An Introduction to Parallel Programming Solutions, Chapter 1

Jinyoung Choi and Peter Pacheco

July 26, 2011

1. The "+1" is for the '\0(null) character terminating the string. So if we use `strlen(greeting)` instead of `strlen(greeting)+1` the terminating null character won't be transmitted. As with any erroneous program, the exact behavior depends on the implementation. With some implementations the program crashes with a seg fault. With others, the program is apparently correct.

   If we use `MAX_STRING` instead of `strlen(greeting)+1`, the output of the program is identical. However, in this case, we're sending the entire array `greeting` instead of just the text of the greeting (and the terminating null character).

2. If n isn't evenly divisible by `comm_sz`, we won't use the correct number of trapezoids. For example, if $n = 15$ and `comm_sz = 4`, the current version of the program will use $15/4 = 3$ trapezoids on each process, which will mean it will only use a total of $3 \times 4 = 12$ trapezoids. In order to use 15 trapezoids, we need to assign some of the processes 4 trapezoids instead of 3. More precisely $15 \bmod 4 = 3$ processes should get 4 trapezoids, and one should get 3. So we could modify the code so that the first 3 processes get 4 trapezoids, and the last process gets 3.

   One way to do this is to compute the integer quotient and the remainder of $n/\text{comm\_sz}$. All the processes will get $n/\text{comm\_sz}$ trapezoids, while the first $n \bmod \text{comm\_sz}$ will get an extra trapezoid:

```
int quotient = n / comm_sz;
int remainder = n % comm_sz;
if (my_rank < remainder) {
   local_n = quotient + 1;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
} else {
```

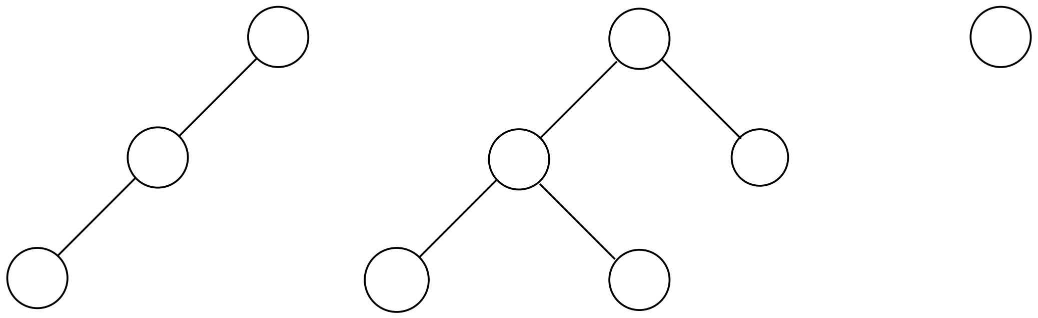

Figure 1: Binary trees

```
   local_n = quotient;
   local_a = a + my_rank*local_n*h + remainder*h;
   local_b = local_a + local_n*h;
}
```

3. The local variables are

   ```
   my_rank, local_n, local_a, local_b, local_int, source.
   ```

   The global variables are

   ```
   comm_sz, n, a, b, h, total_int.
   ```

4. We can send the messages to process 0 for printing. See `ex3.4_mpi_output_inorder.c`

5. We use induction on the number n of leaves. So suppose T is a complete binary tree with $n = 1$ leaf. If T is *any* binary tree with 1 leaf, then T consists of a single path from the root to the leaf. But if T is a complete binary tree, every nonleaf has two children. So if there's more than one node in the tree, there will be more than one leaf. See Figure 1. So the leaf is the root, which has depth 0 and depth $= \log_2(1)$.

