

# Experiments with bsplu

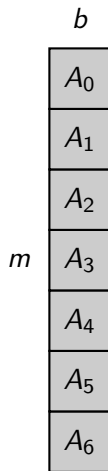
Sections 2.6–2.7 of Parallel Scientific Computation, 2nd edition

Rob H. Bisseling

Utrecht University



# Tall-and-skinny matrix



$m \times b$  matrix  $A$   
 $m \gg b$

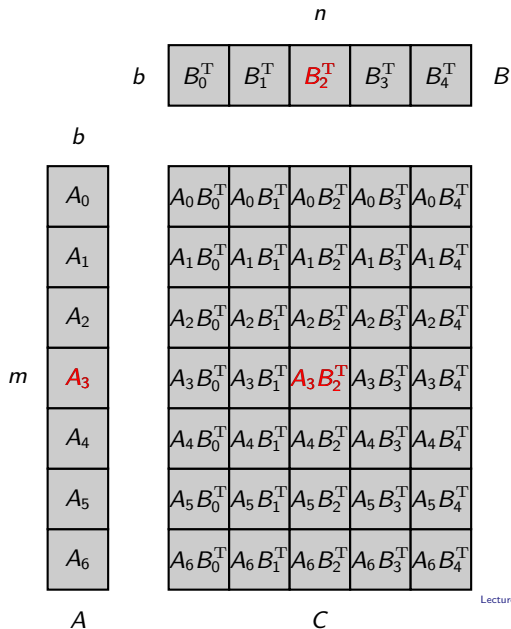


Burj Khalifa, Dubai (828 m)

Source: <https://www.burjkhalifa.ae>



Operation  $C := C - AB^T$  for tall-and-skinny  $A, B$



## Sequential tall-and-skinny matrix multiplication function

```
void matmat_tall_skinny(double **A, double **B,  
                        double **C,  
                        long m, long n, long b,  
                        long i0, long j0){
```

```
/* This function multiplies the  $m$  by  $b$  matrix  $A$   
   and the transpose of the  $n$  by  $b$  matrix  $B$   
   and subtracts the result  $A(B^T)$  from  
   the submatrix  $C(i0:i0+m-1, j0:j0+n-1)$ .
```

*The function is written in a cache-friendly way  
for tall-and-skinny matrices  $A$  and  $B$  ( $b \ll m, n$ ),  
using a block size  $b$ .*

```
*/
```

- ▶ Good habit: write the interface and the input/output specification of a function **before** you write its program text.
- ▶  $b \times b$  blocks fit into cache.



## Loops of the matrix multiplication function

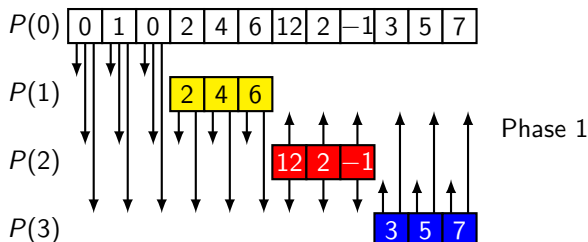
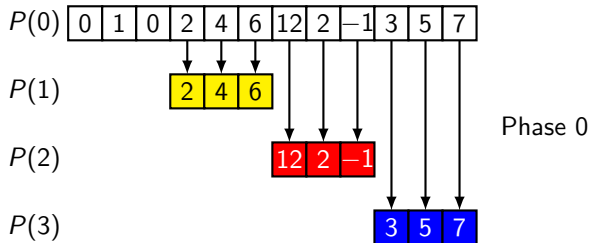
```
for (long ia=0; ia<m; ia+=b){
    long imax= MIN(ia+b,m);
    for (long jb=0; jb<n; jb+=b){
        long jmax= MIN(jb+b,n);

        // Multiply a block from A with a block from B
        for (long i=ia; i<imax; i++){
            for (long j=jb; j<jmax; j++){
                double sum= 0.0;
                for (long k=0; k<b; k++)
                    sum += A[i][k]*B[j][k];
                C[i0+i][j0+j] += sum;
            }
        }
    }
}
```

- ▶ 2 outer loops running over the blocks of  $A$  and  $B$ .
- ▶ 3 inner loops to multiply a block of  $A$  and a block of  $B$ .
- ▶  $A, B, C$  are accessed by row: good for row-wise storage.



## Two-phase broadcast in blocks



## Broadcast function

```
void bsp_broadcast(double *x, long n, long src, long s0,  
                  long stride, long p0, long phase){
```

```
/* Broadcast the vector x of length n from processor  
src to processors s0+t*stride, 0 <= t < p0.  
The vector x must have been registered previously.  
Processors are numbered in 1D fashion.
```

```
phase = phase of two-phase broadcast (0 or 1)  
Only one phase is performed, without synchronization.
```

```
*/
```

- ▶ Standard 1D-2D identification  $P(s, t) \equiv P(s + tM)$ .
- ▶ stride = 1, p0 = M: broadcast within processor column.  
stride = M, p0 = N: broadcast within processor row.
- ▶ No sync inside the function to allow combining supersteps.



