Introduction to the Bulk Synchronous Parallel model

Albert-Jan N. Yzelman

11th of October, 2013

Derived from the book slides by Prof. dr. Rob H. Bisseling, found at

`www.math.uu.nl/people/bisseling/Education/PA/pa.html`

Programming using BSPlib

1 Programming using BSPlib

2 BSP benchmarking

3 The sequential LU decomposition

4 Parallel LU decomposition

Available BSP libraries

Distributed-memory computers:

1. Oxford BSP library, see `www.bsp-worldwide.org`
2. BSPonMPI, see `bsponmpi.sourceforge.net`

Shared-memory computers, see `www.multicorebsp.com`:

1. MulticoreBSP for Java
2. **MulticoreBSP for C**

Compilation of BSPlib programs

Oxford BSP Toolset (Hill, McColl, Stefanescu, Goudreau, Lang, Rao, Suel, Tsantilas, Bisseling; 1998)

```
bspcc bspinprod.c;
bsprun -npes <P> ./a.out
```

BSPonMPI (van Suijlen; 2006)

```
mpcc bspinprod.c -lbsponmpi;
./a.out
```

MulticoreBSP for C (Yzelman, Bisseling, Roose, Meerbergen; 2012)

```
cc bspinprod.c -lmcbsp -pthread -lrt;
./a.out
```

MulticoreBSP for Java (Yzelman, Bisseling; 2010)

```
javac -cp MulticoreBSP.jar bspinprod.java;
java -cp MulticoreBSP.jar bspinprod
```

Other BSP libraries

Other (distributed memory) BSP-style libraries (incompatible API):

- MapReduce

  Google Inc.; 2004.
- Pregel

  Malewicz, Austern, Bik, Dehnert, Ilan Horn, Czajkowski (Google Inc.); 2010.
- Apache Hama

  Yoon et al.; 2010
- Paderborn University BSP library (PUB)

  Bonorden, Juurlink, von Otte, Rieping; 1998.
- Bulk Synchronous Parallel ML (BSMLlib)

  Gava, Gesbert, Hains, Tesson; 2000.
- Python/BSP

  Hinsen, Sadron; 2003.
- Cloudscale BSP

  McColl et al. (Cloudscale Inc.); 2012.

Hello World!

```
#include "bsp.h" / "mcbsp.h"

int main(int argc, char **argv) {
    bsp_begin( 4 );

    printf( "Hello world from thread %d out of %d!\n",
        bsp_pid(), bsp_nprocs() );

    bsp_end();
}
```

Hello World!

An example using MulticoreBSP for C and static linkage:

```
$ gcc -o hello hello.c lib/libmcbsp1.1.0.a -pthread -lrt
$ ./hello
Hello world from thread 3 out of 4!
Hello world from thread 2 out of 4!
Hello world from thread 0 out of 4!
Hello world from thread 1 out of 4!
$ ...
```

Hello World!

```
#include "mcbsp.h"
int P;

void hello() {
    bsp_begin( P );
    printf( "Hello world from thread %d!\n",
        bsp_pid() );
    bsp_end();
}

int main(int argc, char **argv) {
    bsp_init( hello, argc, argv );
    scanf( "%d", &P );
    hello();
}
```

Summary

- There are multiple implementations of the BSPlib interface.
- Other BSP-style libraries for parallel programming exist; some stay close to the BSP model, others do not.
- We have seen the syntax for compilation for various BSPlib libraries, and applied these on a ‘Hello world!’ example.
- The complete program `bspinprod` should now be clear from the last section. Try to compile it and to run it!

BSP benchmarking

1 Programming using BSPlib

2 BSP benchmarking

3 The sequential LU decomposition

4 Parallel LU decomposition

Benchmarking: art, science, magic?

“There are three kinds of lies: lies, damned lies, and statistics” (Benjamin Disraeli, 1804–1881)

- 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.
- Often, the benchmark result is obtained by ruthless reduction of a large quantity of data to one statistical figure, the flop rate.

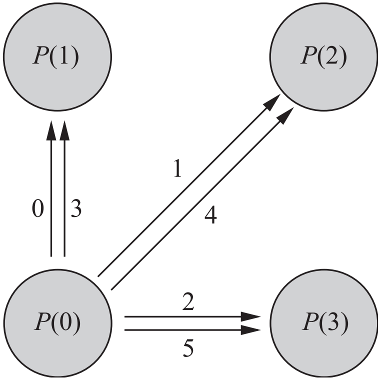
Sequential benchmarking

- Already for sequential computers, **benchmarking is difficult**; different programs can run at very different speeds.
- Reaching only 10% of the peak rate of a computer is quite common. No one is embarrassed. Hush!
- Highest rates are obtained by algorithms that use matrix–matrix multiplication, such as implemented in the **BLAS** level 3 operation DGEMM. (BLAS = Basic Linear Algebra Subprograms).
- The arithmatic intensity is key; if the flop/byte is high, the problem is CPU bound and usually can be well-optimised.
- In contrast, irregular scalar operations which involve single numbers and not vectors or matrices, are bandwidth-bound.
- A reasonable intermediate rate is obtained for vector–vector operations, such as the BLAS level 1 DAXPY, defined by $\mathbf{y} := \alpha\mathbf{x} + \mathbf{y}$. We use this operation for sequential benchmarking.

