Tyler Petrochko

CPSC 524 Parallel Programming

Assignment 3

3/3/17

Building/Running

My code for Tasks 2, 3, 4, and 5 are contained respectively in the following files:

blocking.c

nonblocking.c

loadbalancing.c

general.c

To build the code, run "make" from the directory including the Makefile, making sure that the appropriate modules have been loaded, which can be done with the following bash command:

```
$ module load Langs/Intel/15; module load Langs/Intel/15 MPI/OpenMPI/1.8.6-intel15
```

To generate the appropriate output files, run the command "bash times.sh" on a qsub instance with the appropriate number of processors. My qsub command is

```
$ qsub -I -l procs=8,tpn=8,mem=34gb,walltime=15:00 -q fas_devel
```

This creates the following files:

serial.txt

general.c

blocking_1.txt

blocking_2.txt

blocking_4.txt

blocking_8.txt

nonblocking_1.txt

nonblocking_2.txt

nonblocking_4.txt

nonblocking_8.txt

loadbalancing_1.txt

loadbalancing_2.txt

loadbalancing_4.txt

loadbalancing_8.txt

To get the remaining files

blocking2nodes.txt

blocking4nodes.txt

nonblocking2nodes.txt

nonblocking4nodes.txt

I re-ran the command on a qsub instance with 2 or 4 nodes, i.e.

$ qsub -I -l procs=8,tpn=16,mem=34gb,walltime=15:00 -q fas_devel

$ qsub -I -l procs=8,tpn=32,mem=34gb,walltime=15:00 -q fas_devel

and changed the filenames accordingly.

In each file, <*description*>_<*number*>.txt includes the results for appropriate *description* task with p = *number*. E.g., loadbalancing_8.txt includes the results for Task 4 with p = 8. The file "serial.txt" includes the results for Task 1.

Note that in each output file, the tables providing overall timing are printed interleaved with the per-processor breakdown of communication and computation timing. I used Microsoft Excel to extract these into separate tables for the purposes of this report.

Including each *.txt* output would be excessive for the purposes of this report, so I've included them in the *.tar.gz* but have not pasted them here. They cover the tables listed below as well as individual processor load-balancing information.

To ensure correctness, I included a flag VERIFY at the beginning of each .c program that checks against the serial computation using matmul.c. For testing (and as-is) I turned this feature off, but it can easily be renabled by changing the line

#define VERIFY 0

…to

#define VERIFY 1

Task 1: Serial Program

The following is the output I received when running the serial program

N	TIME (secs)
1000	0.1817
2000	2.2711
4000	19.3616

8000 153.6509

Task 2: Blocking MPI Parallel Program

Each column "COMP" column is the total time spent computing across all nodes in the cluster, and "COMM" is total communication time across all nodes (so it makes sense that "COMP" + "COMM" do not add up to the "TIME" column).

For p = 1

N	TIME	COMP	COMM
1000	0.1909	0.183333	0.002127
2000	2.3174	2.293484	0.007151
4000	19.5828	19.489805	0.027857
8000	155.0219	154.650664	0.110832

For p = 2

N	TIME	COMP	COMM
1000	0.1549	0.183473	0.141582
2000	1.6992	1.99189	1.384328
4000	16.5028	20.735287	12.251069
8000	132.0661	166.795712	97.052262

For p = 4

N	TIME	COMP	COMM
1000	0.1085	0.184248	0.265125
2000	0.8551	1.451708	1.86611
4000	11.3606	19.358875	21.830939
8000	94.2921	166.309294	174.313335

For p = 8

N	TIME	COMP	COMM
1000	0.0644	0.186714	0.437167
2000	0.4774	1.441115	2.683034
4000	6.8054	20.742941	30.773539
8000	60.9901	195.535713	260.737269

Using 2 nodes, p = 8

N	TIME	COMP	COMM
4000	7.505	20.891302	33.411493
8000	65.3442	198.572865	279.178624

Using 4 nodes, p = 8

N	TIME	COMP	COMM
4000	7.5068	19.636056	28.842727
8000	65.3655	187.712397	237.532742

Regarding the raw performance, scalability, and load balance, there are a few ways to improve the program. Firstly, it seems like the overwhelming majority of the time is spent communicating rather than processing. One reason for this is that all the MPI calls are *blocking*, preventing each processor from doing anything while sending/receiving data. To improve this, we could rewrite the program (as in Task 3) using asynchronous calls and maintaining that while processing, *each processor is sending and receiving data in the background*. Doing so would offer substantial speed benefits.

Another source of improvement could come from the load balance. If we break down the time spend communicating and running computations by the processor rank, we get the following table

RANK	COMP	COMM
0	4.78107	41.974535
1	13.525792	36.502015
2	20.028059	36.510479
3	25.896456	32.200436
4	28.774148	32.520766
5	32.675181	29.761891
6	34.14155	25.308759

7	35.713457	25.958388

As we can see, the first processor (Rank 0) does considerably less computation than other nodes, but overwhelmingly more communication. This happens because the first N / p rows of the matrix are considerable smaller (because we don't store 0 elements), so the processor of Rank 0 has to do very little actual computation, but spends most of its time waiting for other processors to assign it work. We can remedy this (as in Task 4) by varying the number of rows each processor is assigned, so that the 0^{th} processor gets the most number of rows, while the p^{th} gets the least.