   So suppose that if S is a complete binary tree with m leaves, where $1 \le m < n$, then the depth of the leaves is $\log_2(m)$. Now suppose T is a complete binary tree with $n > 1$ leaves. Consider the parents of the leaves of T. If d is the depth of the leaves, then each parent has depth $d - 1$. So all of the parents have the same depth. Furthermore, every ancestor of of a parent has two children, since it has two children in T. So the tree S that we obtain by removing all the leaves from T is a complete binary tree, and the leaves of S are the parents of the leaves of T. Since each parent is the parent of two leaves in T, S has $n/2$ leaves, and, by the induction hypothesis, the depth of its leaves is $\log_2(n/2)$. But then, the depth of the leaves of T is

$$1 + \log\left(\frac{n}{2}\right) = 1 + \log(n) - \log(2) = 1 + \log(n) - 1 = \log(n).$$

So by the principle of mathematical induction, in a complete tree with n leaves, each leaf has depth $\log(n)$.

6. (a) We can use the distribution we outlined in 3.1. This would assign components as follows.

$$\begin{aligned} \text{Process 0} &: x_0, x_1, x_2, x_3 \\ \text{Process 1} &: x_4, x_5, x_6, x_7 \\ \text{Process 2} &: x_8, x_9, x_{10} \\ \text{Process 3} &: x_{11}, x_{12}, x_{13} \end{aligned}$$

(b) With a cyclic distribution, we simply assign one component to each successive process — returning to process 0 after assigning to process `comm_sz` — until all the components have been assigned:

$$\begin{aligned} \text{Process 0} &: x_0, x_4, x_8, x_{12} \\ \text{Process 1} &: x_1, x_5, x_9, x_{13} \\ \text{Process 2} &: x_2, x_6, x_{10} \\ \text{Process 3} &: x_3, x_7, x_{11} \end{aligned}$$

(c) With a block-cyclic distribution, we can proceed as we did with the cyclic distribution, except that instead of assigning one component to each successive process, we assign two — the block size:

$$\begin{aligned} \text{Process 0} &: x_0, x_1, x_8, x_9 \\ \text{Process 1} &: x_2, x_3, x_{10}, x_{11} \\ \text{Process 2} &: x_4, x_5, x_{12}, x_{13} \\ \text{Process 3} &: x_6, x_7 \end{aligned}$$

Note that the block and block-cyclic distributions are not uniquely determined. For example, in the block-cyclic distribution we might split up the final block(s) among

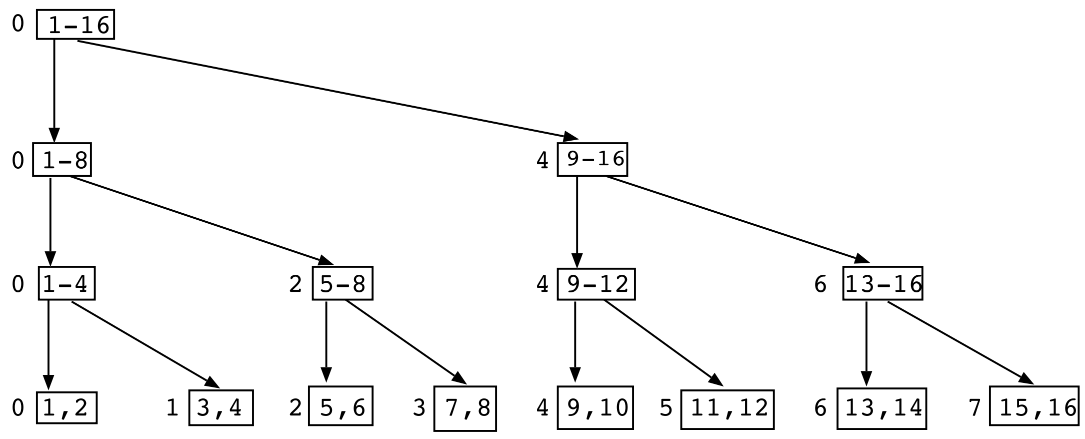

Figure 2: Scatter

the processes. In our example, we might have

Process 0	:	x_0, x_1, x_8, x_9
Process 1	:	x_2, x_3, x_{10}, x_{11}
Process 2	:	x_4, x_5, x_{12}
Process 3	:	x_6, x_7, x_{13}

7. See `ex3.7_mpi_coll_one_proc.c`. When we ran the program `MPI_Bcast` returned with no change to the input buffer. The other collectives simply copied the contents of the input buffer to the output buffer.