## Main loop of the broadcast function

```
long s= bsp_pid(); // 1D processor number
long b= (n%p0==0 ? n/p0 : n/p0+1); // block size

if ((phase==0 && s==src) ||
    (phase==1 && s0 <= s && s < s0+p0*stride &&
     (s-s0)%stride==0)){

    /* Participate */
    ...
}
```

- ▶ In phase 0, **only  $P(0)$**  participates.
- ▶ In phase 1, **all destination processors** of phase 0.





## Loop for participating processor

```
for (long t=0; t<p0; t++){
    long dest= s0+t*stride;
    long t1 = (phase==0 ? t : (s-s0)/stride);

    long nbytes= MIN(b,n-t1*b)* sizeof(double);
    if (nbytes>0 && dest!=src)
        bsp_put(dest,&x[t1*b],x,
                t1*b*sizeof(double),nbytes);
}
```

- ▶ In phase 0, the block number `t1` of the [target for the `bsp\_put`](#) is simply `t`. In phase 1, it satisfies `s= s0+t1*stride`.
- ▶ The phases are similar, so we combined the program texts and saved a few lines of code.
- ▶ Still, the combined program text is easiest to understand if you [read](#) it for each phase [separately](#).



## Data are not sent back to the source

```
if ( nbytes > 0 && dest != src )  
    bsp_put ( dest, &x[t1*b], x, t1*b*sizeof(double), nbytes );
```

- ▶ This optimization does not affect the BSP cost, but it **reduces the communication volume**, which cannot be bad.



## Two-phase broadcast of row and column $k$

```
void bsp_broadcast(double *x, long n, long src, long s0,  
                 long stride, long p0, long phase);
```

```
...
```

```
/* Phase 0 of two-phase broadcasts */
```

```
if (k%N==t){  
    /* Store new column k in Lk */  
    for (long i=kr1; i<nr; i++)  
        Lk[i-kr1]= a[i][kc];  
}
```

```
...
```

```
bsp_broadcast(Lk, nr-kr1, s+(k%N)*M, s, M, N, 0);  
bsp_broadcast(Uk, k0cb-kc1, (k%M)+t*M, t*M, 1, M, 0);  
bsp_sync();
```

```
/* Phase 1 of two-phase broadcasts */
```

```
bsp_broadcast(Lk, nr-kr1, s+(k%N)*M, s, M, N, 1);  
bsp_broadcast(Uk, k0cb-kc1, (k%M)+t*M, t*M, 1, M, 1);  
bsp_sync();
```



# Local and global indices for cyclic distribution

Global

12	0	4	7	-1	2	15	11	3	-2
0	1	2	3	4	5	6	7	8	9

Local

12	-1	3
0	1	2

0	2	-2
0	1	2

4	15
0	1

7	11
0	1

Global index:  $i$

Local index on  $P(s)$ :  $i$

Relation:  $i = i \cdot p + s$

```
/* Initialize permutation vector pi */  
for (long i=0; i<nr; i++)  
    pi[i]= i*M+s; /* global row index */
```



## Putting data directly into a 2D array

```
a = matallocd(nr, nc); // in bsplu_test.c
```

```
void bsplu(long M, long N, long n, long *pi, double **a)
/* Set pointer for 1D access to A */
double *pa= NULL;
if (nr>0)
    pa= a[0];
bsp_push_reg(pa, nr*nc*sizeof(double));
...
/* Swap rows k and r for cols in range k0..k0+b-1 */
bsp_permute_rows(M, Src, Dest, nperm, pa, nc, k0c, k0cb);
...
}

/* Store row Src[i] of A in row r=Dest[i] on P(r%M,t) */
long r= Dest[i];
bsp_put(r%M+t*M,&pa[(Src[i]/M)*nc+jc], pa,
        ((r/M)*nc+jc)*sizeof(double),
        (jc1-jc)*sizeof(double));
```



## Rebroadcast of invalidated column elements

```
/* Obtain column k0 of L in range k0+1..k0+b-1 */
long k0r1= nloc(M,s,k0+1);
long k0rb= nloc(M,s,k0+b);
for (long i= k0r1; i<k0rb; i++){
    long tj= k0%N;
    long ncj= nloc(N,tj,n);
    bsp_get(s+tj*M,pa,(i*ncj+k0/N)*sizeof(double),
           &Lk[i-k0r1],sizeof(double));
}
bsp_sync();
```

- ▶ Some elements of  $L$  were **not valid any** more, because their permutation was postponed.
- ▶ The program text shows how to **get the correct values** for column  $k_0$  in rows  $k_0 + 1, \dots, k_0 + b - 1$  after the postponed permutations were carried out.



