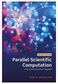
BSP Benchmarking

Sections 1.5–1.7 of Parallel Scientific Computation, 2nd edition

Rob H. Bisseling

Utrecht University

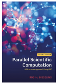


Benchmarking: art, science, magic?

"There are three kinds of lies: lies, damned lies, and statistics"

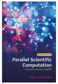
(wrongly attributed in 1907 by Mark Twain to Benjamin Disraeli, who probably never said this)

- Benchmarking is the activity of comparing performance.
- Computer benchmarking involves running computer programs to see how certain computer systems perform. This checks both the hardware and the system software.
- The benchmark result is obtained by ruthless reduction of a large quantity of data to one statistical figure, the flop rate.



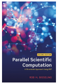
Sequential benchmarking

- Already for sequential computers, benchmarking is difficult, because different programs can run at very different speeds on the same machine.
- Reaching only 2% of the peak rate of a computer is quite common these days, especially for irregular computations. No one is embarrassed. Hush!
- The lowest computing rates are obtained for scalar operations, which involve single numbers.
- Higher rates can be obtained for operations on vectors and matrices.



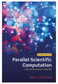
Basic Linear Algebra Subprograms

- Matrix and vector operations have been implemented efficiently in the Basic Linear Algebra Subprograms (BLAS) library.
- The highest computing rates can be achieved by algorithms that use matrix–matrix multiplication, such as the BLAS level-3 operation DGEMM.
- An intermediate rate is obtained for vector–vector operations, such as the BLAS level-1 operation DAXPY, defined by $\mathbf{y} := \alpha \mathbf{x} + \mathbf{y}$.
- We use the DAXPY for sequential benchmarking.

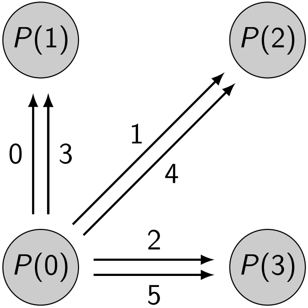


BSP benchmarking

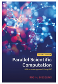
- We must be ruthless, but a single number will not work. Thus we measure: r for computation, g for communication, and l for synchronization.
- The aim is to obtain useful values of r, g, l that help us in predicting performance of algorithms without actually running an implementation.
- Most of our troubles in this endeavour come from the difficulty of sequential benchmarking.
- A cache is a small memory close to the CPU that stores recently accessed data. There may be a tiny primary (L1) cache, a larger secondary (L2) cache farther away, etc.
- Computations in primary cache are much faster than others. We may have to distinguish rates r_1, r_2, etc. (but we won't).



Communication pattern for BSP benchmark program

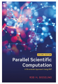


- $P(0)$ sends a data word to $P(1)$, then to $P(2)$, $P(3)$, $P(1)$, $P(2)$, $P(3)$.
- The other processors also send data in this cyclic fashion.
- The pattern is a 6-relation.

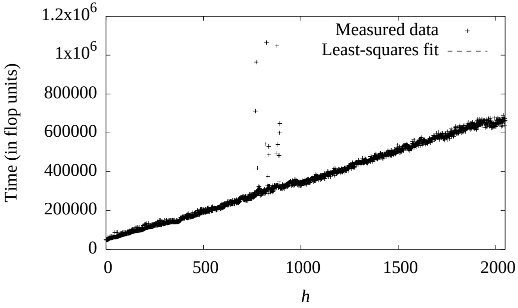


Full h-relation

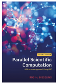
- We measure a full h-relation, where every processor sends and receives exactly h data.
- Our intentions are the worst: we try to measure the slowest possible communication. We put single data words into other processors in a cyclic fashion.
- This reveals whether the system software indeed combines data for the same destination and whether it can handle all-to-all communication efficiently, which is the basis of BSP.
- ‘Underpromise and overdeliver’ is the motto: actual communication performance can only be better. We call the value of g obtained by our benchmarking program `bspbench` pessimistic.
- By sending larger packets of data, instead of single words, we can measure an optimistic g-value.



Time of an h-relation on 32-core compute server Gemini



- Hardware: compute server Gemini of the Faculty of Science of Utrecht University, with two Intel Xeon E5-2683 CPUs, each with 16 cores, running at 2.1 GHz.
- Software: Scientific Linux operating system; MulticoreBSP for C, v2.0.4, which is a BSP library for shared memory.
- Trying to be kind to other users: $p = 24 < p_{\mathrm{max}} = 32$.
- $r = 2.3$ Gflop/s, $g = 309$, and $l = 46\,224$.