8. (a) Figure 2 shows the stages in scattering the integers $1, 2, \ldots, 16$.
   (b) Figure 3 shows the stages in gathering the integers $1, 2, \ldots, 16$ if they're initially distributed using a block distribution.

9. See the source file `ex3.9_mpi_vect_mult.c`.

10. This is OK because both formal arguments corresponding to `local_n` are input argument. The rule prohibiting aliasing only applies if one of the arguments is an output or an input/output argument.

11. (a)
```
prefix_sums[0] = vect[0];
for (i = 0; i < n; i++)
   prefix_sums[i] = prefix_sums[i-1] + vect[i];
```

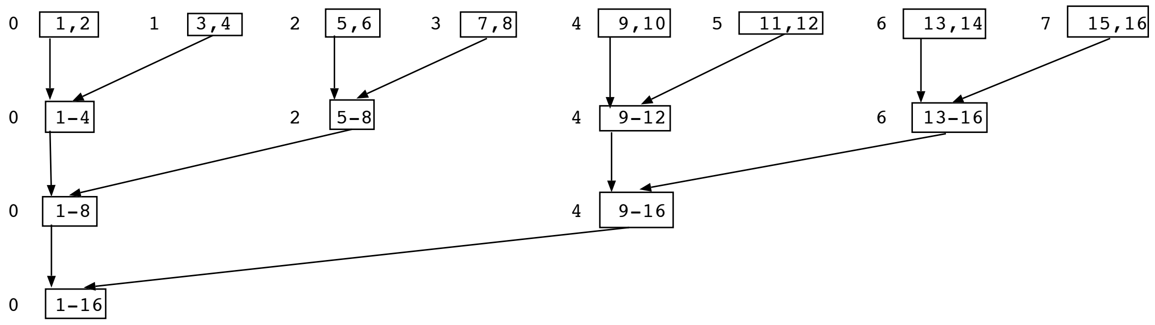

Figure 3: Gather

Also see the source file `ex3.11a_serial_prefix_sums.c`.

(b) The following algorithm should work if n, the order of the vector, is evenly divisible by `comm_sz`.

```
/* First compute prefix sums of my local vector */
loc_prefix_sums[0] = loc_vect[0];
for (loc_i = 1; loc_i < loc_n; loc_i++)
   loc_prefix_sums[loc_i] = loc_prefix_sums[loc_i-1] + loc_vect[loc_i];

if (my_rank != 0) {
   /* If I'm not 0 receive sum of preceding components */
   MPI_Recv(&sum_of_preceding, 1, MPI_DOUBLE, my_rank-1, 0, comm,
         MPI_STATUS_IGNORE);

   /* Add in sum of preceding components to my prefix sums */
   for (loc_i = 0; loc_i < n; loc_i++)
      loc_prefix_sums[loc_i] += sum_of_preceding;
}

/* Now send my last element to the next process */
if (my_rank != comm_sz - 1)
   MPI_Send(&loc_prefix_sums[loc_n-1], 1, MPI_DOUBLE, my_rank+1, 0, comm);
```

See the source file `ex3.11b_mpi_prefix_sums.c`.

