CSE 6220: Assignment 2

Lukas Ott (lott7)

March 20, 2024

Professor Srinivas Aluru has explicitly stated that programming assignments 2 and 3 cannot be posted publicly, since this would be a violation of academic integrity. However, since the results obtain in programming assignment 2 are of great interest, I wanted to share the report which is part of the submission. People interested in the details of this project can reach out to me at lott7@gatech.edu.

I have implemented a parallel algorithm to transpose a square matrix of integers of size $n \times n$. The deliverable contains a driver file with the algorithm implementation, `main.cpp`, a Makefile to compile the program, and a README file where you will find instructions on how to run.

The transposition of the matrix is done with the `Alltoall` communication primitive. In addition to MPI's implementation, I have created two `Alltoall` routines, one with arbitrary permutations and one with hypercubic permutations. I will now compare their runtime as a function of the problem size and the number of processors.

1 The two plots (8 pts).

Figure 1 provides the runtimes for 8 and 16 processors, with matrix size going from $n = p$ to $n = 32768 = 2^{15}$.

2 Theoretical analysis for space and time complexities for both all-to-all implementation and the matrix transpose (4 pts).

There are n^2 elements block-distributed among p processors. It is assumed that $\frac{n}{p}$ is a division without remainder; furthermore, $p \leq n$. Each processor receives $\frac{n^2}{p}$ elements of the matrix. Each of these elements needs to be reordered to ensure they are sent to the correct processor in `Alltoall`. This is accounted for in our `main.cpp` file. This operation takes $O(\frac{n^2}{p})$. Furthermore, after `Alltoall`, we have to reorder the elements so that they are in the correct matrix order, which is also of order $O(\frac{n^2}{p})$. The computation time for the transposition is thus $O(\frac{n^2}{p})$.

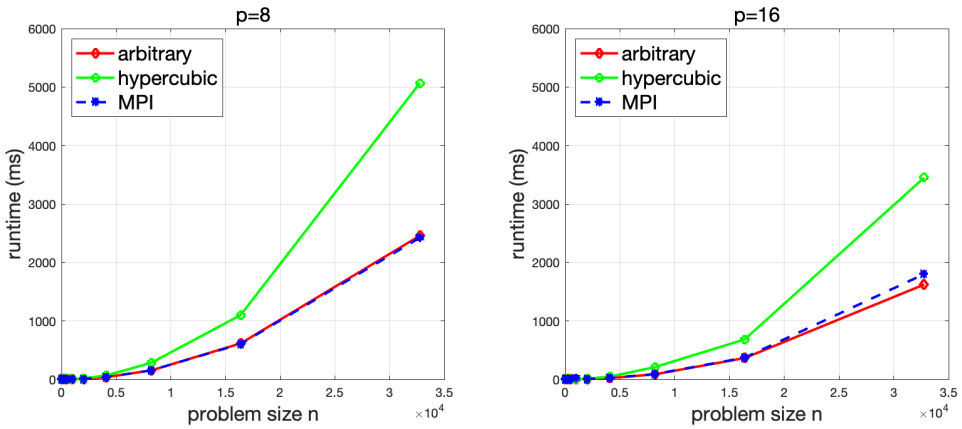

Figure 1: Runtime as a function of matrix size for $p = 8$ processors (left) and $p = 16$ processors (right).

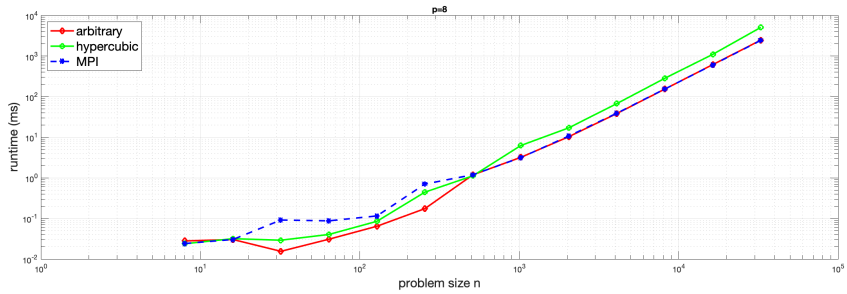

Figure 2: Runtime as a function of matrix size for $p = 8$ processors (log-log plot).

Each processor has to send $\frac{1}{p}$ of its data to other processors. Thus the message size is $m = \frac{n^2}{p^2}$. Theoretically, the communication times should be

- arbitrary permutation: $O(\tau p + \mu \cdot m \cdot p) = O(\tau p + \mu \frac{n^2}{p})$
- hypercubic permutation: $O(\tau \log p + \mu \cdot m \cdot p \log p) = O(\tau \log p + \mu \cdot \frac{n^2}{p} \log p)$

3 Empirical analysis, observations, and conclusion (8 pts).

Looking at Fig. 1, I observe that `mpi_Alltoall` and `arbitrary_Alltoall` have the same behavior for large n. I choose to analyze runtimes for large matrices because these are of higher fidelity, which makes sense since parallel programs are usually designed to solve large problems when utilizing multiple processors. If we look at the difference in time between $n = 2^{14}$ and $n = 2^{15}$ in table 1. We observe that for arbitrary permutations and MPI, the time precisely quadruples as the n doubles, confirming a $O(n^2)$ complexity. Similarly, doubling p is observed to reduce time, although

not by a factor of two because of unavoidable communication patterns. The runtime also more than quadruples with $p = 16$ when I double n, probably due to the same phenomenon.

n		2^{14}	2^{15}
p=8	mpi	606.1701775	2423.873425
	arbitrary	614.8262024	2457.192898
	hypercubic	1100.23427	5067.827225
p=16	mpi	371.4795113	1800.681114
	arbitrary	364.823103	1622.255802
	hypercubic	683.8421822	3450.319052

Table 1: Numerical results of runtimes in ms, for large matrix transposition.

Hypercubic permutations usually work well for lower n, as can be seen in Fig. 2 on a logarithmic scale for more clarity. However, for larger problem sizes, hypercubic permutations exhibited larger runtimes than arbitrary permutations, which is to be expected from the theoretical analysis. We can observe the difference in time between $p = 8$ and $p = 16$. At $p = 8$ and $n = 2^{15}$, we have $t = 5067$ ms, so the theoretical runtime for $p = 16$ should be

$$\frac{5067}{2}\frac{\log 16}{\log 8} = \frac{5067}{2}\frac{4}{3} = 3378, \tag{1}$$

and the observed runtime was actually 3450, which is quite close.

In conclusion, for large n, the $O(\mu \cdot \frac{n^2}{p} \log p)$ from hypercubic behaves worse than the $O(\mu \frac{n^2}{p})$ from arbitrary permutations. The runtimes shown here are the result of a single run, and using more statistics could smooth out some of the discrepancies.