HW 2

Carlos Vazquez Gomez

February 16, 2019

1 Implementation Using MPI Natives

Here, MPI treats MPI_COMM_WORLD as a single, asymmetric tree. P_0 simply does MPI_Scatter on the array, then each process performs its integration, and finally, each process does MPI_Gather into P_0. P_0 eventually receives an array of doubles of size np, and it adds all their elements to get the final integral value.

2 Custom Implementation

My custom implementation assigns each process to a tree of size a power of 2. Then, P_0 (which has the entire workload stored in memory), sends the appropriate chunks to each tree root (virtual address = 0). Figure 1 shows an example with 13 processes. This method is expected to be fast because the number of processes is usually sub-exponential, so the burden that P_0 carries in distributing and collecting each tree's data, on the order $\log np$, is insignificant.

First, P_0 sends the data corresponding to ranks $8 \rightarrow 11$ (T_1) to process 8, whose virtual rank is 0. P_0 also sends the data corresponding to rank 12 (T_2) to process 12 (also virtual rank 0). Every process that is not P_0 is blocked on a MPI_Probe call. P_8 and P_{12} are return from MPI_Probe when P_0 sends them data, and they MPI_Get_count to allocate space for it. Then, propagate in a loop to processes in their tree. The processes targeted by their propagation return from their MPI_Probe, and also propagate in a loop. The stopping condition for the propagation is that the array of data being propagated contains only that processor's share of computation. Then, the entire tree's processes begin integrating at approximately the same time (horizontal dashed lines in Figure 1).

After the integration, the tree begins gathering into its corresponding virtual rank 0 process (P_0, P_8, and P_{12}). During this time, P_0 is continuously performing MPI_Recv (from senders in its own tree). Even as it does this, P_{12} and P_8 asynchronously complete their gathers, and send to P_0 their tree sum. However, there is no need for barriers because they send to P_0 via a unique tag, which P_0 does not listen on until after it gathers all of tree 0. P_0 then waits for

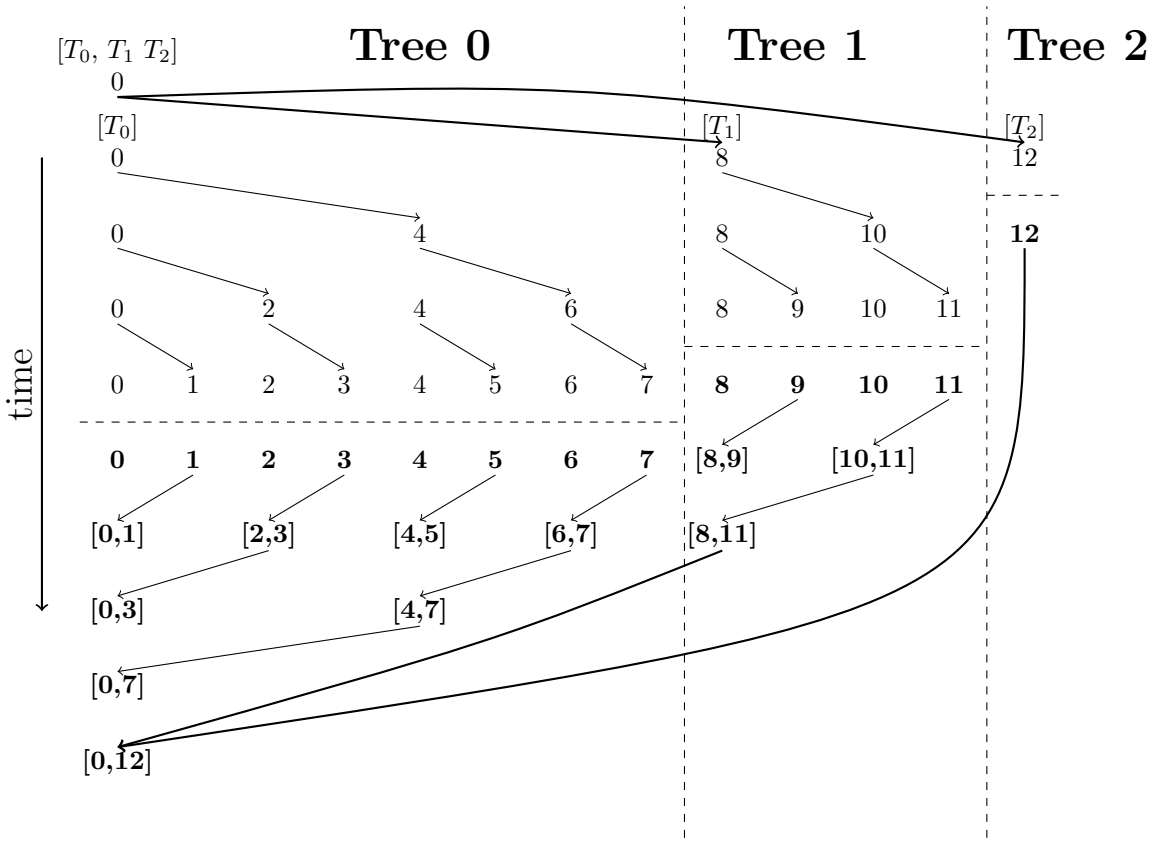

Figure 1: Communication involved in a 13 process computation (custom implementation)

two MPI_Recv on this unique tag, accumulating their sums along the way, and therefore has now the final integral value.