BSP benchmarking

- We must be ruthless, but a single number will not work. Thus we look to the BSP model and measure r for **computation**, g for **communication**, and l for **synchronisation**.
- 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; computation speed is hard to determine.
- Apart from arithmatic intensity, **caching** plays a big role:
  - A cache is a small memory close to the CPU that stores recently accessed data. There may be a tiny primary cache, a larger secondary cache farther away, etc.
  - Higher-level caches are bigger but slower.
  - 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

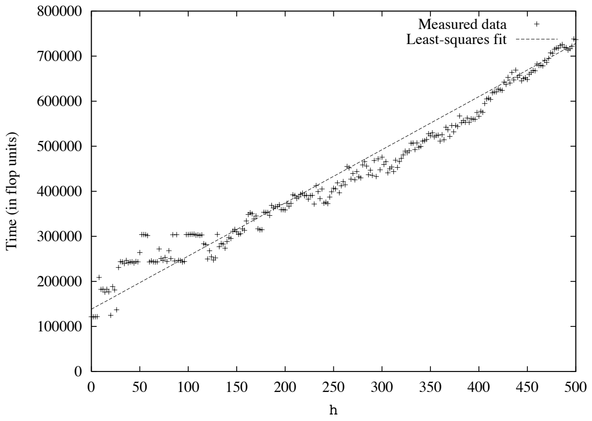


$P(0)$ sends data to $P(1)$, $P(2)$, $P(3)$, $P(1)$, $P(2)$, $P(3)$. The other processors also send data in this cyclic fashion.

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. This is after all the basis of BSP!
- 'Underpromise and overdeliver' is the motto: actual communication performance can only be better; the g obtained by our benchmarking program `bspbench` is pessimistic.
- The Oxford BSP toolset has another benchmarking program, `bspprobe`, which measures optimistic g-values.

Time of an *h*-relation on two connected PCs



Two 400 MHz Pentium II PCs, both running Linux, connected by Fast Ethernet (100 Mbit/s) and a Cisco Catalyst switch:

$r = 122$ Mflop/s, $g = 1180$, and $l = 138324$.

Least-squares fit

- Two measurements would suffice for obtaining a straight line, but we want to use all $h_1 - h_0 + 1$ data points available in $[h_0, h_1]$.
- We minimise the squared error

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

- We find the best choice for g and l when

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

  and solving the resulting 2×2 linear system.
- Measure $t_i = T_{\mathrm{comm}}(i)$ for all $h_0 \leq i \leq h_1$. Then:

$$\frac{\partial E}{\partial g} = \sum_{i=h_0}^{h_1} 2i(ig + l) - 2t_i, \quad \frac{\partial E}{\partial l} = \sum_{i=h_0}^{h_1} 2(ig + l) - 2t_i.$$

Least-squares fit

- Two measurements would suffice for obtaining a straight line, but we want to use all $h_1 - h_0 + 1$ data points available in $[h_0, h_1]$.
- We minimise the squared error

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

- We find the best choice for g and l when

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

  and solving the resulting 2×2 linear system.
- Measure $t_i = T_{\mathrm{comm}}(i)$ for all $h_0 \leq i \leq h_1$. Then:

$$\frac{h_1(h_1+1)(2h_1+1) - h_0(h_0+1)(2h_0+1)}{3} \cdot g + (h_0 + h_1) \cdot l = 2 \sum_{i=h_0}^{h_1} it_i.$$

Least-squares fit

- Two measurements would suffice for obtaining a straight line, but we want to use all $h_1 - h_0 + 1$ data points available in $[h_0, h_1]$.
- We minimise the squared error

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

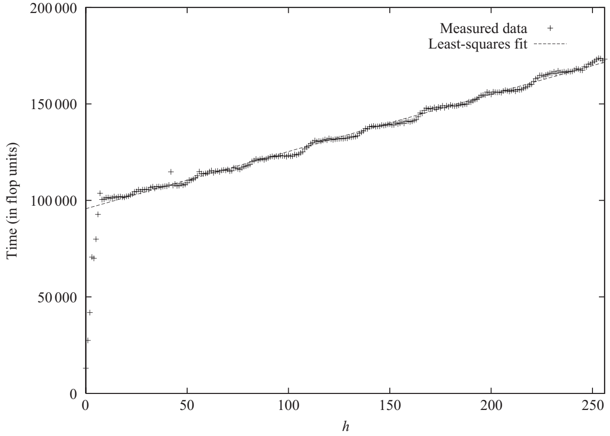
- We find the best choice for g and l when

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

  and solving the resulting 2×2 linear system.
- Measure $t_i = T_{\mathrm{comm}}(i)$ for all $h_0 \leq i \leq h_1$. Then:

$$(h_0 + h_1) \cdot g + 2(h_1 - h_0 + 1) \cdot l = 2 \sum_{i=h_0}^{h_1} t_i.$$

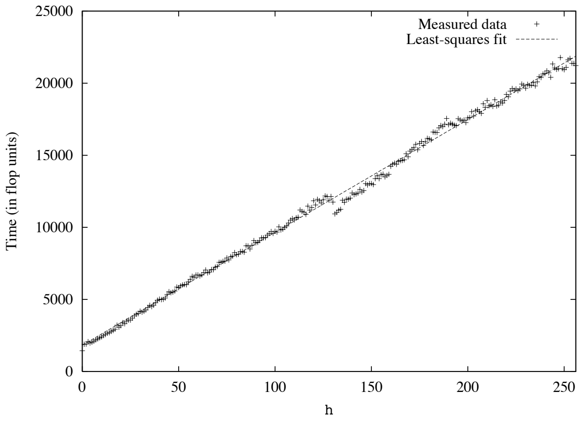
Time of an h-relation on an 8-processor SGI Origin



Silicon Graphics Origin 2000, Compiler plays tricks: measured value of r too high. Choose h_0 and h_1 judiciously! Here, $h_0 = p$.

$r = 326$ Mflop/s, $g = 297$, and $l = 95\ 686$.

Time of an h-relation on a 64-processor Cray T3E



Sending more data takes less time (cf. $h \approx 130$). Weird! Explanation: switching to a different data packing mechanism.

