HW2 - High Performance Computing

Andrew Szymczak, ajs855
March, 2015

GitHub Repository

- `https://github.com/Turing-Apparatus/HPC-HW2.git`

Organization

The `c` code and the report pdf are in the root directory. The tex code for the report is in the Tex/ folder and any images used are under Tex/Images/. Output from `ssort` is put into the Output/ folder.

Commands

`make`	compile `output*.c` and `ssort.c`
`make clean`	remove all executables, tex garbage, and ssort output.
`make runsolved (NP) (EXT)`	run `mpi_solved(EXT)` with (NP) processors. NP defaults to 4 and EXT defaults to 1 if ommitted. For example `runsolved EXT=2` will execute `mpirun -np 4 ./mpi_solved2`
`make runssort (NP) (N) (S)`	shortcut for `mpirun -np NP ./ssort N S`. NP defaults to 4. N and S are as defined in the HW.

1. Finding MPI Bugs

1. The `tag` parameter was wrong.
2. Datatype mismatch from sender to receiver (`MPI_INT` to `MPI_FLOAT`).
3. Missing `MPI_Init` at the beginning and `MPI_Finalize` at the end.
4. `MASTER` is missing the `reduce` operation.
5. The timings start off fast and then jump to slow. This is due to the asynchronous nature of message passing. When process 0 performs the send operation, it will either be blocked until the message is copied by the reciever or by the the system buffer (if the buffer exists and is not full). In the latter case, control will resume before the message is received and Wtime will be called nearly instantly. The simplest fix is to use synchronous `MPI_Ssend`. Also, the datatype parameter should probably be `MPI_CHAR` rather than control will resume before the message is received and Wtime will be called nearly instantly. This explaines the jump in timings. The simplest fix is to use synchronous `MPI_Ssend`. Also, the datatype parameter should probably be `MPI_CHAR` rather than `MPI_BYTE`.
6. Process 2 should not call `MPI_Waitall` because it does not perform any non-blocking operations. Also, `OFFSET` should be initialized to 0 for all processes. Fixing these, the code will run, but there is still a danger. We are doing a non-blocking send/recieve in a loop, overwriting the message buffer in each iteration. Generally it is safer to call an `MPI_Wait` in each loop iteration or alternatively promote the buffer to an iterable. Since we don't care about / don't process the message in any way, it doesn't matter for this code.
7. `count` should be a constant 1.

You can run the code in two ways. Ommiting $(NP) defaults it to 4.

```
make runsolved NP=4 EXT=2
mpirun -np 4 ./mpi_solved2
```

2. Sample Sort

I seed the time by `srand ((rank+1)*time(NULL))`. This isn't truly random since it oversamples, but it's good enough. To choose each sample set, I sort and then pick s equispaced points. I use `MPI_Gather` to send all the sample sets to *root*, which sorts and then broadcasts the P equispaced splits. Each process then counts the number of elements in each split (`scounts`) as well as positions (`sdispls`). I use an `MPI_Alltoall` to send `scounts[i]` from processors j into `rcounts[j]` in processor i over all i, j. Then each processor sets up their bucket by computing positions `rdispls` directly from `rcounts`. Then they pass their data via an `MPI_Alltoallv` and then sort and write to a file. Command line arguments are N the number of elements per processor and s the size of each sample set. You can run the code in two ways. Ommitting $(NP) defaults it to 4.

```
make runssort NP=64 N=100000 S=1000
mpirun -np 64 ./ssort 100000 1000
```

Distributed memory parallel Jacobi smoother. Use MPI to write a parallel version of the Jacobi smoother from Homework 0

- The first and last element of $\boldsymbol{u}^{(i)}$ are the shared indices. First we pass the previous values $u_{-2}^{(i)} \to u_0^{(i+1)}$ for all $i < p-1$. Then we perform the Jacobi iteration. Then we pass the new values $u_1^{(i)} \to u_{-1}^{(i-1)}$ for all $i > 0$.
- I use an `MPI_Allreduce` to calculate the residual on every processor. Since this is an expensive process, I only do it once every 100 iterations.
- Since my machine only utilizes four processors, my first scaling test was for $p = 1, 2, 3, 4$. Let I denote the number of iterations, and t the total time in seconds.

N	I	p	$\lvert Au - f\rvert$	t
2520	5000000	1	0.932	54.62
2520	5000000	2	0.932	28.26
2520	5000000	3	0.932	22.08
2520	5000000	4	0.932	20.12

  Note how the residual is independent of p, as expected. Each processor handles 2 messages and N/p of the Jacobi work in each iteration. Assuming full connectivity (so that all messages are done in parallel), we can expect $\mathcal{O}(N/p)$ time (as marked by the `X`'s). The slowdown is due to the suboptimal parallelisation and overhead of message passing.
- My second scaling test only establishes the high overhead of message passing. Increasing $p > 4$ on my local machine effectively sequentializes crosstalk. Since there are $2p-2$ messages passed in every iteration, we can expect the computation time to grow $\sim p$.

N	I	p	$\lvert Au - f\rvert$	t
1200	100000	20	22.24	10.40
1200	100000	40	22.24	21.83
1200	100000	60	22.24	33.17
1200	100000	80	22.24	48.06
1200	100000	100	22.24	63.77

- Gauss-Seidel would be more dificult to program because of the implicit dependence in each iteration. We could try to do something analagous to the first bullet point: First we pass the old values $u_1^{(i)} \to u_{-1}^{(i-1)}$ for all $i < p-1$. Then we perform the Jacobi iteration. Then we pass the new values $u_{-2}^{(i)} \to u_0^{(i+1)}$ for all $i < p-1$. The problem is that you can't do this in parallel. Before $u^{(i)}$ can preform Jacobi, it must wait to recieve the new value $u_{-2}^{(i-1)}$, which isn't sent until all lower rank processes are complete. The process is essentially sequential, and even worse off due to message passing.