on:
    {
        step = num_procs / current_z
        left = (rank - step + num_procs) % num_procs
        right = (rank + step + num_procs) % num_procs
        if proc is already on
        {
            unpack results from u_extra into u_ and interpBuf
            MPI_Send(interpBuf, left);
        }
        else if proc is turning on this time
        {
            MPI_Recv(interpBuf, right)
            turn processor on
        }
    }
}
```

*Alternate Approach: Moving the Grid Onto Fewer Cores Sooner*

We initially considered shifting the problem onto fewer cores before the per-core work was down to 1 element wide. In order to decide at what point the communication overhead exceeded the computation that would need to be done if the data was split between fewer cores we ran some sample problems for very small data sets and found that our code was only slower when we
reached the point where each processor only holds one plane of data on the Z axis, and was slightly faster running with two planes of data per core as opposed to four planes of data per half as many cores. Because these tests did not include the time needed to condense the data by transferring between processors it means we would have to take a pretty similar communication hit to condense our data. Using this information we decided to instead only reduce cores when we could no longer evenly divide up the Z axis.

**Communication**

In this implementation, smoothing and the residual computation are $3 \times 3 \times 3$ convolutions. This requires that all processors maintain a 1-element-wide buffer which mirrors the boundary cells of neighboring processors. In the one processor case, this is still a concern with global boundaries. In the serial implementation, the left buffer mirrors the right boundary, the right buffer mirrors the left boundary, and so on for the remaining dimensions.

It was fairly simple to extend this code to perform interprocess mirroring. In the 1D blocked approach processor boundaries consist of $N_x \times N_y \times 1$ slices. In the communication method, all processors send each of its two boundary slices to the processor which boarders the slice. This mirroring occurs after restricting, smoothing, and calculating the residual.

Interpolation, however, must be handled differently. If processors were turned off during the down-cycle, then they must be turned back on during interpolation. Because the grid size doubles during interpolation, each processor which is already on must compute the data for two processors. The processor which is being turned on is then sent all its data, via MPI message, from a neighboring processor which was already on.

```c
void comm3(Matrix u, int n1, int n2, int n3)
{
    if processor is off: return
    if we are not running serially:
    {
        step = max(1, num_procs / current_z_size)
        left = (rank + num_procs - step) % num_procs
        right = (rank + step) % num_procs

        MPI_Irecv(recBufl, left, reqL)
        MPI_Irecv(recBufR, right, reqR)
    }

    set X boundaries the same as serial code
    set Y boundaries the same as serial code

    if we are not running serially:
    {
        for i2=0 to n2-1:
        {
            for i1=0 to n1-1:
            {
```
sendBufL[i2*n1 + i1] = u[1][i2][i1];
sendBufR[i2*n1 + i1] = u[n3-2][i2][i1];
}
)

MPI_Send(sendBufL, left)
MPI_Send(sendBufR, right)

test to see if either the left or right receive is done
if left receive is done first:
{
  for i2=0 to n2-1:
    for i1=0 to n1-1:
      u[0][i2][i1] = recBufL[i2*n1 + i1]

  wait for reqR to finish

  for i2=0 to n2-1:
    for i1=0 to n1-1:
      u[n3-1][i2][i1] = recBufR[i2*n1 + i1];
  }
else
{
  process right first, then left
  }

else: // only one proc
  set Z boundaries the same as serial code
  }
}

Calculating the L2 Norm
Since all of our grids are now divided up among processors in order to calculate the L2 norm for any of them we had to modify the norm2u3 routine. This routine now sums the squared value of each non boundary element in the matrix passed to the function and then through the MPI_Gather routine passes the sums from each of the cores to our root processor which then uses the global matrix dimensions to calculate the final L2 norm.