$r = 35$ Mflop/s, $g = 78$, and $l = 1825$.

bspbench: initialising the communication pattern

```
for( i = 0; i < h; i++ ) {
    src[ i ] = (double)i;
    if( p == 1 ) {
        destproc [ i ] = 0;
        destindex[ i ] = i;
    } else {
        /* destination processor 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

```
bsp_sync();

time0 = bsp_time();

for( iter = 0; iter < NITERS; iter++ ) {
    for( i = 0; i < h; i++ )
        bsp_put( destproc[ i ], &src[ i ], dest,
                 destindex[ i ] * SZDBL, SZDBL
               );
    bsp_sync();
}
time1 = bsp_time();
```

Adjust `NITERS` to obtain an accurate measurement.

Comparing BSP parameters ($p = 8$)

Computer	r (Mflop/s)	g (flop)	l (flop)	g (μs)	l (μs)
Cray T3E	35	31	1 193	0.88	34
IBM RS/6000 SP	212	187	148 212	0.88	698
SGI Origin 2000	326	297	95 686	0.91	294

- Machines become obsolete quickly. All of the above machines have in the mean time been replaced by faster successors.
- Newer machines will be benchmarked in the laboratory class of this course.

Advice from the trenches

- Always plot the benchmark results. This gives insight in 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. In that case, the least-squares fit does not give an accurate g.
- 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.
  (Think of: a new compiler, faster communication switches, Challenge Projects that gobble up network resources, and so on.)

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 a table with r, g, l values for many different machines.
- This table should be kept up to date to reflect new architectures appearing. You can do it! (Similar to the LINPACK benchmark used to determine the Supercomputer Top 500.)
- BSP benchmarking can be done using BSPlib (`bspbench`, `bspprobe`), but also MPI-1 (`mpibench`).

The sequential LU decomposition

1 Programming using BSPlib

2 BSP benchmarking

3 The sequential LU decomposition

4 Parallel LU decomposition

Solving linear systems is important

Applications often have as their core a linear system solver.

- Building bridges. Finite element models in engineering give rise to linear systems involving a stiffness matrix.
- Designing aircraft. Boundary element methods lead to huge dense linear systems of equations.
- Optimising oil refineries. Linear programming by interior point methods requires solving a sparse linear system (with many zero coefficients) at every step of the computation.

Lower and upper triangular matrices

$$A = \begin{bmatrix} 1 & 4 & 6 \\ 2 & 10 & 17 \\ 3 & 16 & 31 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 4 & 6 \\ 0 & 2 & 5 \\ 0 & 0 & 3 \end{bmatrix} = LU.$$

- L is unit lower triangular if $l_{ii} = 1$ for all i and $l_{ij} = 0$ for all $i < j$.
- U is upper triangular if $u_{ij} = 0$ for all $i > j$.

Triangular systems are easier to solve

Let $A = LU$. Then

$$A\mathbf{x} = \mathbf{b} \iff L(U\mathbf{x}) = \mathbf{b} \iff L\mathbf{y} = \mathbf{b} \text{ and } U\mathbf{x} = \mathbf{y}.$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 16 \\ 44 \\ 78 \end{bmatrix} \Longrightarrow \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 16 \\ 12 \\ 6 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 4 & 6 \\ 0 & 2 & 5 \\ 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 16 \\ 12 \\ 6 \end{bmatrix} \Longrightarrow \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}.$$

Objective

For a given A, find L, U such that $LU = A$, with

$$L = \begin{pmatrix} l_{00} & & \\ \vdots & \ddots & \\ l_{n0} & \cdots & l_{nn} \end{pmatrix}, \text{ and } U = \begin{pmatrix} u_{01} & \cdots & u_{0n} \\ & \ddots & \vdots \\ & & u_{nn} \end{pmatrix}.$$

The upper-right and bottom-left white-spaces of L, resp., U contain zero entries.

Fixing the diagonal

If $A = \hat{L}\hat{U}$, then we can take $D = \text{diag}(\hat{U})$ and write $A = DLU$ with U now of the following form:

$$U = \begin{pmatrix} 1 & u_{01} & \cdots & u_{0n} \\ & \ddots & \ddots & \vdots \\ & & \ddots & u_{n-1,n} \\ & & & 1 \end{pmatrix}.$$

The lower-triangular part can now be taken as $L = D\hat{L}$. This form of the LU decomposition is **unique** if A is invertable and at least one decomposition $A = \hat{L}\hat{U}$ exists.

In-place storage

Fixing the diagonal enables us to store the matrices L and U within an $n \times n$ matrix, just like A itself:

$$LU = \begin{pmatrix} l_{00} & u_{01} & \cdots & u_{0(n-1)} & u_{0n} \\ l_{10} & l_{11} & \ddots & u_{1(n-1)} & u_{1n} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ l_{(n-1)0} & l_{(n-1)1} & \cdots & l_{(n-1)(n-1)} & u_{(n-1)n} \\ l_{n0} & l_{n1} & \cdots & l_{n(n-1)} & l_{nn} \end{pmatrix}.$$

We can now store and compute the LU **in place**. This saves memory and prevents data movement. In-place algorithms are often a good idea.

Constructing the algorithm

We go by example:

$$A = \begin{pmatrix} 2 & 0 & 3 \\ 1 & 2 & 0 \\ 7 & 0 & 5 \end{pmatrix} = \begin{pmatrix} l_{00} & 0 & 0 \\ l_{10} & l_{11} & 0 \\ l_{20} & l_{21} & l_{22} \end{pmatrix} \cdot \begin{pmatrix} 1 & r_{01} & r_{02} \\ 0 & 1 & r_{12} \\ 0 & 0 & 1 \end{pmatrix}.$$

1. We process the first row.

Constructing the algorithm

We go by example:

$$A = \begin{pmatrix} 2 & 0 & 3 \\ 1 & 2 & 0 \\ 7 & 0 & 5 \end{pmatrix} = \begin{pmatrix} l_{00} & 0 & 0 \\ l_{10} & l_{11} & 0 \\ l_{20} & l_{21} & l_{22} \end{pmatrix} \cdot \begin{pmatrix} 1 & r_{01} & r_{02} \\ 0 & 1 & r_{12} \\ 0 & 0 & 1 \end{pmatrix}.$$

1. We process the first row.

Constructing the algorithm

We go by example:

$$A = \begin{pmatrix} 2 & 0 & 3 \\ 1 & 2 & 0 \\ 7 & 0 & 5 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ l_{10} & l_{11} & 0 \\ l_{20} & l_{21} & l_{22} \end{pmatrix} \cdot \begin{pmatrix} 1 & r_{01} & r_{02} \\ 0 & 1 & r_{12} \\ 0 & 0 & 1 \end{pmatrix}.$$

1. We process the first row.

Constructing the algorithm

We go by example:

$$A = \begin{pmatrix} 2 & 0 & 3 \\ 1 & 2 & 0 \\ 7 & 0 & 5 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ l_{10} & l_{11} & 0 \\ l_{20} & l_{21} & l_{22} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & r_{02} \\ 0 & 1 & r_{12} \\ 0 & 0 & 1 \end{pmatrix}.$$

1. We process the first row.

Constructing the algorithm

We go by example:

$$A = \begin{pmatrix} 2 & 0 & 1.5 \\ 1 & 2 & 0 \\ 7 & 0 & 5 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ l_{10} & l_{11} & 0 \\ l_{20} & l_{21} & l_{22} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1.5 \\ 0 & 1 & r_{12} \\ 0 & 0 & 1 \end{pmatrix}.$$

1. We process the first row: divide a_{0j} by a_{00}.

