# A BSP matrix multiplication implementation

The BSP algorithm for matrix multiplication presented below was presented in the seminal work of [1]. It works for $p \leq n^2$. Each processor is assigned the task of computing an $n/\sqrt{p} \times n/\sqrt{p}$ submatrix of the product $A \times B$. The input matrices $A$ and $B$ are divided into $p$ block-submatrices, each one of dimension $m \times m$, where $m = n/\sqrt{p}$. We call this distribution of the input among the processors *block* distribution. This way, element $A(i,j)$, $0 \leq i < n, 0 \leq j < n$, belongs to the $(j/m) * \sqrt{p} + (i/m)$-th block that is subsequently assigned to the memory of the same-numbered processor. Let $A_i$ (respectively, $B_i$) denote the $i$-th block of $A$ (respectively, $B$) stored in processor $i$. With these conventions the algorithm in [1] can is described in Figure 1. The following Proposition describes the performance of the aforementioned algorithm.

**begin** MULT_A ($C$,$A$,$B$,$n$,$p$)
1. Let $m = n/\sqrt{p}$ ;
   Each processor is also assigned a unique processor number $q$;
2. Let $p_i = q \bmod \sqrt{p}$ ; $p_j = q/\sqrt{p}$ ; $C_q = 0$;
3. $a_l \leftarrow A_{p_i + l * \sqrt{p}}$, $0 \leq l < \sqrt{p}$;
4. $b_l \leftarrow B_{p_j * \sqrt{p} + l}$, $0 \leq l < \sqrt{p}$;
5. **for** $0 \leq l < \sqrt{p}$ **do**
       $C_q = C_q + a_l \times b_l$;

**end** MULT_A

Figure 1: Procedure MULT_A.

**Proposition 1** *Algorithm* MULT_A *for multiplying two $n \times n$ matrices $A$ and $B$ stored according to the block distribution requires, for any $p \leq n^2$, computation time $C_{mul}(n)$ that is given by*

$$C_{mul}(n) = \max\{L, \frac{(2n-1)n^2}{p}\},$$

*and communication time $M_{mul}(n)$ that is given by the expression*

$$M_{mul}(n) = \max\{L, g\frac{2n^2}{\sqrt{p}}\}.$$

One immediately realizes that algorithm MULT_A is not memory efficient since it requires more local memory per processor – by a factor of $\sqrt{p}$ – than the required one. Algorithm MULT_B shown in Figure 2 is the memory efficient variant of MULT_A. It is not synchronization efficient though since its number of supersteps is not constant any more; it has been increased by a factor of $\sqrt{p}$. The performance of algorithm MULT_B is summarized in Proposition 2.

**begin** MULT_B ($C$,$A$,$B$,$n$,$p$)
1. Let $m = n/\sqrt{p}$ ;
   Each processor is also assigned a unique processor number $q$;
2. Let $p_i = q \bmod \sqrt{p}$ ; $p_j = q/\sqrt{p}$ ; $C_q = 0$;
3. **for** $0 \leq l < \sqrt{p}$ **do**
       **begin**
4.     $a \leftarrow A_{((p_i+p_j+l) \bmod \sqrt{p}) * \sqrt{p} + p_i}$;
5.     $b \leftarrow B_{((p_i+p_j+l) \bmod \sqrt{p}) + p_j * \sqrt{p}}$;
6.     $C_q = C_q + a \times b$;
       **end**

**end** MULT_B

Figure 2: Procedure MULT_B.

**Proposition 2** *Algorithm* MULT_B *for multiplying two $n \times n$ matrices $A$ and $B$ stored according to the block distribution requires, for any $p \leq n^2$, computation time $C_{mul}(n)$ that is given by*

$$C_{mul}(n) = \sqrt{p}\ \max\{L, \frac{(2n-1)n^2}{p^{3/2}}\}$$

*and communication time $M_{mul}(n)$ that is given by the expression*

$$M_{mul}(n) = \sqrt{p}\ \max\{L, g\frac{2n^2}{p}\}$$

In order to show the efficiency of algorithm design on the BSP model we present some experimental results for matrix multiplication on Cray T3D; additional results can be found in the author's Web page. Algorithm MULTT_B is a variation of MULT_B where in order to multiply $A$ with $B$, matrix $A$ is first transposed and the loop for matrix multiplication is changed accordingly. This way the access patterns for both $A$ and $B$ are the same (column - column as opposed to row - column) thus improving locality (cache usage), and subsequently program performance.