Least-squares fit

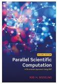
- Two measurements would suffice for obtaining a straight line, but we want to use all available data in an interval $[h_0, h_1]$.
- We minimize the error

$$E_{\mathrm{LSQ}}(g, l) = \sum_{h=h_0}^{h_1} (T_{\mathrm{comm}}(h) - (hg + l))^2,$$

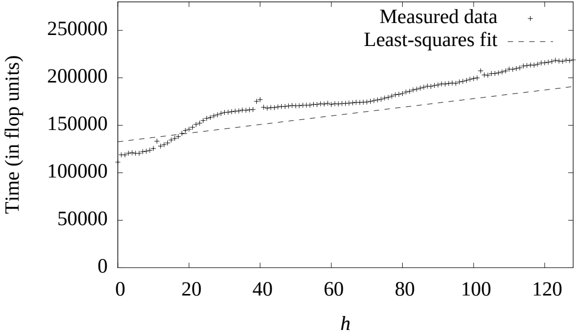
  where $T_{\mathrm{comm}}(h)$ is the measured time, and $hg + l$ the time predicted by the model.
- The best choice for g and l is obtained by setting

$$\frac{\partial E}{\partial g} = \frac{\partial E}{\partial l} = 0$$

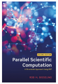
  and solving the resulting 2×2 linear system.



Time of an h-relation for $p = 32$ on Cartesius



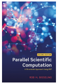
- Hardware: Dutch national supercomputer Cartesius at SURFsara in Amsterdam. One Broadwell node with 32 cores, running at 2.6 GHz.
- Software: MulticoreBSP for C, v2.0.4.
- $r = 5.711$ Gflop/s, $g = 455$, and $l = 132\,618$.



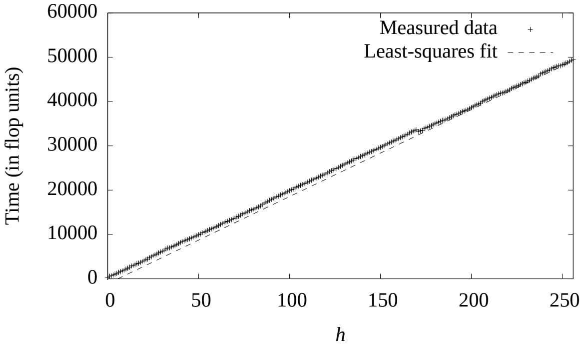
Benchmarked BSP parameters p, g, l on Cartesius

p	g	l	$T_{\mathrm{comm}}(0)$
1	197	–	294
2	199	18 408	6 759
3	215	24 438	8 932
4	225	38 275	14 291
5	247	30 783	17 970
6	262	38 670	20 322
7	242	56 010	24 781
8	274	49 655	27 609
12	300	82 374	40 879
16	330	93 365	52 653
20	403	103 090	70 562
24	409	107 769	88 262
28	451	124 240	106 754
32	455	132 618	111 267

- The time of a 0-relation $T_{\mathrm{comm}}(0) \leq l$.

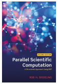


Time of an h-relation for $p = 1$ on Cartesius



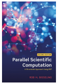
Plotting helps understand strange behaviour:

- Negative l: both g, l are small and of the same order.
- Sending more data takes less time for $h \approx 170$: switching too late to a different data packing mechanism.



bspbench: initializing the communication pattern

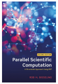
```
#define MAXH 2048        // maximum h in h-relation

long destproc[MAXH], destindex[MAXH];
double src[MAXH];

for (long i=0; i<h; i++){
    src[i]= (double)i;
    if (p==1){
        destproc[i]= 0;
        destindex[i]= i;
    } else {
        // destination proc is one of the p-1 others
        destproc[i]= (s+1 + i%(p-1)) %p;
        // destination index is in my own part of dest
        destindex[i]= s + (i/(p-1))*p;
    }
}
```



bspbench: measuring the communication time

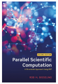
```
#define NITERS 1000    // number of iterations

bsp_sync();
double time0= bsp_time();

for (long iter=0; iter<NITERS; iter++){
    for (long i=0; i<h; i++)
        bsp_put(destproc[i],&src[i],dest,
                destindex[i]*sizeof(double),
                sizeof(double));
    bsp_sync();
}

double time1= bsp_time();
double time= time1-time0;
```

- Increase `NITERS` to obtain more accurate measurements and smoother plots.
- But if `NITERS` is too large, you will wait forever.



Advice from the trenches

- Always plot the benchmark results. This gives insight into your machine and reveals the accuracy of your measurement.
- Be suspicious of artefacts. Negative g values may occur if g is small and l is huge. Then, the least-squares fit gives an inaccurate g and you have to enlarge the measurement interval $[h_0, h_1]$.
- Run the benchmark at least three times. If the best two runs agree, you can be reasonably confident.
- Parallel computers are like the weather: they change all the time. Always run a benchmark program before running an application program, just to see what machine you have today.
- Possible changes: new compiler, faster communication switches, Challenge Projects that gobble up network resources.



Summary

- Benchmarking is difficult.
- Machines have quirks, surprises are plenty, and measurements are often inaccurate.
- With all these caveats, it is still useful to have the r, g, l values for many different machines.
- BSP benchmarking can be done for
  - BSPlib/C by using `bspbench.c` from BSPedupack v2.0;
  - MPI-1/C by using `mpibench.c` from MPIedupack v1.0;
  - Bulk/C++ by using `benchmark.cpp` written by Jan-Willem Buurlage.