Constructing the algorithm

We go by example:

$$A = \begin{pmatrix} 2 & 0 & 1.5 \\ 1 & 2 & 0 \\ 7 & 0 & 5 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ l_{10} & l_{11} & 0 \\ l_{20} & l_{21} & l_{22} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1.5 \\ 0 & 1 & r_{12} \\ 0 & 0 & 1 \end{pmatrix}.$$

1. We process the first row: divide a_{0j} by a_{00}.
2. Try processing a_{10}: $l_{10} = a_{10}$.

Constructing the algorithm

We go by example:

$$A = \begin{pmatrix} 2 & 0 & 1.5 \\ 1 & 2 & 0 \\ 7 & 0 & 5 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 1 & l_{11} & 0 \\ l_{20} & l_{21} & l_{22} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1.5 \\ 0 & 1 & r_{12} \\ 0 & 0 & 1 \end{pmatrix}.$$

1. We process the first row: divide a_{0j} by a_{00}.
2. Proceed with the remainder of the first column:

   $l_{20} = a_{20}$.

Constructing the algorithm

We go by example:

$$A = \begin{pmatrix} 2 & 0 & 1.5 \\ 1 & 2 & 0 \\ 7 & 0 & 5 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 1 & l_{11} & 0 \\ 7 & l_{21} & l_{22} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1.5 \\ 0 & 1 & r_{12} \\ 0 & 0 & 1 \end{pmatrix}.$$

1. We process the first row: divide a_{0j} by a_{00}.
2. Processing the remainder of the first column:

   $l_{j0} = a_{j0},\ 0 < j \leq n$; i.e., do nothing.

Constructing the algorithm

We go by example:

$$A = \begin{pmatrix} 2 & 0 & 1.5 \\ 1 & 2 & 0 \\ 7 & 0 & 5 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 1 & l_{11} & 0 \\ 7 & l_{21} & l_{22} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1.5 \\ 0 & 1 & r_{12} \\ 0 & 0 & 1 \end{pmatrix}.$$

1. We process the first row: divide a_{0j} by a_{00}.
2. Processing the remainder of the first column (a_{10}, a_{20}):
   $l_{j0} = a_{j0},\ 0 < j \leq n$; i.e., do nothing.
3. Can we simply recurse and perform the same operation again?

Constructing the algorithm

We go by example:

$$A = \begin{pmatrix} 2 & 0 & 1.5 \\ 1 & 2 & 0 \\ 7 & 0 & 5 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 1 & l_{11} & 0 \\ 7 & l_{21} & l_{22} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1.5 \\ 0 & 1 & r_{12} \\ 0 & 0 & 1 \end{pmatrix}.$$

1. We process the first row: divide a_{0j} by a_{00}.
2. Processing the remainder of the first column (a_{10}, a_{20}):
   $l_{j0} = a_{j0},\ 0 < j \leq n$; i.e., do nothing.
3. Can we simply recurse and perform the same operation again?
   No! 7 and 1.5 add an extra contribution to the submatrix.

Constructing the algorithm

We go by example:

$$A = \begin{pmatrix} 2 & 0 & 1.5 \\ 1 & 2 & 0-1.5 \\ 7 & 0 & 5-10.5 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 1 & l_{11} & 0 \\ 7 & l_{21} & l_{22} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1.5 \\ 0 & 1 & r_{12} \\ 0 & 0 & 1 \end{pmatrix}.$$

1. We process the first row: divide a_{0j} by a_{00}.
2. Processing the remainder of the first column:

   $l_{j0} = a_{j0},\ 0 < j \leq n$; i.e., do nothing.
3. Can we simply recurse and perform the same operation again?

   Yes: first correct submatrix in preperation for recursion by taking an outer product and subtracting.

Constructing the algorithm

We go by example:

$$A = \begin{pmatrix} 2 & 0 & 1.5 \\ 1 & 2 & -1.5 \\ 7 & 0 & -5.5 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 1 & l_{11} & 0 \\ 7 & l_{21} & l_{22} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1.5 \\ 0 & 1 & r_{12} \\ 0 & 0 & 1 \end{pmatrix}.$$

1. We process the first row: divide a_{0j} by a_{00}.
2. Processing the remainder of the first column:
   $l_{j0} = a_{j0},\ 0 < j \leq n$; i.e., do nothing.
3. Correct submatrix in preperation for recursion;
   taking an outer product and subtracting.
4. Recurse on the $(n-1) \times (n-1)$ submatrix.

Constructing the algorithm

We go by example:

$$A = \begin{pmatrix} 2 & 0 & 1.5 \\ 1 & 2 & -1.5 \\ 7 & 0 & -5.5 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 1 & 2 & 0 \\ 7 & l_{21} & l_{22} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1.5 \\ 0 & 1 & -\frac{3}{4} \\ 0 & 0 & 1 \end{pmatrix}.$$

1. We process the first row: divide a_{0j} by a_{00}.
2. Processing the remainder of the first column:
   $l_{j0} = a_{j0},\ 0 < j \leq n$; i.e., do nothing.
3. Correct submatrix in preperation for recursion;
   taking an outer product and subtracting.
4. Recurse on the $(n-1) \times (n-1)$ submatrix.

Constructing the algorithm

We go by example:

$$A = \begin{pmatrix} 2 & 0 & 1.5 \\ 1 & 2 & -\frac{3}{4} \\ 7 & 0 & -5.5 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 1 & 2 & 0 \\ 7 & 0 & l_{22} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1.5 \\ 0 & 1 & -\frac{3}{4} \\ 0 & 0 & 1 \end{pmatrix}.$$

1. We process the first row: divide a_{0j} by a_{00}.
2. Processing the remainder of the first column:

   $l_{j0} = a_{j0},\ 0 < j \leq n$; i.e., do nothing.
3. Correct submatrix in preperation for recursion;

   taking an outer product and subtracting.
4. Recurse on the $(n-1) \times (n-1)$ submatrix.

Constructing the algorithm

We go by example:

$$A = \begin{pmatrix} 2 & 0 & 1.5 \\ 1 & 2 & -\frac{3}{4} \\ 7 & 0 & -5.5-0 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 1 & 2 & -\frac{3}{4} \\ 7 & 0 & -5.5 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1.5 \\ 0 & 1 & -\frac{3}{4} \\ 0 & 0 & 1 \end{pmatrix}.$$

1. We process the first row: divide a_{0j} by a_{00}.
2. Processing the remainder of the first column (a_{10}, a_{20}):
   $l_{j0} = a_{j0},\ 0 < j \leq n$; i.e., do nothing.
3. Correct submatrix in preperation for recursion;
   taking an outer product and subtracting.
4. Recurse on the $(n-1) \times (n-1)$ submatrix.

Constructing the algorithm

We go by example:

$$A = \begin{pmatrix} 2 & 0 & 1.5 \\ 1 & 2 & -\frac{3}{4} \\ 7 & 0 & -5.5 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 1 & 2 & -\frac{3}{4} \\ 7 & 0 & -5.5 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1.5 \\ 0 & 1 & -\frac{3}{4} \\ 0 & 0 & 1 \end{pmatrix}.$$

1. We process the first row: divide a_{0j} by a_{00}.
2. Processing the remainder of the first column (a_{10}, a_{20}):
   $l_{j0} = a_{j0},\ 0 < j \leq n$; i.e., do nothing.
3. Correct submatrix in preperation for recursion;
   taking an outer product and subtracting.
4. Recurse on the $(n-1) \times (n-1)$ submatrix.

Memory-efficient sequential LU decomposition

input:	A : $n \times n$ matrix, $A = A^{(0)}$.
output:	A : $n \times n$ matrix, $A = L - I_n + U$, with
	L : $n \times n$ unit lower triangular matrix,
	U : $n \times n$ upper triangular matrix,
	I_n : $n \times n$ identity matrix,
	such that $LU = A^{(0)}$.