Pseudo Code:

double norm2u3(Matrix r, int n1, int n2, int n3, int nx, int ny, int nz)
{
  for i3=1 to n3-1:
    for i2=1 to n2-1:
      for i1 = 1 to n1-2:
        norm += (r[i3][i2][i1])^2

  MPI_Gather norms into rec_buff on root processor

  if processor is root then

norm = sum of rec_buff entries

norm = sqrt( norm / (nx*ny*nz ))

return norm

Testbed

Our performance results are all performed on the Trestles system at SDSC with the following specifications per node [3]:

Processors:
    Model: AMD Magny-Cours
    Cores: 8
    Clock Speed: 2.4 GHz
    Processors per node: 4

Memory:
    Capacity: 64 GB DDR3
    Bandwidth: 171 GB/s
    Link Bandwidth: 8 GB/s
    MPI Latency: 1.3 µs

Total Nodes: 324

Results

Strong Scaling: Running Time (in seconds)
size: 256 x 256 x 256    depth: 8    niters: 4

<table>
<thead>
<tr>
<th># cores</th>
<th>OpenMP</th>
<th>MPI</th>
<th>MPI (nocomm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.67</td>
<td>3.72</td>
<td>3.69</td>
</tr>
<tr>
<td>2</td>
<td>1.50</td>
<td>1.91</td>
<td>1.86</td>
</tr>
<tr>
<td>4</td>
<td>0.772</td>
<td>1.11</td>
<td>0.970</td>
</tr>
<tr>
<td>8</td>
<td>0.457</td>
<td>0.650</td>
<td>0.583</td>
</tr>
<tr>
<td>16</td>
<td>0.459</td>
<td>0.375</td>
<td>0.301</td>
</tr>
<tr>
<td>32</td>
<td>0.415</td>
<td>0.241</td>
<td>0.178</td>
</tr>
<tr>
<td>64</td>
<td>0.818*</td>
<td>0.172</td>
<td>0.106</td>
</tr>
</tbody>
</table>

* On OpenMP with 64 cores, this poor performance likely indicates that this only ran the problem on a single 32-core node
Weak Scaling: Running Time (in seconds)

size: 256 x 256 x Nz  depth: 8  niter: 4

<table>
<thead>
<tr>
<th># cores</th>
<th>Nz</th>
<th>OpenMP</th>
<th>MPI</th>
<th>MPI (nocomm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>256</td>
<td>2.57</td>
<td>3.71</td>
<td>3.60</td>
</tr>
<tr>
<td>2</td>
<td>512</td>
<td>3.25</td>
<td>3.75</td>
<td>3.73</td>
</tr>
<tr>
<td>4</td>
<td>1024</td>
<td>3.52</td>
<td>3.76</td>
<td>3.75</td>
</tr>
<tr>
<td>8</td>
<td>2048</td>
<td>3.73</td>
<td>3.92</td>
<td>3.78</td>
</tr>
<tr>
<td>16</td>
<td>4096</td>
<td>4.72</td>
<td>4.14</td>
<td>4.18</td>
</tr>
<tr>
<td>32</td>
<td>8192</td>
<td>10.9*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

* With 32 cores, both the original code and our MPI implementation experienced abnormal behavior causing it to not run correctly
Analysis

Although our code initially runs slower than the OpenMP version, as we increase the number of cores we observe better scaling, particularly in the strong scaling study. The large difference we see in strong scaling indicates that as our problem size becomes small compared to the number of processors we do a much better job of handling that situation. This is important because in these cases there needs to be much better management of communication since it becomes a larger percentage of the program’s running time.

In the strong scaling study, on 8 or more processors, some processors are being turned off at the coarsest levels. This should slightly impact scalability, although the majority of execution time is spent at the finer grain grids because that’s where the majority of the computation is. In the weak scaling study, we don’t encounter this, because the problem size per processor is constant.

We encountered problems with the 32-core weak scaling case. The OpenMP implementation ran much slower than expected. Also for this case, the MPI implementation timed out (given 5 minutes wall-clock time). Due to time constraints, we were unable to determine whether the MPI execution deadlocked or was just slow. It is yet unclear whether these problems were related.
Conclusions

While our final code does scale much better than the OpenMP implementation as we increase the number of cores it is important to consider development time as well whenever actually constructing parallel code. We believe that if you have very limited computational resources relative to development manpower, or if execution time is the most important goal, then MPI is a great way to squeeze out extra performance especially with large numbers of processors. OpenMP on the other hand requires very little modifications to serial implementations to properly function and even provided better performance in cases with small amounts of processors. In our opinion this makes OpenMP preferable over MPI when you have fewer development hours to spend, or the final code won't be run on more than a few cores.