Another observation is that using more nodes decreased performance. This makes sense, because inter-process communication requires going across the network, rather than using a shared hardware bus. This is exacerbated by the round-robin distribution of processors to nodes; using the ring-passing techniques therefore ensures that every time a processor passes on a column-block, it must do so to a different node.

Implementing these changes would reduce the amount of time spent waiting on other processors to finish and even out the work distribution, resulting in higher concurrency and thus more raw performance and better scalability.

Task 3: Non-Blocking MPI Parallel Program

For p = 1

N	TIME	COMP	COMM
4000	19.5293	19.436168	0.02815
8000	154.6211	154.233682	0.109828

For p = 2

N	TIME	COMP	COMM
4000	17.3557	20.262736	11.235173
8000	138.8256	162.378293	88.001559

For p = 4

N	TIME	COMP	COMM
4000	10.0242	20.846549	17.152336
8000	80.7618	178.203444	125.316666

For p = 8

N	TIME	COMP	COMM
4000	5.9736	24.787742	22.304928

8000	51.6828	231.203073	168.42623

Using 2 nodes, p = 8

N	TIME	COMP	COMM
4000	6.7988	24.880381	23.12032
8000	56.3732	231.134896	167.509019

Using 4 nodes p = 8

N	TIME	COMP	COMM
4000	6.8418	25.152306	23.214959
8000	56.2919	231.468686	166.930126

As expected using non-blocking calls increased raw performance (decreased overall computation time) and drastically reduced the amount of time spent communicating between nodes. Unlike Task 2, in this case the time spent communicating is always less than computing, suggesting that there are far fewer "blockages" that prevent parallel computation. However, as revealed by a similar table to the one in Task 2, there is still a load balancing issue between nodes:

RANK	COMP	COMM
0	6.214370	39.538066
1	17.338651	32.192972
2	23.905756	24.751780
3	32.172593	17.894723
4	32.430901	17.406580
5	39.917734	11.715399
6	37.548697	13.523830
7	41.674371	11.402880

The 0^{th} processor still does dramatically less computation than later processors, and thus spends the most time waiting for other processors to assign it work. We fix this in Task 4 by varying the number of rows assigned to each processor.

Task 4: Load Balance

For p = 1

N	TIME	COMP	COMM
4000	19.5124	19.446154	0.0276

8000	154.547	154.282646	0.110234

For p = 2

N	TIME	COMP	COMM
4000	16.3069	21.010044	9.859771
8000	129.8897	168.023142	76.349311

For p = 4

N	TIME	COMP	COMM
4000	8.1398	20.872706	8.64696
8000	64.9332	179.491922	53.598811

For p = 8

N	TIME	COMP	COMM
4000	5.1044	26.617269	13.533231
8000	46.6615	245.969221	103.907208

As expected, using a variable-width row distribution greatly increased performance. This shows in the reduction in overall computation time as well as communication time. The results are most evident in the work distribution between individual processors:

RANK	COMP	COMM
0	40.111553	6.263535
1	33.841555	10.974373
2	34.504438	8.072795
3	30.126725	13.186198
4	30.952891	9.451508
5	27.337253	15.546039
6	26.734458	16.552978
7	22.360348	23.859782

Now that these load-balancing changes have been made, the computation and communication time fluctuates very little from processor to processor. This allows the processors to “share” the workload easier and achieve better scalability, since better parallelism helps achieve a near p-way speedup.

Despite these optimizations, the maximum speedup achieved was only 154.547 / 46.6615 ≈ 3.3x, despite using 8 processors. At this level of speedup, the efficiency of the program is only 3.3 / 8 ≈ 41%.

Task 5: Generalization

To generalize my Task-4 program, I implemented a simple "padding scheme." Before any workers are assigned rows or columns, the master determines if the matrix columns can easily be split up, i.e. if N modulo p is zero. If not, the master pads the matrix by adding p – (N modulo p) extra rows and columns.

Then, the master and workers proceed as usual. When the master collects the processed matrix from the workers, it uses a resize_matrix function to bring the matrix back to the original size. This adds some extra computational overhead, but the time spent resizing is largely outweighed by the other computational costs (and time spent communicating). Running the generalized version on N = 7633 and p = 7 took 46.3 seconds, with the following per-processor performance breakdown:

RANK	COMP	COMM
0	38.386218	4.872115
1	29.181310	11.649721
2	33.192631	5.245922
3	25.575175	14.886961
4	29.608588	7.575787
5	22.028394	19.740690
6	24.068174	19.229825

Overall, the raw performance and fairness between processors is similar to that in Task 4, suggesting there is very little performance lost as the result of these changes.

Unlike with other tasks, I enabled verification in general.c, so the program first computes a matrix using the above scheme, then computes a second matrix using the provided serial code, then compares them. If any matrix elements do not match, the program reports this. As mentioned before, I included this functionality in Tasks 2-4 for development purposes, but disabled it in the submitted source code.