```
for $k := 0$ to $n-1$ do
    for $i := k+1$ to $n-1$ do
        $a_{ik} := a_{ik}/a_{kk}$;
    for $i := k+1$ to $n-1$ do
        for $j := k+1$ to $n-1$ do
            $a_{ij} := a_{ij} - a_{ik}a_{kj}$;
```

Row permutations needed

LU decomposition breaks down immediately in stage 0 for

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix},$$

because we try to divide by 0.

- A solution is to permute the rows suitably.
- Thus, we compute a permuted LU decomposition,

$$PA = LU.$$

- Here, P is a permutation matrix, obtained by permuting the rows of I_n.
- Output of LU decomposition of A: L, U, P.

Permutations and permutation matrices

Let $\sigma : \{0, \ldots, n-1\} \rightarrow \{0, \ldots, n-1\}$ be a permutation.
We define the permutation matrix P_σ corresponding to σ by

$$(P_\sigma)_{ij} = \begin{cases} 1 & \text{if } i = \sigma(j) \\ 0 & \text{otherwise.} \end{cases}$$

Thus, column j of P_σ is 1 in row $\sigma(j)$, and 0 everywhere else.

Lemma 2.5 Properties of P_σ

Let $\sigma : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ be a permutation. Let $\mathbf{x}$ be a vector of length n and A an $n \times n$ matrix. Then

$$(P_\sigma \mathbf{x})_i = x_{\sigma^{-1}(i)}, \quad \text{for } 0 \le i < n,$$

$$(P_\sigma A)_{ij} = a_{\sigma^{-1}(i),j}, \quad \text{for } 0 \le i,j < n,$$

$$(P_\sigma A P_\sigma^T)_{ij} = a_{\sigma^{-1}(i),\sigma^{-1}(j)}, \quad \text{for } 0 \le i,j < n.$$

Proofs: see book.

LU decomposition with row permutations

input: A : $n \times n$ matrix, $A = A^{(0)}$.

output: A : $n \times n$ matrix, $A = L - I_n + U$, with

L : $n \times n$ unit lower triangular matrix,

U : $n \times n$ upper triangular matrix,

π : permutation vector of length n.

```
for $i := 0$ to $n-1$ do
$\pi_i := i$;
for $k := 0$ to $n-1$ do
    $r := \mathrm{argmax}(|a_{ik}| : k \leq i < n)$;
    swap$(\pi_k, \pi_r)$;
    for $j := 0$ to $n-1$ do
        swap$(a_{kj}, a_{rj})$;
    ...
```

LU decomposition with row permutations

input:	A : $n \times n$ matrix, $A = A^{(0)}$.
output:	A : $n \times n$ matrix, $A = L - I_n + U$, with
	L : $n \times n$ unit lower triangular matrix,
	U : $n \times n$ upper triangular matrix,
	π : permutation vector of length n.