# Cori supercomputer at NERSC in Berkeley



Source: <https://www.nersc.gov>

- ▶ Supercomputer named after Nobel-prize winning biochemist **Gerty Cori** (1896–1957).
- ▶ Cray XC40 architecture consisting of 2388 **Intel Haswell nodes**, each with 32 cores running at 2.3 GHz, and 9688 **Intel Knights Landing nodes**, each with 68 cores running at 1.4 GHz.
- ▶ For two Haswell nodes, running BSPonMPI on top of Cray MPICH, the BSP parameters are  $p = 64$ ,  $r = 9.32$  Gflop/s,  $g = 1066$ ,  $l = 1\,842\,436$ .



## Broadcast time and total time $T$ (in s) of LU

$n$	One-phase		Two-phase		
	phase 0	$T$	phase 0	phase 1	$T$
1 000	0.36	<b>1.64</b>	0.23	0.23	1.75
2 000	0.75	<b>3.30</b>	0.49	0.51	3.57
3 000	1.33	<b>5.19</b>	0.77	0.80	5.53
4 000	2.09	<b>7.33</b>	1.07	1.12	7.53
5 000	2.99	9.78	1.40	1.46	<b>9.72</b>
6 000	4.02	12.53	1.75	1.82	<b>12.16</b>
7 000	5.20	15.60	2.11	2.20	<b>14.75</b>
8 000	6.53	18.81	2.53	2.64	<b>17.55</b>
9 000	8.00	22.38	2.98	3.34	<b>20.85</b>
10 000	9.64	26.47	3.44	4.42	<b>24.80</b>

- ▶  $p = 64$ ,  $b = 16$ ,  $1 \times 64$  cyclic distribution.
- ▶ This column distribution leads to **prominent column broadcasts**, and no row swaps or row broadcasts.
- ▶ Time of the phases of the two-phase broadcast is about **equal**, as predicted by BSP cost analysis.



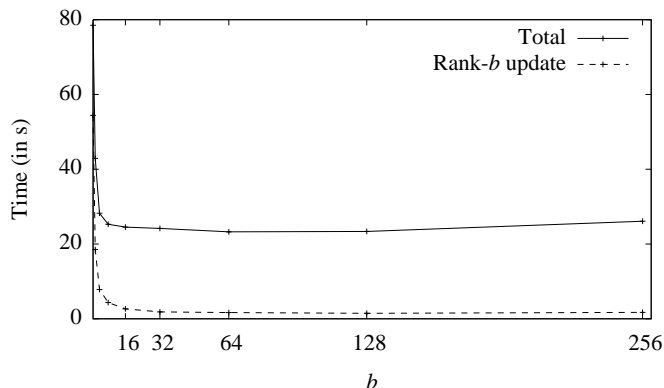


# Any actual savings by two-phase broadcast?

- ▶ **Small difference** in total time between one-phase and two-phase approach.
- ▶ For  $n \leq 4\,000$ , one-phase broadcast is better.
- ▶ For  $n > 4\,000$ , two-phase is better.
- ▶ The **savings are modest** compared to the total time: up to 9.4% for  $n = 10\,000$ .



# Time of LU as a function of algorithmic block size $b$



- ▶  $p = 64$ ,  $n = 10\,000$ ,  $8 \times 8$  cyclic distribution.
- ▶ Choice  $b = 64$  is optimal, but whole range  $b = 16$ – $256$  is fine.



## Breakdown of execution time of LU

Superstep	Operation	Time
(0)/(1)	local pivot search	69
(2)/(3)	global pivot search	71
(4)/(5)	row swap	51
(6)	phase 0 of column broadcast	72
(7)	phase 1 of column broadcast	189
(0')	instant matrix update	73
(8)	postponed row permutation	2
(9)	phase 0 of postponed row broadcast	49
(10)	phase 1 of postponed row broadcast	73
(9')	postponed rank-1 update	48
(Rb)	postponed rank- $b$ update	424
Total	LU decomposition	1122

- ▶  $p = 256$ ,  $n = 100\,000$ ,  $b = 64$ ,  $16 \times 16$  cyclic distribution.
- ▶ The time given is the total time (in s) of the superstep in the whole algorithm.
- ▶ The rank- $b$  update takes 37.8% of the total time, which indicates **reasonable efficiency**.



# Summary

- ▶ High performance in linear algebra computations can be achieved by formulating algorithms using **matrix multiplication**.
- ▶ Tall-and-skinny matrices can be multiplied **in cache** by splitting them into smaller square blocks.
- ▶ Broadcasts of large vectors were observed to be fastest when using the **two-phase approach**.
- ▶ A parallel algorithm is best described using **global indices**, but an actual parallel program should use **local indices**.
- ▶ Measuring the time of separate supersteps is a way of getting **intimate knowledge** of your program.

