PLSPS - Assignment Report

Mehmet Berk Cetin
m.cetin@student.vu.nl

Testing Setup

We used one node and different amounts of cores within that node. This way, communication between cores could be faster than communication between nodes.
The nodes we ran our experiments in DAS5 have CPUs with the following specifications:

- Model: Intel(R) Xeon(R) CPU E5-2630 v3
- 8 cores
- 2.40 GHz
- Cache size: 20480 KB

Sequential Computational Job

We first fill the array either ascending or random. Then we loop through the array and pass the element as an argument to the test() or test_imbalanced() function and check the return value. When the total number of trues is 100, we terminate the job and return the time taken to complete the job. We run this program at one node and one core.

Fixed Computational Job (Test)

The method we use to solve the fixed computational job is similar to Successive Over Relaxation with early termination.
We solved the fixed computational program using collective communications between processors. The reason is that each processor does a fixed amount of computation, hence the time to finish the computation and return the result with a 1 or 0 is almost the same for all the processors.
We initially scattered the array to all processors, including the root process. After receiving a piece of the array, each processor starts computing, looping 25M times, and returns a 1 or 0 based on the value given to the test function. After the value is returned, we use All reduce to synchronize the number of trues to each process so all of them can terminate as soon as the number of trues becomes 100. We ran each setup, work type, array filling, number of processors, 5 or 6 times and got the average of the times measured.

In Figure 2, ascending array filling with fixed computation gets super linear speedup for 2 and 4 cores. The reason might be that the communication/computation overhead is really low. Each processor does O(N), N being 25M * nr_procs, computation and O(2) communication, where the total communication is amount of communication is O(M*M) because the root process does additional communication. O(2) communication is required because each processor, except the

root proc which does additional communication, communicates with the root process and replaces it's local number of trues with the global number of trues. Therefore, the communication overhead is very low and that results in superlinear speedup in the performance. Nonetheless, as the number of processors grows, the communication overhead grows exponentially while the computation is fixed, which leads to decreased speedup and efficiency when compared to 2 and 4 processors.
Another reason, perhaps the biggest for superlinear speedup, can be the part of the scattered array that contains high values, above 190, which lead to the sin (values between sin(210) and sin(330) are below -0.5 which returns true for the test function) function having a value less than -0.5. Moreover, in rand fixed computation, this isn't the case. Array values are distributed randomly in the range of 0 and 499. If an array has mostly low values, either getting to 100 trues is going to take very long, or the processors are never going to reach 100 trues and finish all the computation without terminating, which can lead to reduced speedup and efficiency. In Figure 1 and 2, If we compare the speedup and efficiency of random and ascending array fillings for fixed computation, we can see that ascending array filling has higher values for 2, 4, and 8 cores.

Imbalanced Computational Job (Test Imbalanced)

We solved the imbalanced computational program using peer to peer communication. The reason is that each processor does a different amount of computation, thus the time to finish the computation and return the result with a 1 or 0 can be significantly different for each processor. Therefore, if we use collective communication to keep track of the total number of trues, some processors would be idle for some time and reduce overall speedup.

We use an algorithm similar to the Traveling Salesman Problem, which uses dynamic work distribution. In our work, we initially loop through the array and send each processor, excluding the root processor which only does work distribution in test imbalanced, the value of the array. Sending work element by element to each processor is imperative for the ascending array because if we scatter the array, the processor that gets the array part that has high values would do much more computation than the other processor, hence taking more time to finish leading to load imbalance. We use n-1 worker processors and 1 root (master) processor. This way, the master processor can distribute work immediately, instead of first completing its own computation which may take long and then sending work, after a process finishes and requests for work. Each time a processor requests for work and sends the returning value from test_imbalanced(), the root processor receives the returning value, updates the total number of trues and sends work to the processor.
Figure 2 depicts that ascending and random arrays with imbalanced computations get a speedup close to linear. The reason can be that the communication/computation overhead is low. Each process does a dynamic amount of computation and a fixed amount of communication, which is O(1). The total amount of computation in ascending filling is 501*(500 / 2) * 100000 and the total amount of computation is N*M, which is 500 x number of processors. Since the communication overhead isn't exponential, the increase of the speedup of imbalanced work doesn't decrease as much as fixed work when the number of processors increases from 2 to 8.

