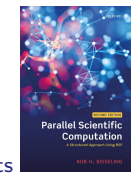
Weights for the FFT

Section 3.5 of Parallel Scientific Computation, 2nd edition

Rob H. Bisseling

Utrecht University



Weights for sequential computation

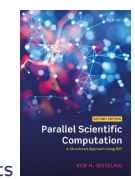
- The weights of the FFT are the powers of ω_n that are needed in the FFT computation: $1, \omega_n, \omega_n^2, \ldots, \omega_n^{n/2-1}$.
- We can compute these powers by

$$\omega_n^j = \mathrm{e}^{-2\pi ij/n} = \cos\frac{2\pi j}{n} - i\sin\frac{2\pi j}{n}.$$

- Computing the weights by successive multiplication

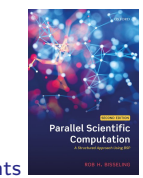
$$\omega_n^{j+1} = \omega_n \cdot \omega_n^j$$

is less accurate and not recommended.



Cost in flops

- Typically, computing a sine or cosine costs 10 flops in double precision accuracy.
- If we compute a weight each time we need it, we perform 20 flops extra for every 10 flops (complex $*, +, -$) in the inner loop of the FFT. This would triple the total cost.
- Alternative: compute the weights once and store them in a table.



Using symmetry to compute weights faster

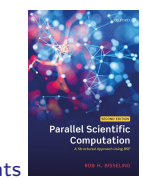
- We can save half the computations by using

$$\omega_n^{n/2-j} = \mathrm{e}^{-2\pi i(n/2-j)/n} = \mathrm{e}^{-\pi i}\mathrm{e}^{2\pi ij/n} = -\overline{(\omega_n^j)}.$$

  Thus, we only need to compute $1, \omega_n, \omega_n^2, \ldots, \omega_n^{n/4}$.
- Taking negatives and complex conjugates is extremely cheap.
- Similarly, we can halve the work again by using

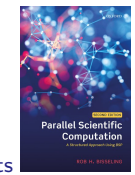
$$\omega_n^{n/4-j} = -i\overline{(\omega_n^j)}.$$

  Now, we only need to compute $1, \omega_n, \omega_n^2, \ldots, \omega_n^{n/8}$.
- The total cost of the weight initialization is thus about $20 \cdot n/8 = 2.5n$ flops.



Weights for parallel computation

- A brute-force approach: store the complete table of weights on every processor.
- This approach is nonscalable in memory: in the sequential case, we store n vector components and $n/2$ weights. In the parallel case, n/p vector components and $n/2$ weights per processor.
- Furthermore, for small n or large p, the $2.5n$ flops of the weight initialization may be much more than the $(5n \log_2 n)/p$ local flops of the FFT.
- Some replication of weights is inevitable: stages $k = 2, 4, \ldots, n/p$ are the same on all processors and hence need the same weights.
- Our goal is to find a memory-scalable approach that adds only a few flops to the overall count.

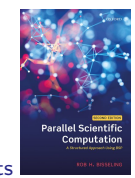


Memory scalability

- We call the memory requirements $M(n, p)$ of a BSP algorithm scalable if this amount satisfies

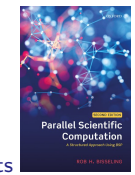
$$M(n, p) = \mathcal{O}\left(\frac{M_{\text{seq}}(n)}{p} + p\right).$$

- Motivation of the $\mathcal{O}(p)$ term: BSP algorithms are based on all-to-all communication supersteps, where each processor deals with $p - 1$ others, and needs $\mathcal{O}(p)$ buffer memory for storing communication meta-data.
- For example, calling a `bsp_push_reg` primitive for a variable gives rise to $p - 1$ remote adresses being stored locally.
- Assumption: all processors possess the same amount of memory.

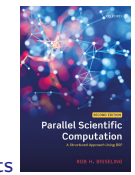


Approach: dry run of the parallel algorithm

- Perform a dry run of the parallel algorithm to compute and store the needed weights in the order of their use.
- This initializes the weight table.
- For stage k and the group-cyclic distribution with cycle c, we need to compute the $\frac{k}{2c}$ complex weights of the local part of the butterfly matrix B_k.