(c) The following algorithm should also work if n, the order of the vector, is evenly divisible by `comm_sz`

```
/* First compute prefix sums of local data */
loc_prefix_sums[0] = loc_vect[0];
for (loc_i = 1; loc_i < loc_n; loc_i++)
   loc_prefix_sums[loc_i] = loc_prefix_sums[loc_i-1] + loc_vect[loc_i];

/* Now use butterfly structured communication */
sum = loc_prefix_sums[loc_n-1];
mask = 1;
while (mask < comm_sz) {
   partner = my_rank ^ mask;
   MPI_Sendrecv(&sum, 1, MPI_DOUBLE, partner, 0,
         &tmp, 1, MPI_DOUBLE, partner, 0,
         comm, MPI_STATUS_IGNORE);
   sum += tmp;
   if (my_rank > partner)
      for (loc_i = 0; loc_i < loc_n; loc_i++)
         loc_prefix_sums[loc_i] += tmp;
   mask <<= 1;
}
```

See the source file `ex3.11c_mpi_prefix_sums.c`.

(d) The main thing to beware of is that `MPI_SUM` doesn't carry out prefix sums on a process' local subvector. So simply using `MPI_Scan` with `MPI_SUM` will only work with one element per process. One solution is to use `MPI_Scan` on the sums of the processes' local elements:

```
Perform a prefix sum  of my elements;
my_sum = the sum of my elements;
Perform MPI_Scan of the processes' my_sum's getting pred_sum's;
Add pred_sum - my_sum to each of my elements;
```

This latter approach is implemented in the source file `ex3.11d_mpi_scan.c`.

12. (a) See the source file `ex3.12a_mpi_ringpass_allreduce.c`. The butterfly-structured implementation is much faster than the ring-pass. On one of our systems, with 16 processes, the butterfly is more than 10 times faster than the ring-pass.

(b) See the source file `ex3.12b_mpi_ringpass_prefix.c`.

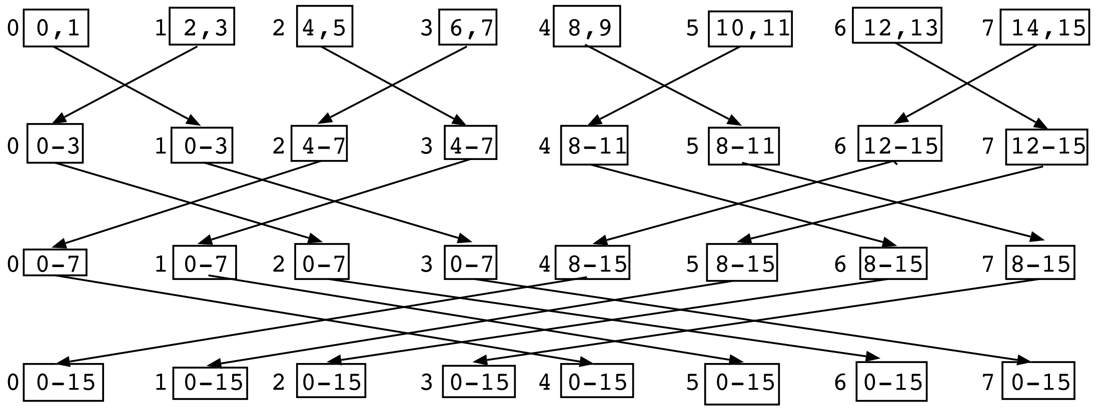

Figure 4: Allgather

13. See the source file `ex3.13_mpi_vect_sum_dot.c`.

14. See the source file `ex3.14_array.c`.

15. The two storage schemes are identical. They both store the elements of an array in the following order in main memory: elements in the first row followed by elements in the second row followed by elements in the third row, etc.

16. See Figure 4.

17. See the source file `ex3.17_mpi_type_contig.c`.

18. See the source file `ex3.18_mpi_type_vect.c`.

19. See the source file `ex3.19_mpi_type_indexed.c`.

20. See `ex3.20_mpi_trap_pack.c`.

21. On the system we used to generate the data in the book, the distributions of the run-times assume *many* different forms. There are combinations of `comm_sz` and matrix order for which the distribution of run-times is unimodal, distributions that are nearly uniform, and distributions that are bimodal. Among the unimodal distributions, there are distributions that are symmetric, distributions that are skewed left, and distributions that are skewed right.