Modified Goals/Setbacks

It took a lot longer than expected to fully understand the serial code. Even after we thought we had a solid understanding, specific (poorly named) variables kept impeding our progress. We eventually had to clean up a lot of unused, or unnecessarily used variables to make the code more manageable. We also found and fixed several bugs in the original code (a few of which only manifest with problems of non-cubic size).

If we were to continue work on this in the future, it would be been nice to implement some more advanced optimizations for comparison, such as a 3D blocked approach where we subdivide our problem into cubes instead of sheets, or a stencil optimization where we use a sliding window that helps reuse values between computations to optimize cache efficiency. We would have also liked to better overlap our computation and communication by wrapping communication around unrelated computations through MPI non-blocking calls, however we don’t see this providing a huge performance benefit because of how closely we match our MPI without communication execution.

Lessons Learned

We struggled with understanding the base code for a while. Eventually, we made a list of all the frequently-used variable names and wrote down their meanings and where they were used. Once we could see them all in one place, it became clear that many of them were unused, or unnecessary, and we were able to clean up the original code. We’re really happy with this strategy, and will likely use it in the future.

Having the initial serial code turned out to be both a blessing and a curse. Although it did provide us with a very efficient multigrid algorithm as a starting point, as a wise man once said, “Hell is other people’s code.” [4][5] However, being able to read, understand and adapt other people’s code is a vital skill, and we have gained a lot more experience in the course of this project.
References


http://www.sdsc.edu/us/resources/trestles/

[4] Michael Curry, Hell is other people’s code
http://exodusdev.com/blog/mike/hell-is-other-peoples-code

http://abstrusegoose.com/432
Team Self Evaluation Form

If you are working in a team on a project, then submit one copy of this self evaluation form. The members of each team should discuss how they worked together and what to write for the evaluation.

(1) List the names of your team members:
   A: Jason Greco
   B: Marlena Fecho

(2) Estimate how much time each team member devoted to this project.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>meetings</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>coding</td>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>writeup</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>planning (alone)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total (including meetings)</td>
<td>74</td>
<td>74</td>
</tr>
</tbody>
</table>

Things we did:
1. Fixed bugs in starter code
2. Separate full and per-processor sizes
3. MPI comm3
4. MPI norm2u3 (L2Norm)
5. Parallel zran3
6. Turn off procs in rprj3
7. Turn on procs in interp

(3) On an attached page, describe (a) what the major responsibilities of each team member were, (b) whether or not you met your milestones (c) the major strengths and weaknesses in how your team worked together, (d) the lessons learned from these events, (e) whether and how you plan to modify your milestones and how your team is organized, and (f) anything else that comes to mind.
(a) Jason: MPI gathering to compute L2Norm  
Marlena: MPI communication for boundary data  
Both: understanding algorithm, understanding serial code, planning approach, reduction of cores implementation, debugging each other's code, testing

(b) It took a lot longer than expected to fully understand the serial code. Even after we thought we understood, specific (poorly named) variables kept impeding our progress. We eventually had to clean up a lot of unused, or unnecessarily used variables to make the code manageable. We also found and fixed a few bugs in the original code that only manifest with problems of non-cubic size.

(c) Our greatest strength is our communication. We discuss all approaches before and while programming. We always work in the same room, so as issues come up, we’re able to bounce ideas off each other and offer debugging support, as well as coordinate repository commits. Our major weakness was that we grossly underestimated the ramp-up time required to fully understand the algorithm and base code.

(d) We struggled with understanding the base code for a while. Eventually, we made a list of all the frequently-used variable names and wrote down their meanings and where they were used. Once we could see them all in one place, it became clear that many of them were unused, or unnecessary, and we were able to clean up the original code. We’re really happy with this strategy, and will likely use it in the future.

(e) We reached all our milestones, however we did slightly modify our approach for improving upon the naive method with core reduction after doing some testing to see what the real communication impact was.

(f) Having the initial serial code has been both a benefit and a curse, since while it does provide us with a very efficient multigrid algorithm, the code itself was not always clear in what it was doing, and had many instances of unnecessary sections of code that were likely once used but never cleaned out of the final implementation.