A real-life butterfly — in case you forgot

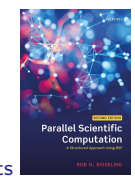


Memory needed for computation supersteps with $c < p$

- For stage k and the group-cyclic distribution with cycle c, we need $\frac{k}{2c}$ complex weights.
- In a computation superstep with $c < p$, we perform stages $k = 2c, 4c, \ldots, \frac{n}{p}c$, which respectively need $1, 2, \ldots, \frac{n}{2p}$ weights.
- The total number of (complex) weights of the superstep is

$$1 + 2 + 4 + \cdots + \frac{n}{2p} = \frac{n}{p} - 1,$$

so that we need to store at most $\frac{2n}{p}$ real numbers (1 complex number $=$ 2 reals).

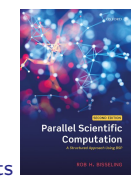


Memory needed for the final computation superstep

- For the final stage $k = n$ and the group-cyclic distribution with cycle $c = p$, we need $\frac{k}{2c} = \frac{n}{2p}$ complex weights.
- Similar to the previous supersteps, the number of weights needed doubles at every stage in the final superstep.
- Therefore, the total number of (complex) weights of the final superstep is

$$\cdots + \frac{n}{4p} + \frac{n}{2p} \leq \frac{n}{p} - 1,$$

so that, here too, we need to store at most $\frac{2n}{p}$ real numbers.



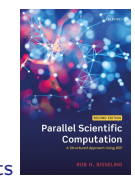
Total memory needed for all computation supersteps

- We have $t+1$ computation supersteps (as well as t communication supersteps), where

$$t = \left\lceil \frac{\log_2 p}{\log_2(n/p)} \right\rceil,$$

  which is the smallest integer with $\left(\frac{n}{p}\right)^t \geq p$.
- The total amount of memory used per processor in reals is

$$2(t+1)\frac{n}{p}.$$



Some algebra (and calculus)

- In a suitable range of values, multiplying r terms x grows faster than adding them.
- For $x \geq 2$ and integer $r \geq 1$, it holds that

$$x^r \geq rx.$$

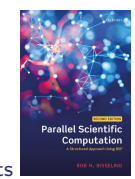
- Proof sketch:
  - For $r = 1$, the inequality holds trivially.
  - For fixed $r \geq 2$, define a function

$$f(x) = x^r - rx.$$

    Then

$$f'(x) = r(x^{r-1} - 1) > 0,$$

    so f is strictly monotonically increasing for $x \geq 2$.
  - The proof is completed by showing that $f(2) \geq 0$, which can be done by induction on r.



Substitution achieves the final result

- Substituting $r = t - 1$ and $x = \frac{n}{p}$ into $rx \leq x^r$ and applying the definition of t, we obtain

$$(t-1)\frac{n}{p} \leq \left(\frac{n}{p}\right)^{t-1} < p.$$

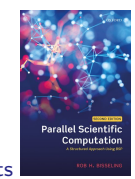
- As a result, the total memory required for the weights is

$$2(t+1)\frac{n}{p} = \frac{4n}{p} + 2(t-1)\frac{n}{p} < \frac{4n}{p} + 2p.$$

- Adding the $\frac{2n}{p}$ memory needed for the vector $\mathbf{x}$ gives the total memory use of the parallel FFT,

$$M_{\mathrm{FFT}} = \frac{6n}{p} + 2p,$$

which is scalable.



Summary

- We can compute a weight ω_n^j of the FFT by evaluating a cosine and a sine. This costs about 10 flops per evaluation.
- Doing this every time we use a weight is too expensive.
- It is better to store the weights in a dry run of the algorithm, in the order of their use.
- Reusing the same weights for several butterfly operations B_k leads to scalable memory use,

$$M(n,p) = \mathcal{O}\left(\frac{M_{\text{seq}}(n)}{p} + p\right).$$

- This approach has the additional advantage of being cache-friendly, because all weights used in a butterfly are stored together, and they are reused several times.