22. It's important to choose the function being integrated and the number of trapezoids so that the elapsed time is a good deal larger than the resolution of the timer. If,

for example, the timer resolution is 1 microsecond, and the actual elapsed time of the trapezoidal rule program is less than 1 microsecond, the time reported by the program could be zero. But even if the actual elapsed time is (say) 2 microseconds, the time reported by the program could be off by as much as 50%.

The resolution of `MPI_Wtime` is returned by `MPI_Wtick`. On the system we used, the resolution is one microsecond.

The following table shows minimum elapsed times on one of our systems. There were ten runs for each problem size and each number of processes. "K" denotes thousands. Times are in milliseconds.

	Number of Trapezoids				
`comm_sz`	50K	100K	150K	200K	250K
1	1.90	3.81	5.70	7.57	9.50
2	1.07	2.09	3.12	4.14	5.18
4	0.58	1.12	1.67	2.21	2.75
8	0.33	0.62	0.92	1.21	1.50
16	0.61	0.75	0.79	0.99	1.11

On this system, the range of elapsed times is larger, relative to the minimum elapsed time, when `comm_sz` is larger and the problem size is smaller.

The following table shows speedups of the trapezoidal rule.

	Number of Trapezoids				
`comm_sz`	50K	100K	150K	200K	250K
1	1.00	1.00	1.00	1.00	1.00
2	1.80	1.82	1.83	1.83	1.84
4	3.31	3.38	3.41	3.43	3.46
8	5.89	6.14	6.21	6.24	6.33
16	3.16	5.08	7.18	7.67	8.57

The following table shows efficiencies of the trapezoidal rule.

	Number of Trapezoids				
`comm_sz`	50K	100K	150K	200K	250K
1	1.00	1.00	1.00	1.00	1.00
2	0.90	0.91	0.91	0.91	0.92
4	0.83	0.85	0.85	0.86	0.86
8	0.74	0.77	0.78	0.78	0.79
16	0.20	0.32	0.45	0.48	0.54

Based on these data, the trapezoidal rule isn't scalable. For example, when we increase from 2 processes to 4 processes, there is no problem size for which the program has an efficiency of 0.9 or better.

23. (a) For the data we collected in the preceding problem, the least-squares estimates for a and b are 3.85×10^{-8} and 1.07×10^{-4}, respectively. With these values the formula in the text predicts the following run-times (in milliseconds).

	Number of Trapezoids				
`comm_sz`	50K	100K	150K	200K	250K
1	1.92	3.85	5.77	7.69	9.62
2	1.07	2.03	2.99	3.95	4.91
4	0.70	1.18	1.66	2.14	2.62
8	0.56	0.80	1.04	1.28	1.52
16	0.55	0.67	0.79	0.91	1.03

(b) The predicted run-times are somewhat accurate. The maximum error in the predicted run-times is 0.27 milliseconds, and the maximum relative error is 0.73. That is, the maximum error in the predicted times is 73% of the actual time. On the other hand, the mean error is 0.092 ms, and the mean relative error is 0.085.

24. Although a zero-length message will contain no data, it will contain "envelope" information such as the tag and communicator. So there shouldn't be any problem with timing a zero-length message. On one of our systems the average time for such a message is about 3.4 microseconds.

25. (a) If the vectors don't fit into cache in the single process run but they do fit into cache in the multiprocess run, it could happen that the multiprocess obtained better than linear speedup, since the average time to add two components might be less: the loads and stores in the multiprocess run might not have to directly access main memory.

(b) The resource limitation in the single process run was the shortage of available cache.

This is the subject of some debate. Here's an example that might be considered superlinear speedup, but doesn't overcome a resource limitation. Suppose we have a serial program that searches a binary tree using depth-first search. If the node we are searching for is near the root, but in the right subtree of the root, the serial program will search the entire left subtree before it finds the desired node. However, if we parallelize the program so that the subtrees of the root are divided among the processes, then, when we run the parallel program with two processes, process 1 will get the right subtree, and it could find the desired node in much less than half the time it took the serial program.

