Parallel Blocked Matrix Multiplication Report, now with 100% more MPI

David Akeley

1 Communication Strategy

I break up the input and output matrices into 128-by-128 tiles. Each worker process (`rank > 0`) is assigned a portion of the output matrix tiles. The master process (`rank = 0`) collects input matrix tiles and sends them with `MPI_Isend` to the worker processes. Input tiles from the right-hand operand matrix are transposed before being sent, to facilitate vectorization. The worker process only sends input tiles needed for a worker process to compute its assigned output tiles; furthermore, each tile is sent at most once to any single worker process – worker processes cache and reuse received tiles. A unique MPI tag is given to each tile to identify it.

The master process is not given a fixed assignment of any output tiles to compute; however, in its spare time, when no workers are ready to be assigned new output tiles, the master process tries to reduce the workload of the most behind worker process by computing some output tiles for it.

The one process case is treated specially: no MPI functions are used at all, and the master process just computes the output tiles in a lexicographical order.

2 Communication API

My implementation relies heavily on asynchronous communication – there's no pre-planned order of sending and receiving matrix tiles. The master process just sends and receives tiles on a dynamic, unpredictable schedule, whether or not the worker processes are ready to receive or produce tiles. This was probably a huge mistake, but it's too late to fix that now.

None of the blocking APIs would be suitable, due to my reliance on asynchronous communication. It doesn't seem like buffered vs. unbuffered had any performance impact, which I tested with this diff:

```
diff --git a/lab2/mpi.cpp b/lab2/mpi.cpp
index c48ae80..f3a7bcb 100644
--- a/lab2/mpi.cpp
+++ b/lab2/mpi.cpp
@@ -21,11 +21,6 @@

 namespace {

-// This should be using template parameters but I'm screwed for time
-// right now.
-float buffer_for_bsend[kI * kJ + kJ * kK * 2];
```

```
-bool did_buffer_attach = false;
-
 // Calculate the dot product of the lhs Length-vector with each of the
 // eight rhs Length-vectors, and return as a 8-vector of floats. Length
 // must be a positive multiple of 32. Interleaving the calculation of
@@ -483,7 +478,7 @@ private:
     {
         gemm_assert(!tile_was_sent(tag, process_rank));
         entry_array[tag].rank_bitfield |= uint64_t(1) << process_rank;
-        MPI_Ibsend(
+        MPI_Isend(
             &tile.data,
             sizeof tile.data / sizeof(float),
             MPI_FLOAT,
@@ -1088,10 +1083,6 @@ public:
     // to chip in and help compute worker processes' tiles.
     void master_loop()
     {
-        if (!did_buffer_attach) {
-            MPI_Buffer_attach(buffer_for_bsend, sizeof buffer_for_bsend);
-            did_buffer_attach = true;
-        }
         // Break up the output matrix into pieces for the workers.  At
         // this point, the workers have started on their 0th
         // assignment.
```

Since buffering didn't improve performance, I decided to go with the lower-risk unbuffered APIs.

3 Problem Size Performance

With 4 processes, I get 56.458 GFlops for problem size 1024^3, 82.7355 GFlops for problem size 2048^3, and 92.8025 GFlops for problem size 4096^3. Just for fun I also tested 8192^3, which gave me 96.9387 GFlops.

It seems that performance increases as matrix size goes up. My guess is that this happens because of the input matrix tile cache – worker processes can re-use inputs more with a larger matrix size, minimizing the proportion of time spent communicating input tiles.

```
ubuntu@ip-172-31-12-119:~/MPI-matrix/lab2$ mpiexec -np 4 ./gemm
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run parallel GEMM with MPI
Time: 1.48098 s
Perf: 92.8025 GFlops
```

```
ubuntu@ip-172-31-12-119:~/MPI-matrix/lab2$ make gemm
mpicxx -Wall -Wextra -std=c++11 -O3  -fno-loop-interchange -fno-loop-unroll-and-jam -
march=native -o gemm ../lab1/gemm.cpp ../lab1/gemm-baseline.a mpi.cpp main.cpp -lgomp
ubuntu@ip-172-31-12-119:~/MPI-matrix/lab2$ mpiexec -np 4 ./gemm
Problem size: 2048 x 2048 x 2048
Initialize matrices a and b

Run parallel GEMM with MPI
Time: 0.207648 s
Perf: 82.7355 GFlops
ubuntu@ip-172-31-12-119:~/MPI-matrix/lab2$ make gemm
mpicxx -Wall -Wextra -std=c++11 -O3  -fno-loop-interchange -fno-loop-unroll-and-jam -
march=native -o gemm ../lab1/gemm.cpp ../lab1/gemm-baseline.a mpi.cpp main.cpp -lgomp
ubuntu@ip-172-31-12-119:~/MPI-matrix/lab2$ mpiexec -np 4 ./gemm
Problem size: 1024 x 1024 x 1024
Initialize matrices a and b

Run parallel GEMM with MPI
Time: 0.0380368 s
Perf: 56.458 GFlops
ubuntu@ip-172-31-12-119:~/MPI-matrix/lab2$ make gemm
mpicxx -Wall -Wextra -std=c++11 -O3  -fno-loop-interchange -fno-loop-unroll-and-jam -
march=native -o gemm ../lab1/gemm.cpp ../lab1/gemm-baseline.a mpi.cpp main.cpp -lgomp
ubuntu@ip-172-31-12-119:~/MPI-matrix/lab2$ mpiexec -np 4 ./gemm
Problem size: 8192 x 8192 x 8192
Initialize matrices a and b

Run parallel GEMM with MPI
Time: 11.3423 s
Perf: 96.9387 GFlops
```