3 Performance Comparison

There are three implementations which we will compare. "native" is uses the native MPI collectives described in section 2, "main" is the implementation described in section 1, and "slow" is my original implementation, in which every process propagated data down to only two children (left and right). Then, it performed its integration and stood idle until its children responded with their sums, at which point the process would send *that* sum to its parent. Instead of waiting idle, the processes could instead be helping to propagate the data to not only 2 children, but as many as possible until all processes (in its tree) have its chunk of data. This is the reason "main" is faster than "slow". Even

# Procs	Billions of points	Times (sec) native	main	slow
1	0.1	0.53	0.55	0.55
2	0.2	0.80	1.13	1.40
3	0.3	1.16	1.79	2.18
4	0.4	1.54	2.35	3.45
5	0.5	1.81	2.92	4.27
6	0.6	2.21	3.51	4.81
7	0.7	2.45	4.12	5.91
8	0.8	2.89	4.72	6.92
9	0.9	3.14	5.22	8.74
10	1.0	3.46	5.77	9.87
11	1.1	3.78	6.39	11.23
12	1.2	4.11	6.89	11.94
13	1.3	4.50	7.91	15.72

Table 1: Weak scaling analysis

though there are only two processes, I suspect that the huge amount of points (100 million per process in my tests) makes copying to MPI buffers quite slow, and therefore a communication protocol that takes more iterations to scatter data is significantly slower.

Table 1 shows the times for each implementation. I use weak scaling analysis by maintaining the points-per-process constant as I increase the processes of each test. Therefore, we can analyze the efficiency of parallelization. For example, communication overhead is evident in the fact that a single "native" process can integrate 100 million points in 0.53 seconds, but when communications with 12 other processes are involved, it needs 4.5 seconds to integrate 100 million points. Scaling efficiency is defined (for weak scaling) as $\frac{t_1}{t_n}$, where t_1 is the time taken for a single process to complete a workload when no other processes exist, and t_n when n processes exist.

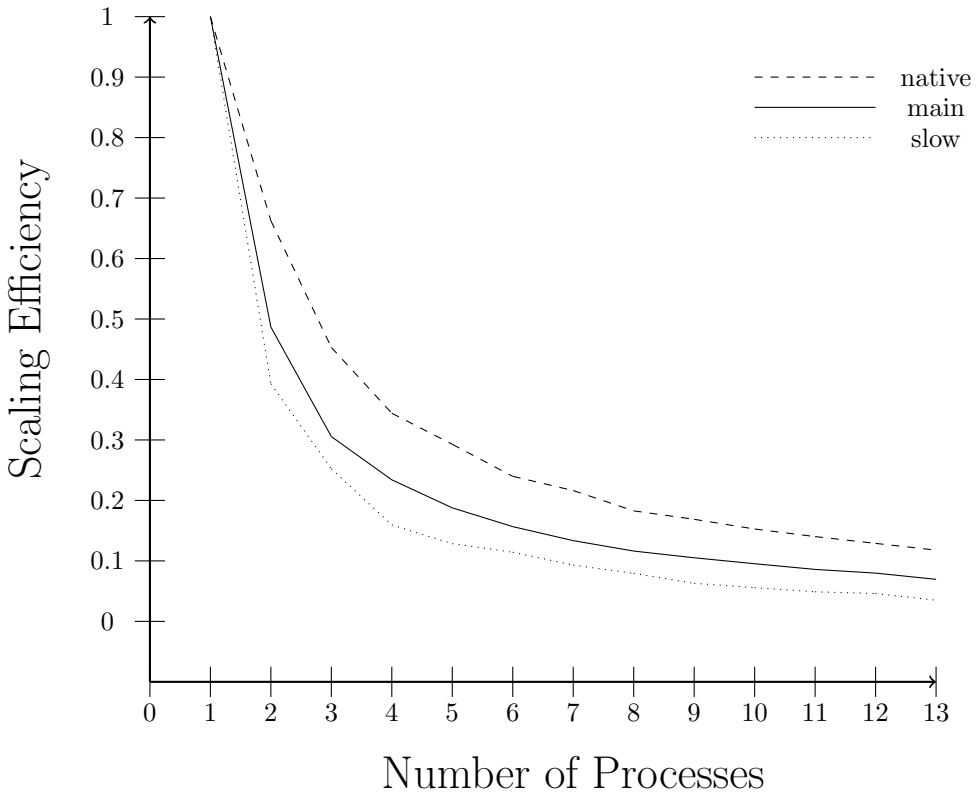

Figure 2: Communication overhead decreases the efficiency of each processes, even though each process has the same workload