```
...
for $i := k+1$ to $n-1$ do
        $a_{ik} := a_{ik}/a_{kk}$;
for $i := k+1$ to $n-1$ do
        for $j := k+1$ to $n-1$ do
                $a_{ij} := a_{ij} - a_{ik}a_{kj}$;
```

Partial row pivoting

- The pivot element in stage k is the largest element a_{rk} in column k. Everything revolves around it. It is farthest from 0 and division by a_{rk} is most stable.
- The pivot row r is thus determined by

$$|a_{rk}| = \max(|a_{ik}| : k \leq i < n).$$

- r is the argument (or index) of the maximum.
- Full pivoting would take the largest pivot from the whole submatrix $A(k\colon n-1, k\colon n-1)$. This gives the best stability, but is more costly (columns must be swapped, too).
  In practice, **partial pivoting** suffices.

The meaning of π

- The algorithm permutes the matrix by a permutation matrix P_σ. We obtain the LU decomposition $P_\sigma A = LU$.
- The same matrix is applied to the initial vector $\mathbf{e} = (0, 1, 2, \ldots, n-1)^T$. We obtain $\pi = P_\sigma \mathbf{e}$.
- Therefore, by Lemma 2.5,

$$\pi(i) = (P_\sigma \mathbf{e})_i = e_{\sigma^{-1}(i)} = \sigma^{-1}(i).$$

- Thus, $\pi = \sigma^{-1}$ and hence

$$P_{\pi^{-1}} A = LU.$$

Sequential time complexity

Lemma 2.7:

$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2}, \quad \sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}.$$

Proof: By induction on n.

The number of flops of the LU decomposition algorithm is

$$\begin{aligned} T_{\mathrm{seq}} &= \sum_{k=0}^{n-1}(2(n-k-1)^2 + n-k-1) = \sum_{k=0}^{n-1}(2k^2+k) \\ &= \frac{(n-1)n(2n-1)}{3} + \frac{(n-1)n}{2} \\ &= (n-1)n\left(\frac{2n}{3}+\frac{1}{6}\right) = \frac{2n^3}{3} - \frac{n^2}{2} - \frac{n}{6}. \end{aligned}$$

Summary

- Solving a linear system $A\mathbf{x} = \mathbf{b}$ can best be done by:
  - finding an LU decomposition $PA = LU$;
  - permuting $\mathbf{b}$ into $P\mathbf{b}$;
  - solving the triangular systems $L\mathbf{y} = P\mathbf{b}$ and $U\mathbf{x} = \mathbf{y}$.
- The LU decomposition costs about $2n^3/3$ flops and each triangular system solve about n^2 flops.
- It is always difficult to keep permutations and their inverses apart. In theoretical analysis, it is sometimes easier to work with permutation matrices than with the corresponding permutations.
- We defined the matrix P_σ; its jth column is 1 in row $\sigma(j)$, and 0 everywhere else.
- An important connection between a permutation σ and the matrix P_σ is given by $(P_\sigma\mathbf{x})_i = x_{\sigma^{-1}(i)}$.

Parallel LU decomposition

1 Programming using BSPlib

2 BSP benchmarking

3 The sequential LU decomposition

4 Parallel LU decomposition

Designing a parallel algorithm

- Main question: how to distribute the data?
- What data? The matrix A and the permutation π.
- Data distribution + sequential algorithm
  $\longrightarrow$ computation supersteps.
- Design backwards: insert preceding communication supersteps following the need-to-know principle.

Data distribution for the matrix A

- The bulk of the work in the sequential computation is the update

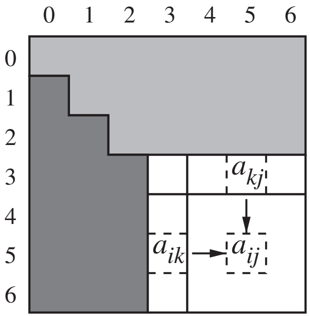
$$a_{ij} := a_{ij} - a_{ik}a_{kj}$$

  for matrix elements a_{ij} with $i, j \geq k+1$, taking $2(n-k-1)^2$ flops.
- The other operations take only $n-k-1$ flops. Thus, the data distribution is chosen mainly by considering the matrix update.

Data distribution for the matrix A

- The bulk of the work in the sequential computation is the update
$$a_{ij} := a_{ij} - a_{ik}a_{kj}$$
for matrix elements a_{ij} with $i, j \geq k+1$, taking $2(n-k-1)^2$ flops.
- The other operations take only $n-k-1$ flops. Thus, the data distribution is chosen mainly by considering the matrix update.
- Elements a_{ij}, a_{ik}, a_{kj} may not be on the same processor. Who does the update?
- Many elements a_{ij} must be updated in stage k, but only few elements a_{ik}, a_{kj} are used, all from column k or row k of the matrix. Moving those elements around causes less traffic.
- Therefore, the owner of a_{ij} computes the new value a_{ij} using communicated values of a_{ik}, a_{kj}.

Matrix update by operation $a_{ij} := a_{ij} - a_{ik}a_{kj}$



Update of row i uses only one value, a_{ik}, from column k. If we distribute row i over only N processors, then a_{ik} needs to be sent to at most $N - 1$ processors.

Matrix distribution

- A matrix distribution is a mapping

$$\phi : \ \{(i,j) : 0 \leq i,j < n\} \rightarrow \{(s,t) : 0 \leq s < M \wedge 0 \leq t < N\}$$

from the set of matrix index pairs to the set of processor identifiers. The mapping function ϕ has two coordinates,

$$\phi(i,j) = (\phi_0(i,j), \phi_1(i,j)).$$

Matrix distribution

- A matrix distribution is a mapping

$$\phi: \ \{(i,j) : 0 \leq i,j < n\} \rightarrow \{(s,t) : 0 \leq s < M \wedge 0 \leq t < N\}$$

  from the set of matrix index pairs to the set of processor identifiers. The mapping function ϕ has two coordinates,

$$\phi(i,j) = (\phi_0(i,j), \phi_1(i,j)).$$

- Here, we number the processors in 2D fashion, with $p = MN$. This is just a numbering.
- Processor numberings have no physical meaning. Assume BSPlib randomly renumbers the processors at the start!
- A processor row $P(s,*)$ is a group of N processors $P(s,t)$ with $0 \leq t < N$. A processor column $P(*,t)$ is a group of M processors $P(s,t)$ with $0 \leq s < M$.

Cartesian matrix distribution

$$\begin{array}{r|c|c|c|c|c|c|c|}
 & t=0 & 2 & 1 & 2 & 0 & 1 & 0 \\ \cline{2-8}
s=0 & \mathbf{00} & 02 & 01 & 02 & \mathbf{00} & 01 & \mathbf{00} \\ \cline{2-8}
0 & \mathbf{00} & 02 & 01 & 02 & 00 & 01 & 00 \\ \cline{2-8}
1 & 10 & 12 & 11 & 12 & 10 & 11 & 10 \\ \cline{2-8}
0 & 00 & 02 & 01 & 02 & 00 & 01 & 00 \\ \cline{2-8}
1 & 10 & 12 & 11 & 12 & 10 & 11 & 10 \\ \cline{2-8}
0 & 00 & 02 & 01 & 02 & 00 & 01 & 00 \\ \cline{2-8}
1 & 10 & 12 & 11 & 12 & 10 & 11 & 10 \\ \cline{2-8}
\end{array}$$

A matrix distribution is called Cartesian if $\phi_0(i,j)$ is independent of j and $\phi_1(i,j)$ is independent of i:

$$\phi(i,j) = (\phi_0(i), \phi_1(j)).$$

Parallel matrix update

