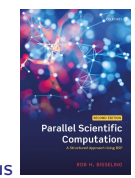
Experimental Results for Samplesort on Cartesius

Section 1.10 of Parallel Scientific Computation, 2nd edition

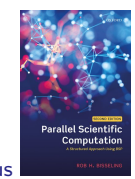
Rob H. Bisseling

Utrecht University



Experimental setup

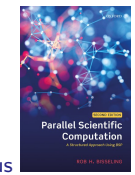
- Hardware: we use a Broadwell node with $p = 32$ cores of the supercomputer Cartesius at SURF in Amsterdam.
- BSP library: we view the node as a 32-core shared-memory machine and run MulticoreBSP for C.
- Software: we run the program `bspsort` from BSPedupack version 2.0, which implements a parallel regular samplesort.
- We also run a sequential program for proper comparison.
- Test problem: we sort n random numbers from the interval [0,1].



Time (in s) of parallel regular samplesort

p		Length n				
		10^4	10^5	10^6	10^7	10^8
1	(seq)	0.0011	0.0124	0.146	1.714	19.51
1	(par)	0.0012	0.0136	0.156	1.836	20.83
2		0.0009	0.0075	0.087	0.948	10.69
4		0.0008	0.0046	0.046	0.501	5.56
8		0.0009	0.0032	0.027	0.271	2.88
16		0.0015	0.0040	0.019	0.166	1.59
32		0.0032	0.0047	0.022	0.129	0.99

- The sequential sort is the system quicksort in C.
- The sequential time grows as $\mathcal{O}(n \log n)$.
- For large n, good parallel speedups are obtained.

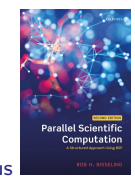


Speedup

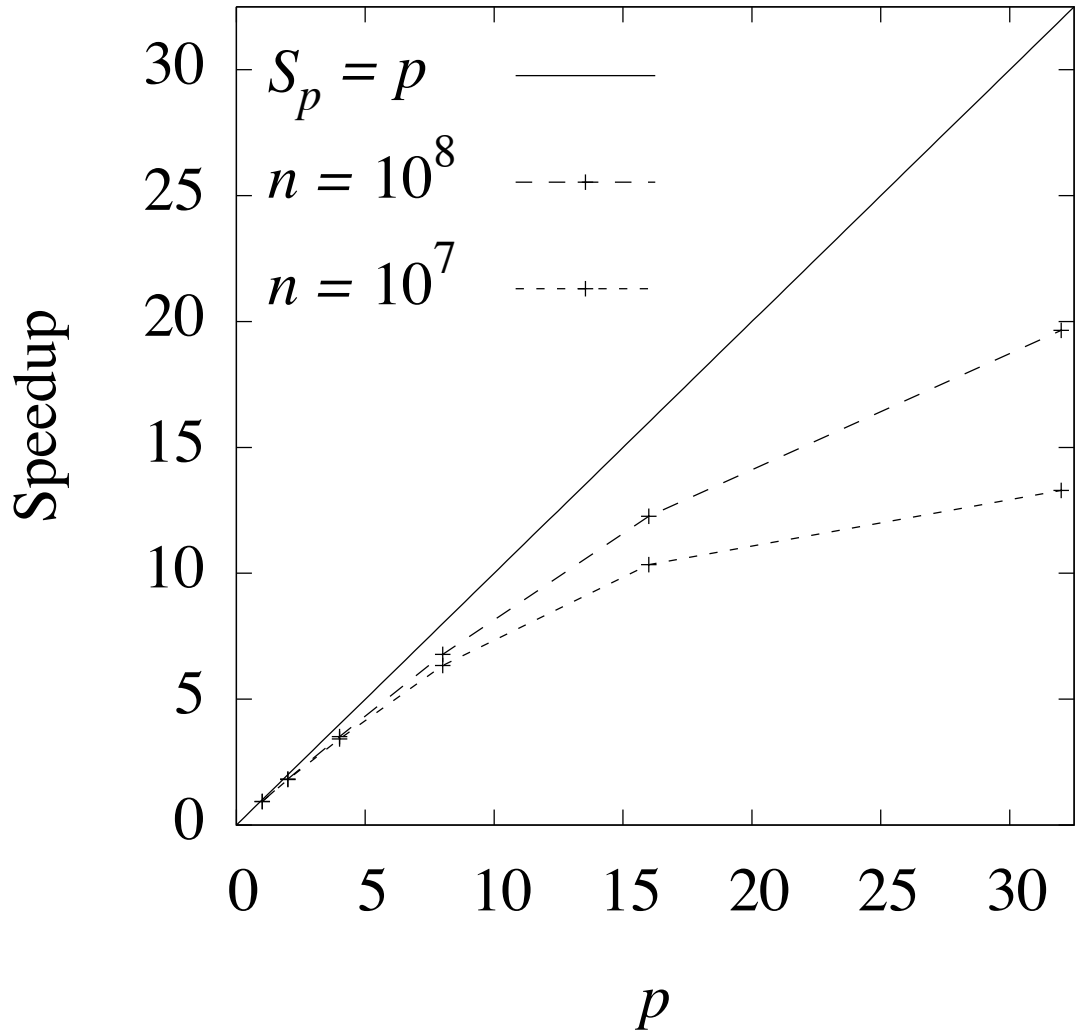
- The speedup of a parallel program is the increase in speed of the program running on p processors compared with the speed of a sequential program,

$$S_p(n) = \frac{T_{\text{seq}}(n)}{T_p(n)}.$$

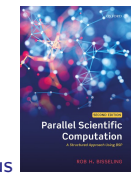
- Comparing to T_1 instead of T_{seq} would be too flattering, since the parallel program run for $p = 1$ will have overhead such as superfluous calculations.
- Often, you can obtain a good sequential program by simplifying the parallel program: substituting $p = 1, s = 0$, removing syncs, and replacing puts by memory copies.
- This is a rather mechanical process, which can be accompanied by a good drink.
- In this process, keeping track of run time gains may give insight into further optimization opportunities for the parallel program.



