Project 3

Oi Lam Sou and Keith Luong

Phases

1. Program #1: Matrix multiplication using MPI
2. Program #2: Mandelbrot set

Program #1: Matrix multiplication using MPI

The performance of matrix multiplication this time is optimized by MPI. The idea is quite similar to theading, except that it is done so through message passing. This main characteristics of MPI allows the use of remote processors. As a result, the program becomes more scalable as it's no longer bounded by a local host. In `mmm_mpi.c`, we initialized MPI by `MPI_Init(&argc,&argv);`, then we obtain the size of the communicator created and the rank of a task by `MPI_Comm_size(MPI_COMM_WORLD,&numtasks);` and `MPI_Comm_rank(MPI_COMM_WORLD,&taskid);` respectively. Task with rank `0` will be the master while tasks with ranks larger than `0` will be the workers/slaves. The distribution of workload is static in this program. Each worker will be assigned the same amount of workload `rowpertask = n / (numtasks - 1)` at once. If the matrix order is not divisible by number of tasks, the remainder will be made up by adding one more row.

Master task

`MASTER` is responsible for allocating memory space for all matrices and initializing `A` and `B` so that it can send them to the workers. It starts by sending all parameters and input needed for computation to workers, including the matrix order `n`, `offset` for `A` and `C`, segment of rows `myrow`, the whole matrix `B` and the respective segment of matrix `A`. Messages from `MASTER` to workers are tagged with `1`.

```
...
MPI_Send(&B[0], n * n, MPI_DOUBLE, tid, 1, MPI_COMM_WORLD);
MPI_Send(&A[offset], myrow * n, MPI_DOUBLE, tid, 1, MPI_COMM_WORLD);
...
```

Then, `MASTER` will receive the results from workers including a segment of `C`, size of the segment (number of rows) `myrow` and `offset` in order to put the segment of `C` in the right place:

```
...
MPI_Recv(&offset, 1, MPI_INT, tid, 2, MPI_COMM_WORLD, &status);
MPI_Recv(&myrow, 1, MPI_INT, tid, 2, MPI_COMM_WORLD, &status);
MPI_Recv(&C[offset], myrow * n, MPI_DOUBLE, tid, 2, MPI_COMM_WORLD, &status);
...
```

Messages from workers to `MASTER` is tagged as `2`.

Worker tasks

A worker is responsible for the actual computation of matrix multiplication. First it receives the parameters needed for computation: `n` (matrix order), `offset` (for `A` and `C`) and `myrow` (number of rows). Using this information, worker is able to allocate appropriate memory space for the matrices. Then, it will receive the whole `B` and the portion of `A` in order to calculate its portion of `C`. Finally, it will send the calculated portion of `C` and its respective parameters to `MASTER` so that the portion of `C` will be put in the correct place.

MPI is terminated by `MPI_Finalize();`.

Timing measurement

`mpirun -n N ./mmm_mpi 500`

N	Running time #1	Running time #2	Running time #3	Mean runtime
2 (serial)	0.065661	0.066007	0.076338	0.069335
4	0.037143	0.036668	0.032147	0.035319
8	**0.032724**	**0.034662**	**0.032600**	**0.033329**
16	0.066835	0.050057	0.051531	0.056141
32	0.117624	0.109797	0.118193	0.115205
64	0.436475	0.405924	0.382945	0.408448

`mpirun -n N ./mmm_mpi 2000`

N	Running time #1	Running time #2	Running time #3	Mean runtime
2 (serial)	5.407144	5.438125	5.380642	5.408637
4	**2.920352**	**2.918215**	**2.921997**	**2.920188**
8	3.018931	3.011139	3.003037	3.011036
16	3.107592	3.128444	3.125412	3.120484
32	3.473519	3.550525	3.452923	3.492322

`mpirun --hostfile csif_hostfile -np N mmm_mpi 500`

N	Running time #1	Running time #2	Running time #3	Mean runtime
2	0.108223	0.104075	0.103919	0.105406
3	**0.036386**	**0.037831**	**0.037508**	**0.037242**
4	**0.039219**	**0.039254**	**0.039255**	**0.039242**
8	0.886669	0.884652	0.882320	0.884547
16	2.389745	2.401332	2.366564	2.385880
32	5.110934	5.080035	5.080933	5.090634

local computer (multi-cores) vs. multiple computers (multi-nodes)

Using `mpirun` with different configurations, we can run the program on a local system or multiple computers. Apparently, multiple computers allow higher scalability in terms of the number of nodes. However, for the same number of processes, local system outperforms MPI that's done over a network. It makes sense because in a system of multiple computers, there're more overheads passing messages between different computers. We observed from the above table that, in a local system with 8 processors, the multiplication of matrices finishes fastest when `MPI_COMM_SIZE` equals 8, which is the number of processors. It fits our expectation because all the processors are all working and the communication overhead is offset by acceleration in calculation brought by parallelism of MPI. However, the "golden" number is not the same for a network of computers. We've found size of `MPI_COMM_WORLD` = 3 or 4 performs best in a network of csif computers, which tells us that the overhead of communicating between computers is rather significant compared to the complexity of matrix multiplication. Since these processes are loosely coupled in a network of computers, bandwidth is lower and latency becomes higher. This explains why the `--hostfile` option returns much slower than a local system.

small matrices vs. large matrices

We've noticed that for matrices with large orders (e.g. 2000, 3000, 5000...etc.), 8 is no longer the best number of processes. The difference in performance becomes more significant as the order gets larger. However, the difference is very insignificant so we can just pretend `-n 8` performs as good as it does for small matrices.