```
(8)   **if** $\phi_0(k) = s \wedge \phi_1(k) = t$ **then** put $a_{kk}$ in $P(*, t)$;

(9)   **if** $\phi_1(k) = t$ **then for all** $i : k < i < n \wedge \phi_0(i) = s$ **do**
          $a_{ik} := a_{ik}/a_{kk}$;
```

Parallel matrix update

```
(8)  if $\phi_0(k) = s \wedge \phi_1(k) = t$ then put $a_{kk}$ in $P(*, t)$;

(9)  if $\phi_1(k) = t$ then for all $i : k < i < n \wedge \phi_0(i) = s$ do
         $a_{ik} := a_{ik}/a_{kk}$;

(10) if $\phi_1(k) = t$ then for all $i : k < i < n \wedge \phi_0(i) = s$ do
         put $a_{ik}$ in $P(s, *)$;
     if $\phi_0(k) = s$ then for all $j : k < j < n \wedge \phi_1(j) = t$ do
         put $a_{kj}$ in $P(*, t)$;

(11) for all $i : k < i < n \wedge \phi_0(i) = s$ do
         for all $j : k < j < n \wedge \phi_1(j) = t$ do
             $a_{ij} := a_{ij} - a_{ik}a_{kj}$;
```

Parallel pivot search

```
(0)  if $\phi_1(k) = t$ then $r_s := \mathrm{argmax}(|a_{ik}| : k \leq i < n \wedge \phi_0(i) = s)$;
(1)  if $\phi_1(k) = t$ then put $r_s$ and $a_{r_s,k}$ in $P(*, t)$;
```

Parallel pivot search

```
(0)   if $\phi_1(k) = t$ then $r_s := \mathrm{argmax}(|a_{ik}| : k \leq i < n \wedge \phi_0(i) = s)$;

(1)   if $\phi_1(k) = t$ then put $r_s$ and $a_{r_s,k}$ in $P(*, t)$;

(2)   if $\phi_1(k) = t$ then
            $s_{\max} := \mathrm{argmax}(|a_{r_q,k}| : 0 \leq q < M)$;
            $r := r_{s_{\max}}$;

(3)   if $\phi_1(k) = t$ then put $r$ in $P(s, *)$;
```

Two parallelisation methods

- The need-to-know principle: exactly those nonlocal data that are needed in a computation superstep should be fetched in preceding communication supersteps.
- Matrix update uses first parallelisation method: look at lhs (left-hand side) of assignment, owner computes.
- Pivot search uses second method: look at rhs of assignment, compute what can be done locally, reduce the number of data to be communicated.
- In pivot search: first a local search, then communication of the local winner to all processors, finally a redundant (replicated) search for the global winner.
- Broadcast of r in (3) is needed later in (4). Designing backwards, we formulate (4) first and then insert (3).

Distribution for permutation π

- Store π_k together with row k, somewhere in processor row $P(\phi_0(k), *)$.
- We choose $P(\phi_0(k), 0)$. This gives a true distribution.
- We could also have chosen to replicate π_k in processor row $P(\phi_0(k), *)$. This would save some **if**-statements in our programs.

Index and row swaps

```
(4)  **if** $\phi_0(k) = s \wedge t = 0$ **then** put $\pi_k$ as $\hat{\pi}_k$ in $P(\phi_0(r), 0)$;
     **if** $\phi_0(r) = s \wedge t = 0$ **then** put $\pi_r$ as $\hat{\pi}_r$ in $P(\phi_0(k), 0)$;

(5)  **if** $\phi_0(k) = s \wedge t = 0$ **then** $\pi_k := \hat{\pi}_r$;
     **if** $\phi_0(r) = s \wedge t = 0$ **then** $\pi_r := \hat{\pi}_k$;
```

Index and row swaps

```
(4)  if $\phi_0(k) = s \wedge t = 0$ then put $\pi_k$ as $\hat{\pi}_k$ in $P(\phi_0(r), 0)$;
     if $\phi_0(r) = s \wedge t = 0$ then put $\pi_r$ as $\hat{\pi}_r$ in $P(\phi_0(k), 0)$;

