High Performance Computing
Homework 1

Sunli Tang

March 9, 2015

1 MPI Ring Communication

First, let me mention that I run all the tests on my Mac, which has 2 physical cores and 4 logical cores. Hence, to see the scaling or to test the time elapsed per communication, it only make sense if I restrict the number of processors no more than 2.

I run int_ring.c using 2 processors and $N = 10000$. The result shows that the time elapsed per communication is about 5.40×10^{-6} seconds. Then, I set the message to be an array of size $2e6$ Bytes. Note that, I only update the first element of the message every time, so that the time elapsed will not be dominated by updating a big array. In this test, I get the time elapsed per communication, for message size $2e6$ Bytes is about 5.36×10^{-5} seconds, which is slower by a factor of 10. It is not clear to me how to estimate the bandwidth using these information, but clearly, the larger messages passing through processors, the longer time it takes in each communication.

2 Distributed Memory Parallel Jacobi Smoother

Again, except the test to see independence of result with number of processors, the other tests are restricted on no more than 2 processors, for the reason I stated in the first section.

First, I fix the size of the matrix and the number of iterations, and vary the number of processors. For simplicity, it is sufficient to set the size of matrix and number of iterations to be small. Of course the solution will not be accurate, but as long as those "inaccurate" solutions are the same, we can conclude the result is independent to number of processors we choose. In the following test, I fix the size of matrix $N = 8$, run 5 iterations per simulation, and vary the number of processors from 1 to 8. The result is attached in table 1. The script file *idp_p.sh* will run this test for you.

Second, we want to see the strong scalability, that is, if we fix the amount of work (i.e. fix size of matrix and number of iterations here), and estimate the cpu time, hopefully we will see that cputime $\sim \frac{1}{\text{number of processors}}$. Here, we fix

======Test for 1 Processor(s)====
pt = 1, x = 0.111111, u(x) = 0.017288; p.w.error = 6.667052e-02
pt = 2, x = 0.222222, u(x) = 0.026088; p.w.error = 1.125077e-01
pt = 3, x = 0.333333, u(x) = 0.030102; p.w.error = 1.412133e-01
pt = 4, x = 0.444444, u(x) = 0.031385; p.w.error = 1.548414e-01
pt = 5, x = 0.555556, u(x) = 0.031385; p.w.error = 1.548414e-01
pt = 6, x = 0.666667, u(x) = 0.030102; p.w.error = 1.412133e-01
pt = 7, x = 0.777778, u(x) = 0.026088; p.w.error = 1.125077e-01
pt = 8, x = 0.888889, u(x) = 0.017288; p.w.error = 6.667052e-02
======Test for 2 Processor(s)====
pt = 1, x = 0.111111, u(x) = 0.017288; p.w.error = 6.667052e-02
pt = 2, x = 0.222222, u(x) = 0.026088; p.w.error = 1.125077e-01
pt = 3, x = 0.333333, u(x) = 0.030102; p.w.error = 1.412133e-01
pt = 4, x = 0.444444, u(x) = 0.031385; p.w.error = 1.548414e-01
pt = 5, x = 0.555556, u(x) = 0.031385; p.w.error = 1.548414e-01
pt = 6, x = 0.666667, u(x) = 0.030102; p.w.error = 1.412133e-01
pt = 7, x = 0.777778, u(x) = 0.026088; p.w.error = 1.125077e-01
pt = 8, x = 0.888889, u(x) = 0.017288; p.w.error = 6.667052e-02
======Test for 4 Processor(s)====
pt = 1, x = 0.111111, u(x) = 0.017288; p.w.error = 6.667052e-02
pt = 2, x = 0.222222, u(x) = 0.026088; p.w.error = 1.125077e-01
pt = 5, x = 0.555556, u(x) = 0.031385; p.w.error = 1.548414e-01
pt = 6, x = 0.666667, u(x) = 0.030102; p.w.error = 1.412133e-01
pt = 7, x = 0.777778, u(x) = 0.026088; p.w.error = 1.125077e-01
pt = 8, x = 0.888889, u(x) = 0.017288; p.w.error = 6.667052e-02
pt = 3, x = 0.333333, u(x) = 0.030102; p.w.error = 1.412133e-01
pt = 4, x = 0.444444, u(x) = 0.031385; p.w.error = 1.548414e-01
======Test for 8 Processor(s)====
pt = 1, x = 0.111111, u(x) = 0.017288; p.w.error = 6.667052e-02
pt = 2, x = 0.222222, u(x) = 0.026088; p.w.error = 1.125077e-01
pt = 7, x = 0.777778, u(x) = 0.026088; p.w.error = 1.125077e-01
pt = 8, x = 0.888889, u(x) = 0.017288; p.w.error = 6.667052e-02
pt = 3, x = 0.333333, u(x) = 0.030102; p.w.error = 1.412133e-01
pt = 4, x = 0.444444, u(x) = 0.031385; p.w.error = 1.548414e-01
pt = 5, x = 0.555556, u(x) = 0.031385; p.w.error = 1.548414e-01
pt = 6, x = 0.666667, u(x) = 0.030102; p.w.error = 1.412133e-01

Table 1: Independence of Number of Processors

time = 0.009966 s; np = 1; N = 65536; time * np = 0.009966
time = 0.005416 s; np = 2; N = 65536; time * np = 0.010832
time = 0.005863 s; np = 4; N = 65536; time * np = 0.023452
time = 0.003418 s; np = 8; N = 65536; time * np = 0.027344
time = 0.002557 s; np = 16; N = 65536; time * np = 0.040912

Table 2: Strong Scalability

$N = 65536$, and run 100 simulations, the resule is in table 2. The script file *strong_scale.sh* will do this test for you. For np = 1 and 2, we see that the product of cputime and np is about a constant. This demonstrates the strong scalability. However, for np > 2, it seems the cputime does not scale any more. The reason for this is, as stated before, my machine only have 2 physical cores.

Third, we mention here that the parallelization of Jacobi iteration is easy, because the updating of $u_n(x)$ only depend on $u_{n-1}(x)$. This means, we only require different processors exchange information before each step. However, in Gauss-Seidel iteration, to update $u_n(x_i)$, we need $u_n(x_j)$ for $j < i$, and $u_{n-1}(x_j)$ for $j > i$. This implies, we have to require processors passing message before we calculating each **point**! Hence, parallel Gauss-Seidel in this way is not possible. However, there is a way to do parallelization, using some variations of Red-Black Gauss-Seidel. The advantage of Red-Black Gauss-Seidel is, to update the current solution, we only need the solution of last step.