26. (a) See `ex3.26ab_odd_even.c`.
    (b) See `ex3.26ab_odd_even.c`.
    (c) See `ex3.26c_odd_even.c`. When we ran the program with five hundred random lists each containing 25,000 integers, none were faster with the checks.

27. The following tables show the run-times, speedups and efficiencies of odd-even transposition sort on one of our systems. "M" is millions.

	Run-Times (in seconds)				
	Number of Elements				
`comm_sz`	1M	2M	4M	8M	16M
1	4.10E-02	8.73E-02	2.22E+00	4.65E+00	9.69E+00
2	2.04E-02	4.32E-02	1.10E+00	2.31E+00	4.81E+00
4	1.10E-02	2.24E-02	5.65E-01	1.18E+00	2.46E+00
8	6.73E-03	1.25E-02	2.98E-01	6.22E-01	1.29E+00
16	5.19E-03	8.45E-03	1.70E-01	3.53E-01	7.31E-01

	Speedups				
	Number of Elements				
`comm_sz`	1M	2M	4M	8M	16M
1	1.00	1.00	1.00	1.00	1.00
2	2.01	2.02	2.02	2.02	2.02
4	3.73	3.90	3.93	3.94	3.93
8	6.09	6.98	7.46	7.48	7.50
16	7.89	10.34	13.11	13.19	13.26

	Efficiencies				
	Number of Elements				
`comm_sz`	1M	2M	4M	8M	16M
1	1.00	1.00	1.00	1.00	1.00
2	1.00	1.01	1.01	1.01	1.01
4	0.93	0.97	0.98	0.98	0.98
8	0.76	0.87	0.93	0.93	0.94
16	0.49	0.65	0.82	0.82	0.83

On the basis of the data that were collected, the program is not scalable. For example, with 2 processes, the efficiencies are 1 (or better), while when run with 4 processes the program's maximum efficiency is 0.98.

28. See `ex3.28_mpi_odd_even.c`. The following tables show the run-times, speedups and efficiencies of this version of the sort.

	Run-Times (in seconds)				
	Number of Elements				
`comm_sz`	1M	2M	4M	8M	16M
1	4.11E-02	8.76E-02	2.22E+00	4.65E+00	9.69E+00
2	2.04E-02	4.32E-02	1.10E+00	2.30E+00	4.80E+00
4	1.10E-02	2.23E-02	5.52E-01	1.16E+00	2.41E+00
8	6.66E-03	1.25E-02	2.87E-01	6.00E-01	1.25E+00
16	5.27E-03	8.45E-03	1.64E-01	3.40E-01	7.06E-01

	Speedups				
	Number of Elements				
`comm_sz`	1M	2M	4M	8M	16M
1	1.00	1.00	1.00	1.00	1.00
2	2.01	2.03	2.02	2.02	2.02
4	3.75	3.93	4.03	4.02	4.02
8	6.17	6.98	7.74	7.75	7.76
16	7.79	10.37	13.59	13.69	13.72

	Efficiencies				
	Number of Elements				
`comm_sz`	1M	2M	4M	8M	16M
1	1.00	1.00	1.00	1.00	1.00
2	1.01	1.01	1.01	1.01	1.01
4	0.94	0.98	1.01	1.01	1.01
8	0.77	0.87	0.97	0.97	0.97
16	0.49	0.65	0.85	0.86	0.86

In many cases the run-times are similar to the original version. However, as the problem size increases and the number of processes increases, the run-times of this version become significantly better. This isn't surprising since, as the problem size increases the copies become increasingly expensive, and as the number of processes increases, the number of copies increases.

This version still isn't scalable: there's a marked deterioration in efficiencies when 16 processes are used. However, the efficiencies are somewhat better, and it's conceivable that with larger problems, we might be able to find a rate of increase in the problem size that would make the program scalable.