4 Scalability

My program actually scales decently up to 4 cores. It's not as scalable up to 8, but this may be because those 8 cores are virtual – there's only 4 physical cores on the AWS instance used. Further process number increases are disastrous to performance, probably because a greater proportion of time is spent communicating without any actual compute increase.

```
ubuntu@ip-172-31-12-119:~/MPI-matrix/lab2$ mpiexec -np 1 ./gemm
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run parallel GEMM with MPI
Time: 4.60502 s
Perf: 29.8455 GFlops
```

```
ubuntu@ip-172-31-12-119:~/MPI-matrix/lab2$ mpiexec -np 2 ./gemm
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run parallel GEMM with MPI
Time: 2.94421 s
Perf: 46.6811 GFlops
ubuntu@ip-172-31-12-119:~/MPI-matrix/lab2$ mpiexec -np 4 ./gemm
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run parallel GEMM with MPI
Time: 1.48017 s
Perf: 92.8537 GFlops
ubuntu@ip-172-31-12-119:~/MPI-matrix/lab2$ mpiexec -np 8 ./gemm
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run parallel GEMM with MPI
Time: 1.24326 s
Perf: 110.547 GFlops
ubuntu@ip-172-31-12-119:~/MPI-matrix/lab2$ mpiexec -np 16 ./gemm
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run parallel GEMM with MPI
Time: 5.43222 s
Perf: 25.3007 GFlops
ubuntu@ip-172-31-12-119:~/MPI-matrix/lab2$ mpiexec -np 32 ./gemm
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run parallel GEMM with MPI
Time: 25.3563 s
Perf: 5.4203 GFlops
```

5 OpenMP vs MPI

5.1 Programming Effort

The MPI assignment required much more programming effort than the OpenMP assignment. However, in large part I think that this is due to my poor design choices. I envisioned an "organic" system where the master process dynamically schedules output tiles for computation in whatever order it sees fit, and sends needed input tiles to worker processes "just in time".

This seemed to me at first like it would be more elegant and simpler to implement than static scheduling, which I thought would require more upfront planning by the programmer (me), and require unpleasant tight coupling and dependency assumptions between worker processes. I realized after writing about 700 lines of code (out of the total 1200) that this was not actually the case.

5.2 Performance RANT

Please read my sad story :'(

I took the liberty of benchmarking my lab 1 OpenMP gemm and my MPI gemm on my own gaming desktop, which uses an AMD Ryzen 2700X. I think a single core of that CPU is about the same as an Intel core used on the AWS instance (actually, I think the FPU vector width is half on AMD compared to Intel, so my gaming desktop is actually somewhat worse). Although the 2700X has 8 physical cores, I used 4 threads/processes, to match the 4 physical core AWS instances used.

I got 127 GFlops with OpenMP, 133 GFlops with my MPI implementation, and if I disabled the "callback hack" (described later), I got 139 GFlops. So my MPI performance actually exceeded my OpenMP performance[1] – but only on my personal computer! At first, when I ran my MPI lab on the AWS instance, I got only about 20 GFlops!

I tried all sorts of changes: jiggling with MPI API choices, adding extra buffering, rescheduling, all to no avail. The only thing I tried that actually worked was disabling matrix computation on the master process entirely, i.e. solely allocating output tiles to worker processes to compute. This however wasted 25% of my floating point units, which kept me up at night.

What I realized the next morning is that there's just something wrong with MPICH's so-called asynchronous APIs. It can't tolerate the long communication pauses while the master process was off doing matrix compute work, but for some reason Open MPI, which I use on my personal computer, had no problem with this. This inspired the "callback hack" – basically, while the master is busy computing an output matrix tile, it periodically calls a callback that re-communicates with every worker process. This allowed me to somewhat recover the FPU resources of the master process, although the overhead of such frequent communication is significant. The hack can be toggled with the `GEMM_NO_CALLBACK_HACK` macro, as seen in this code:

```
#ifdef GEMM_NO_CALLBACK_HACK
                compute_output_tile(&_tmp_sum, _sender, tile_i, tile_j,
                    dummy_callback);
#else
                compute_output_tile(&_tmp_sum, _sender, tile_i, tile_j, *this);
#endif
```

[1] This seems counterintuitive to me; I would have expected the implicit communication overhead of a shared memory space to be somewhat less than that of explicit communication. I think the real reason for this is that I only ever transpose a given matrix tile once in my MPI implementation and cache this transposed tile. In my OpenMP implementation I don't recycle this transpose work.

Even with this hack though, I still only got 80 GFlops or so with MPICH on AWS, compared to 107.93 GFlops with 4 OpenMP threads. If I extrapolate my own Ryzen 2700X and Open MPI performance, I estimate that I would have gotten

$$\frac{\texttt{Open MPI on 2700X}}{\texttt{OpenMP on 2700X}}\texttt{OpenMP on AWS} = \frac{139\texttt{GFlops}}{127\texttt{GFlops}}107.93\texttt{GFlops} = 118\texttt{GFlops}$$

If I could use Open MPI for my performance benchmark on AWS (and disabled the un-needed callback hack).

My estimate still puts me in B range, so I do regret the overengineering and code bloat of this homework submission, and wish I chose a more conventional algorithm. Still, I think the primary reason for my poor performance is because I got absolutely fucked over by MPICH. I'm actually really livid about this, which puts me in not the best mental state to tackle my computer architecture midterm. But I have to take that test anyway, so with this I conclude my report.