(5)  if $\phi_0(k) = s \wedge t = 0$ then $\pi_k := \hat{\pi}_r$;
     if $\phi_0(r) = s \wedge t = 0$ then $\pi_r := \hat{\pi}_k$;

(6)  if $\phi_0(k) = s$ then for all $j : 0 \leq j < n \wedge \phi_1(j) = t$ do
         put $a_{kj}$ as $\hat{a}_{kj}$ in $P(\phi_0(r), t)$;
     if $\phi_0(r) = s$ then for all $j : 0 \leq j < n \wedge \phi_1(j) = t$ do
         put $a_{rj}$ as $\hat{a}_{rj}$ in $P(\phi_0(k), t)$;

(7)  if $\phi_0(k) = s$ then for all $j : 0 \leq j < n \wedge \phi_1(j) = t$ do
         $a_{kj} := \hat{a}_{rj}$;
     if $\phi_0(r) = s$ then for all $j : 0 \leq j < n \wedge \phi_1(j) = t$ do
         $a_{rj} := \hat{a}_{kj}$;
```

Optimising the matrix distribution

- We have chosen a Cartesian matrix distribution ϕ to limit the communication.
- We now specify ϕ further to achieve a good computational load balance and to minimise the communication.
- Maximum number of local matrix rows with index $\geq k$:

$$R_k = \max_{0 \leq s < M} |\{i : k \leq i < n \wedge \phi_0(i) = s\}|.$$

Maximum number of local matrix columns with index $\geq k$:

$$C_k = \max_{0 \leq t < N} |\{j : k \leq j < n \wedge \phi_1(j) = t\}|.$$

- The computation cost of the largest superstep, the matrix update (11), is then $2R_{k+1}C_{k+1}$.

Example

	$t=0$	2	1	2	0	1	0
$s=0$	00	02	01	02	00	01	00
0	00	02	01	02	00	01	00
1	10	12	11	12	10	11	10
0	00	02	01	02	00	01	00
1	10	12	11	12	10	11	10
0	00	02	01	02	00	01	00
1	10	12	11	12	10	11	10

$$R_0 = 4, C_0 = 3 \text{ and } R_4 = 2, C_4 = 2$$

Bound for R_k

$$R_k \geq \left\lceil \frac{n-k}{M} \right\rceil .$$

Proof: Assume this is untrue, so that $R_k < \lceil \frac{n-k}{M} \rceil$. Because R_k is integer, we even have $R_k < \frac{n-k}{M}$. Hence all M processor rows together hold less than $M \cdot \frac{n-k}{M} = n-k$ matrix rows. But they hold all matrix rows $k \leq i < n$. Contradiction. $\square$

2D cyclic distribution attains bound

$$\begin{array}{c|c|c|c|c|c|c|c|}
 & t=0 & 1 & 2 & 0 & 1 & 2 & 0 \\ \hline
s=0 & 00 & 01 & 02 & 00 & 01 & 02 & 00 \\ \hline
1 & 10 & 11 & 12 & 10 & 11 & 12 & 10 \\ \hline
0 & 00 & 01 & 02 & 00 & 01 & 02 & 00 \\ \hline
1 & 10 & 11 & 12 & 10 & 11 & 12 & 10 \\ \hline
0 & 00 & 01 & 02 & 00 & 01 & 02 & 00 \\ \hline
1 & 10 & 11 & 12 & 10 & 11 & 12 & 10 \\ \hline
0 & 00 & 01 & 02 & 00 & 01 & 02 & 00 \\ \hline
\end{array}$$

$$\phi_0(i) = i \bmod M, \quad \phi_1(j) = j \bmod N.$$

$$R_k = \left\lceil \frac{n-k}{M} \right\rceil, \quad C_k = \left\lceil \frac{n-k}{N} \right\rceil.$$

Cost of main computation superstep (matrix update)

$$T_{(11),\mathrm{cyclic}} = 2\left\lceil \frac{n-k-1}{M} \right\rceil \left\lceil \frac{n-k-1}{N} \right\rceil \geq \frac{2(n-k-1)^2}{p}.$$

$$\begin{aligned} T_{(11),\mathrm{cyclic}} &< 2\left(\frac{n-k-1}{M}+1\right)\left(\frac{n-k-1}{N}+1\right) \\ &= \frac{2(n-k-1)^2}{p} + \frac{2(n-k-1)}{p}(M+N)+2. \end{aligned}$$

The upper bound is minimal for $M = N = \sqrt{p}$. The second-order term $4(n-k-1)/\sqrt{p}$ is the additional computation cost caused by load imbalance.

Cost of main communication superstep

The cost of the broadcast of row k and column k in (10) is

$$\begin{aligned} T_{(10)} &= (R_{k+1}(N-1) + C_{k+1}(M-1))g \\ &\geq \left(\left\lceil \frac{n-k-1}{M} \right\rceil (N-1) \;+\; \left\lceil \frac{n-k-1}{N} \right\rceil (M-1) \right) g \\ &= T_{(10),\mathrm{cyclic}}. \end{aligned}$$

$$\begin{aligned} T_{(10),\mathrm{cyclic}} &< \left(\left(\frac{n-k-1}{M} + 1 \right) N + \left(\frac{n-k-1}{N} + 1 \right) M \right) g \\ &= \left((n-k-1) \left(\frac{N}{M} + \frac{M}{N} \right) + M + N \right) g. \end{aligned}$$

The upper bound is again minimal for $M = N = \sqrt{p}$. The resulting communication cost is about $2(n-k-1)g$.

Summary

- We determined the matrix distribution, first by restricting it to be Cartesian, then by choosing the 2D cyclic distribution, based on a careful analysis of the main computation and communication supersteps, and finally by showing that a square $\sqrt{p} \times \sqrt{p}$ distribution is best.
- Developing the algorithm goes hand in hand with the cost analysis.
- We now have a correct algorithm and a good distribution, but the overall BSP cost may not be minimal yet. Wait and see . . .