Program #2: Mandelbrot Set

For this program, we utilize MPI to dynamically allocate tasks when generating an image of the Mandelbrot set. Because the image is represented in PGM as a large 2D array of values (ie the number of iterations of the equation used to determine set membership), we decided to parallelize and optimize the computation process by having each worker process in the MPI network calculate a row of values independently. The worker process' calculated row would then be returned to the master process to compile and present the entire matrix.

Master code

After declaring variables in the main function scope to hold the program arguments `x_center, y_center, zoom, and cutoff`, we initialize MPI and call `checkArgc()` in the master process to validate input and `getInput()` to assign the variables from user input. If an error in the format or range of the command line arguments is found, `checkArgc()` exits the program and reports an appropriate error message. Once the program parameters have been taken from command line input, we get the distance between points on the image using `dist` = 2^{-zoom} and together with `x_center` and `y_center` we find the starting pixel.

With the constant values for the image's `x_center, y_center, dist`, and cutoff determined, we nearly have enough information to begin creating tasks for other processes. In order to manage and delegate work to worker processes, the master process requires additional variables to track how many processes are active and working. To do this, we use a `working_tasks` variable local to the master process that is updated whenever task parameters are sent to or received from a process. By monitoring this working_tasks variable, we can ensure that we call `MPI_Recv` to retrieve results for every active process, and not expect any new results if `working_tasks == 0`.

The master process begins by initializing the values for `x_center`, y_center, `dist`, and cutoff in each worker process' scope. In this initialization loop, we also send next, which corresponds to the index of the next row that a worker process calculates. With these worker processes running, we receive calculated rows by waiting for any process to return their row index, get the ID of the process, and use that ID and row index to replace the appropriate row in the master's local copy of the matrix with the worker process' calculated row. We continue incrementing working_tasks and sending new values for next until all of the row indices of the matrix (0-1023) have been sent to a worker process. When this occurs, we send a null value for next with a special tag number (3) to signal the worker processes to stop.

Once `working_tasks == 0`, we get and print the calculation time, matrix checksum, and call `writeImage()` to draw the PGM image using the result matrix.

Worker Code

Each worker process begins by allocating a new int array `plane[]` to store the values calculated for its assigned matrix row. Next, the process' local values for input parameters `x_center, y_center, dist, cutoff` are updated once the values are received from the master using `MPI_Recv`(). After this, each process executes a while loop that exits if a `MPI_Recv()` call to update next receives a message with an `MPI_TAG` unequal to `1` or if the message generates an error. This should occur once the master finds that row_sent matches the number of rows in the matrix. Within the while loop, we call `iteratePoint()` on each index of the row to calculate the number of iterations that the Mandelbrot equation is run for the point's associated complex value. Once values for the row have been assigned, the row index and row values are sent back to the master process with a return `MPI_TAG` value of `2`.

Timing Measurement

```
mpirun -n N ./mandelbrot_mpi -1 -0.2 9 256
```

Number of tasks	Running time #1	Running time #2	Running time #3	Mean runtime
2 (serial)	0.847501	0.844279	0.844985	0.845588
4	0.307523	0.303908	0.304338	0.305256
8	0.253007	0.251352	0.251570	0.251976
16	0.260951	0.262446	0.246086	0.256494
32	0.245656	0.236120	0.240654	0.240810
64	0.295983	0.250135	0.294961	0.280360

```
mpirun --hostfile csif_hostfile -np N ./mandelbrot_mpi -7 5.2 50 256
```

Number of tasks	Running time #1	Running time #2	Running time #3	Mean runtime
2 (serial)	0.018725	0.019527	0.019564	0.019142
4	0.007042	0.012106	0.005639	0.007149
8	0.162833	0.166674	0.175068	0.005535

Number of tasks	Running time #1	Running time #2	Running time #3	Mean runtime
16	0.301167	0.329286	0.337363	0.011991
32	0.363554	0.443882	0.402314	0.016333
64	0.033230	0.583556	0.564833	0.038994

```
mpirun --hostfile csif_hostfile -np N ./mandelbrot_mpi -1 -0.2 9 256
```

N	Running time #1	Running time #2	Running time #3	Mean runtime
2	1.581024	1.757338	1.672732	1.670365
3	0.763472	0.793965	0.769273	0.775570
4	0.625642	0.628855	0.634396	0.629631
5	0.302146	0.284726	0.292505	0.293126
6	0.300831	0.290144	0.283503	0.291493
7	0.346066	0.304367	0.343276	0.331236
8	0.285252	0.294370	0.281040	0.276887
16	0.326629	0.321474	0.330356	0.326153

Based on the run times calculated for different task numbers and cutoff parameters, we can make a few observations. First, we see that for both tests, as number of available tasks increase, mean runtimes decrease until a certain point (n = 16). This behavior is what we would expect of the program, since more tasks being available would correspond to more processes calculating the result matrix at the same time. Run times for higher task numbers may be slightly more elongated due to increased complexity in overhead when managing all of the available processes. By testing the effects that each individual input parameter has on the final runtime, we saw no significant changes in run time when zoom or cutoff values were changed. We saw the greatest difference in run time when changing the `x_center` and `y_center` parameters to be closer or nearer to the center value of (0, 0). The nearer to the center of the image the image is created, the longer the compilation takes. This makes sense, as most values that have membership within the Mandelbrot set can be found near the center point of (0,0), resulting in longer calculation times since there would be more iterations of the Mandelbrot equation in `iteratePoint()`.

Comparing local runtimes to those utilizing different computers, we also clearly see the cost of overhead when MPI is used is significant.

How we tested the project

We ran the test scripts provided to see if our output matches the expected output.

Sources used

1. Project 1 and 2