Speedup of parallel regular samplesort



- Highest speedup achieved: $S_{32}(10^8) = 19.8$.



Superlinear speedup

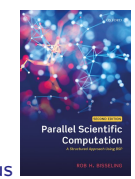
- Usually:
$$0 < S_p(n) \leq p.$$
- Still, a superlinear speedup $S_p(n) > p$ can happen, most likely because of cache effects.
- If each processor has its own cache, increasing p also increases the total problem size n that fits into cache.
- For a critical p, the computation rate r becomes a higher in-cache rate, instead of a lower out-of-cache rate.

Efficiency

- The efficiency is the fraction of the total computing power that is usefully employed. It is defined by

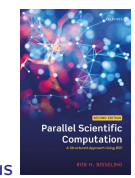
$$E_p(n) = \frac{S_p(n)}{p} = \frac{T_{\text{seq}}(n)}{pT_p(n)}.$$

- Usually $0 < E_p(n) \leq 1$ and ideally $E_p(n) = 1$.



Strong and weak scalability

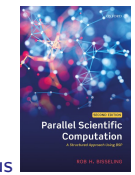
- Keeping the problem size fixed while increasing p is a test of strong scalability. We do this for measuring speedup.
- Sometimes this is difficult to implement, because large problems do not fit into the memory of one processor, and we are interested in the behaviour of exactly those problems.
- This holds especially for distributed-memory architectures, where the available memory grows linearly with p.
- For linear-time problems, a solution is to let the problem size n grow linearly as well with p, which is a test of weak scalability.
- Weak scalability is good if the efficiency stays close to 1.



Breakdown of predicted sorting time for $n = 10^8$

p	$\frac{n}{p} \log_2 \frac{n}{p}$	$p^2 \log_2 p$	$\frac{2n}{p} \log_2 p$	$p(p-1)g$	$\frac{2n}{p} g$	$5l$	T_p
1	0.465	0	0	0	6.899	0	7.364
2	0.224	0	0.018	0	3.485	0.000016	3.726
4	0.108	0	0.018	0	1.970	0.000034	2.095
8	0.052	0	0.013	0.000003	1.199	0.000043	1.264
16	0.025	0	0.009	0.000014	0.722	0.000082	0.756
32	0.012	0.000001	0.005	0.000079	0.498	0.000116	0.515
32	0.537	0.000124	0.247	0.000183	0.213	0.000102	0.998

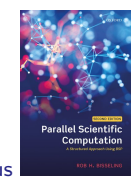
- Predictions (in s) are given for $p \leq 32$. For $p = 32$, they are based on benchmarked values $r = 5.711$ Gflop/s, $g = 455$, $l = 132\,618$.
- The bottom line is the actual measured time (in s).
- The dominant predicted cost is the cost $\frac{2n}{p} g$ of moving the data to their final destination.
- The measured cost is only $\frac{n}{p} g$, because the output block size is balanced for random test data, so $b_s \approx b$.



Misprediction of computing rate

p	$\frac{n}{p}\log_2\frac{n}{p}$	$p^2\log_2 p$	$\frac{2n}{p}\log_2 p$	$p(p-1)g$	$\frac{2n}{p}g$	$5l$	T_p
1	0.465	0	0	0	6.899	0	7.364
2	0.224	0	0.018	0	3.485	0.000016	3.726
4	0.108	0	0.018	0	1.970	0.000034	2.095
8	0.052	0	0.013	0.000003	1.199	0.000043	1.264
16	0.025	0	0.009	0.000014	0.722	0.000082	0.756
32	0.012	0.000001	0.005	0.000079	0.498	0.000116	0.515
32	0.537	0.000124	0.247	0.000183	0.213	0.000102	0.998

- For $p = 32$, the main computation term $\frac{n}{p}\log_2\frac{n}{p}$ representing the local quicksort shows a large discrepancy between prediction (0.012 s) and measurement (0.537 s).
- This must be due to slower scalar operations of the sorting compared to the vector operations of the DAXPY benchmark.
- Better prediction: use a sorting-based benchmark rate r.



Summary

- The speedup of a parallel program run on p processors for a problem of size n is

$$S_p(n) = \frac{T_{\text{seq}}(n)}{T_p(n)}.$$

- The efficiency is

$$E_p(n) = \frac{S_p(n)}{p}.$$

- Usually we have

$$0 < S_p(n) \leq p, \qquad 0 < E_p(n) \leq 1,$$

and ideally

$$S_p(n) = p, \qquad E_p(n) = 1.$$

- Theoretical BSP cost analysis gives insight and helps predict run time behaviour of parallel programs, but we should not get carried away and have unrealistic expectations of our predictions.