Appendices

Number of processors	Fill type	Work type	Time	Efficiency	Speedup
1	asc	fixed	442.24	1.0	1.0
2	asc	fixed	173.95	1.27	2.54
4	asc	fixed	92.84	1.19	4.76
8	asc	fixed	65.32	0.84	6.76
1	asc	imbalanced	353.46	1.0	1.0
2	asc	imbalanced	181.99	0.97	1.94
4	asc	imbalanced	94.46	0.93	3.74
8	asc	imbalanced	51.97	0.85	6.80
1	rand	fixed	328.78	1.0	1.0
2	rand	fixed	195.38	0.84	1.68
4	rand	fixed	113.72	0.72	2.89
8	rand	fixed	60.41	0.68	5.44
1	rand	imbalanced	349.86	1.0	1.0
2	rand	imbalanced	169.51	1.03	2.06
4	rand	imbalanced	90.66	0.96	3.85
8	rand	imbalanced	47.30	0.92	7.39

Table 1: Depicting the time, efficiency, and speedup of the programs separated by array filling, work type, and number of processors

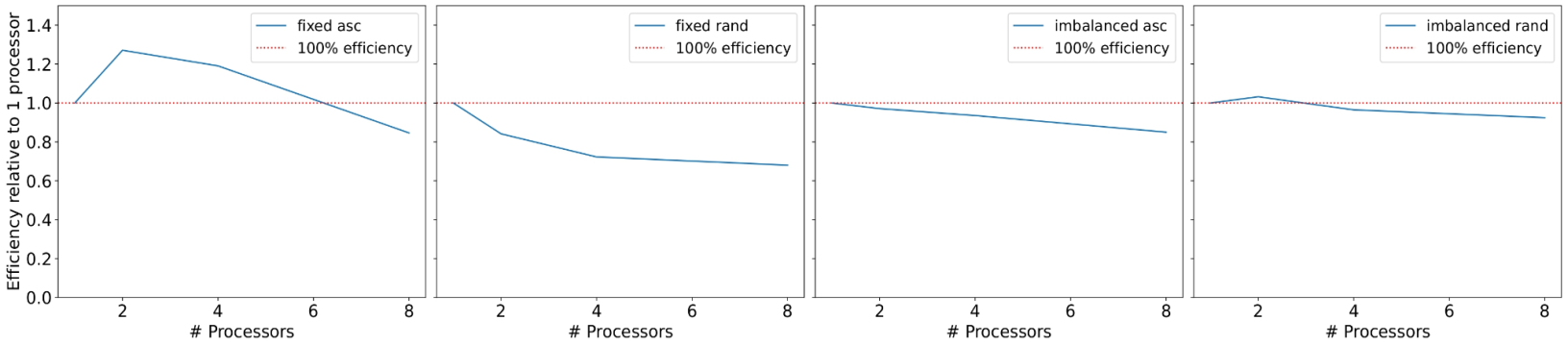

Figure 1: Efficiency curves separated by work type and array filling.

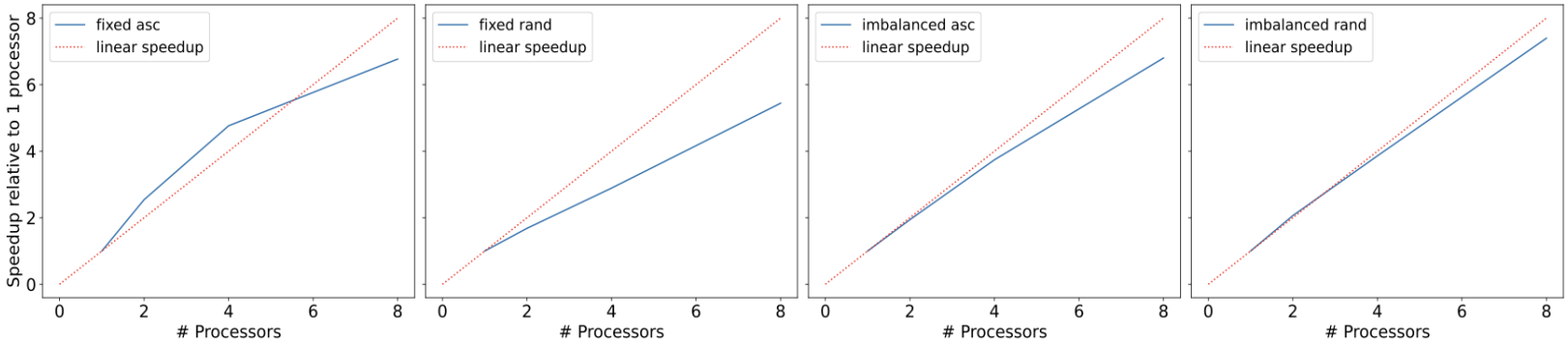

Figure 2: Speedup curves separated by work type and array filling.