Graham Wing

UID: 004738350

Lab 2 Report

Partitioning:

Data and computation are partitioned based on the same parallel blocking scheme I used in lab 1. Each process gets its own subset of matrix A and all of matrix B. Matrix A is partitioned row-wise, so that every process can take advantage of locality within its portion of the matrix. This also means that the computation performed on matrix C will also benefit from locality since the row index of C is determined by the row index of A.

The basic idea is to distribute disjoint rows of A among the processors and broadcast B to all the processors. In my final version of the program, this is done by sending block size chunks of the matrices. Each outermost loop iteration, process 0 sends each other process their respective portion of A for computation. Each process then computes the values for C and sends it back to process 0. B is distributed by sending a block of B needed for computation on A until each process has all of B. This means that B can be sent entirely during the first outermost iteration, then reused in the future. This methodology combined with nonblocking sends and receives allows computation to proceed without having to wait for each process to completely sync up with the others.

Different APIs:

The performance of blocking sends and receives is comparable to that of nonblocking. Buffered however should (theoretically) experience a loss in performance compared to the other two. This is due to explicit requirement for MPI_Bsend to allow access to the send buffer immediately after the call. This means that the data will likely be copied to another buffer (that must be allocated beforehand). Because the matrices in question are very large and require more space than is available in cache, this copying process will likely be very expensive. Thus, MPI_Bsend should perform the worst.

My initial solution was to use blocking MPI_Send and MPI_Recv which achieved a roughly 106 GFlops (for a 4096^3 problem size). This was accomplished by taking the blocking code from Lab 1, adjusting the bounds for each process and placing blocking communication before and after the computation (see Figure 1)

```
int num_rows_per = kI/mpi_size;

if(mpi_rank == 0){
    for(int i = 1; i < mpi_size; i++){
        MPI_Send(a + (num_rows_per * i), num_rows_per * kK, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
        MPI_Send(b, kK*kJ, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
    }
}
else{
    MPI_Recv(a_portion + (num_rows_per * mpi_rank), num_rows_per * kK, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(b_portion, kK*kJ, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Figure 1: Code snippet of blocked send and receive. Collecting solutions into C follows similar idea.

Surprisingly, using MPI_Gather and MPI_Scatter (particularly Gather) resulted in a large performance loss. This was very surprising to me as I would expect that allowing results to gather in any order would have equivalent if not slightly better performance.

My final version (as mentioned in the section above) interwove communication and computation. Looking at Figure 2, we can see that A is being preemptively sent before the other processes need it and C is being requested after each process has completed the requested section. This methodology vaguely syncs the different processes to be within the same outermost loop of each other. The same approach is used for distributed B.

```
//send current portion of a
MPI_Waitall(mpi_size-1, a_requests, MPI_STATUSES_IGNORE);
a_requests = new MPI_Request[mpi_size-1];
//receive c
if(vertical > offset + VERT_BLOCK_SIZE){
    MPI_Waitall(mpi_size-1, c_requests, MPI_STATUSES_IGNORE);
    c_requests = new MPI_Request[mpi_size-1];
}
for(int proc = 1; proc < mpi_size; proc++){
    //receive last part of c
    if (vertical > offset){
        MPI_Irecv(c + (proc * num_rows_per) + vertical - VERT_BLOCK_SIZE, VERT_BLOCK_SIZE * kJ, MPI_FLOAT, proc, 0, MPI_COMM_WORLD, &c_requests[proc-1]);
    }
    //send next portion of a
    if(vert_limit + VERT_BLOCK_SIZE <= (offset + num_rows_per)){
        MPI_Isend(a + (proc * num_rows_per) + vert_limit - offset, VERT_BLOCK_SIZE * kK, MPI_FLOAT, proc, 0, MPI_COMM_WORLD, &a_requests[proc-1]);
    }
}
```

Figure 2: Code snippet of the outermost loop for process 0. Notice that requests for Matrix A are waited on, then 0 waits for C, before requesting more of C and sending more of A.

This version achieved performance of roughly 111 GFlops (problem size of 4096^3).

Performance:

Problem Size	Performance (GFlops)
1024^3	76.519
2048^3	98.649
4096^3	112.415

Figure 3: Testing was performed using 4 processes.

The reduced performance for smaller problem size is due to the higher relative cost of communication. Because the matrices are smaller, the computation within each block finishes faster and there is more opportunity for a process to waste time waiting for communication. Whereas with the larger problems, a larger portion of time is spent computing, making the cost of communication less visible. This also results in the smaller problems have more erratic performance.

Scalability:

Number of Processes	Performance (GFlops)
1	31.378
2	60.665
4	112.415
8	90.599
16	12.058
32	5.966

Figure 4: Scalability measurement. All testing performed on 4096 cubed problem size.

The program scales well within the number of available physical processors (in our case 4). The relation is not completely linear; however, this is to be expected due to the added communication overhead that increasing the number of processors requires. Once beyond the number of physical cores, the scalability completely falls apart. Even within the number of virtual CPUs the scalability falls off. This is presumably due to the usage of communication rather than shared memory inhibiting full utilization of the hyperthreads. Communication requires the thread to be simultaneously active but making any computational progress, leading to a large loss in performance (activity on one thread limits activity of the other thread on the shared core).

Once process count exceeds the number of threads available, the performance becomes terrible (scaling almost inversely). This is due to the increase overhead of communication proportional to performance. Each step of computation gets smaller and requires an increasing amount of communication as process count increases. This naturally causes the inverse relationship seen above. Additionally, because there are more processes than available threads, a process that is causing the root process to wait might not be executing. This causes everything to grind to a halt because there are too many processes for the hardware to support.

Another note, at a certain number of processes, the block size will be larger than the amount of data that each process should be given (in the case of the above problem size, 64 processes). Because I use integer division, this causes the amount of data to be sent to be 0. To resolve this problem, I reduce the number of processes used (see Figure 5), effectively ceasing scale as the process count grows.

```
//incase rows per process goes to 0, decrease the number of processors
//should improve performance since less overhead for such a small problem size
while(mpi_size != 1 && kI/(mpi_size * VERT_BLOCK_SIZE) == 0) mpi_size = mpi_size/2;

//get rid of the unnecessary processes
if(mpi_rank >= mpi_size) return;
```

Figure 5: Solution to process count problem. Decreases the process count so that program may execute correctly.

Comparison to OpenMP:

The MPI implementation was much worse in terms of both performance and effort required. OpenMP was much simpler to use for this problem since I blocked the code in such a way that each thread could compute completely independent of the others. The only place threads had in common was matrix B, which is read only. My MPI solution was basically identical blocked code from Lab 1, but with added code for communication; communication which must be coherent and SPMD (added consistency trouble for programmer). This speaks to the expected performance of each implementation as well. OpenMP scales incredibly well, while MPI does not. The cost of adding a thread with OpenMP (within the systems available thread count) easily pays for itself since all threads can work independently within the same shared memory. MPI however incurs a massive communication cost for process 0 when adding more processes. This added cost indicates that MPI is not as well suited for this problem as OpenMP. Having private memory adds nothing for this type of problem.