| *Algorithm* MULT_B | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | $p=1$ | | $p=4$ | | $p=16$ | | $p=64$ | |
| $n$ | Time (sec) | Mfl rate | Time (sec) | Mfl rate | Time (sec) | Mfl rate | Time (sec) | Mfl rate |
| 256 | 4.1 | 7.9 | 1.1 | 7.8 | 0.28 | 7.4 | 0.03 | 13.9 |
| 512 | 34.0 | 7.8 | 8.4 | 7.9 | 2.1 | 7.7 | 0.56 | 7.4 |
| 1024 | 289.8 | 7.4 | 68.4 | 7.8 | 16.9 | 7.9 | 4.3 | 7.7 |
| 2048 | - | - | - | - | 136.8 | 7.8 | 33.8 | 7.9 |

Table 1: Execution time for MULT_B on the Cray T3D

| *Algorithm* MULTT_B | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | $p=1$ | | $p=4$ | | $p=16$ | | $p=64$ | |
| $n$ | Time (sec) | Mfl rate | Time (sec) | Mfl rate | Time (sec) | Mfl rate | Time (sec) | Mfl rate |
| 256 | 2.3 | 14.3 | 0.58 | 14.4 | 0.15 | 13.7 | 0.03 | 15.1 |
| 512 | 20.7 | 12.9 | 4.7 | 14.1 | 1.16 | 14.4 | 0.30 | 13.5 |
| 1024 | 202.7 | 10.5 | 41.7 | 12.8 | 9.4 | 14.1 | 2.3 | 14.3 |
| 2048 | - | - | - | - | 83.5 | 12.8 | 19.0 | 14.1 |

Table 2: Execution time for MULTT_B on the Cray T3D

Finally, we outline a matrix multiplication algorithm that is computation, communication and synchronization efficient. It fails, however, to be memory efficient, as its memory requirements are a multiplicative factor $p^{1/3}$ from the optimal. Algorithm MULTT_C is outlined in the remainder of this section.

In MULTT_C matrices $A$ and $B$ (and the result $C$) are split into two ways into submatrices. Each matrix ($A$, $B$ and the result $C$) is split into $p$ "physical" block-submatrices, as in the previous algorithms, each of size $n/p^{1/2} \times n/p^{1/2}$. A "physical" block-submatrix indicates the part of the matrix stored in a single physical (processor) location (i.e. block-submatrix $A_i$ is stored in processor $i$). At the same time, each of the three matrices is split into $p^{2/3}$ "virtual" block-submatrices each of size $n/p^{1/3} \times n/p^{1/3}$. A "virtual" block-submatrix indicates the block geometry that will be used in the matrix multiplication algorithm to be outlined below. The elements of a "virtual" block-submatrix may be stored in more than one physical processors.

Whereas in the first two algorithms "physical" and "virtual" block-submatrices coincided in number and dimension, in this communication efficient algorithm are clearly distinguished.

Let the "virtual" block-submatrices be identified as $A_{ij}$, $B_{ij}$ and $C_{ij}$. Matrix multiplication will thus require the computation of all $C_{ij} = \sum_{k=1}^{p^{1/3}} C_{ijk} = \sum_{k=1}^{p^{1/3}} A_{ik}B_{kj}$, where $C_{ijk} = A_{ik}B_{kj}$.

The algorithm consists of the following steps. We name the processors $(i,j,k)$ the way we did in the matrix multiplication algorithm on the hypercubic networks.

**Step 1.** Processor $(i,j,k)$ gets $A_{ik}$ and $B_{kj}$. Note that each of these two "virtual" block-submatrices may originate from more than one processors. Each processor sends at most $2n^2/p$ elements (but each one replicated $p^{1/3}$ times) and receives at most $2n^2/p^{2/3}$ elements. The communication cost of Step 1 is $\max\{L, 2gn^2/p^{2/3}\}$. Subsequently, the two submatrices are multiplied as in the sequential case a step requiring at most $\max\{L, 2n^3/p\}$ time. Partial-submatrix $C_{ijk}$ is thus computed on processor $(i, j, k)$. Each element of such a submatrix is a partial sum of an element $c_{lm}$ of the result matrix $C$.

**Step 2.** Each element of $C_{ijk}$ is transmitted from $(i, j, k)$ to that physical processor that stores the "physical" block-submatrix of $C$ whose elements will be formed as sums of the receiving elements (partial sums) of $C_{ijk}$. Note that each $(i, j, k)$ processor may send its elements to more than one physical processors. At the completion of this step, each of the $p$ processors storing a block-submatrix of $C$ of dimension $n/p^{1/2} \times n/p^{1/2}$ receives at most $p^{1/3} \cdot n^2/p$ such elements (partial sums). The complex communication performed in this step requires time $\max\{L, gn^2/p^{2/3}\}$.

**Step 3.** The received partial sums are added. $p^{1/3}$ partial sums are summed to give an element of $C$ stored at a physical processor, for a total of $n^2/p$ such elements (of a "physical" block-submatrix). The total computation time performed is $\max\{L, n^2/p^{2/3}\}$.

**Proposition 3** *Algorithm* MULT_C *for multiplying two $n \times n$ matrices $A$ and $B$ stored according to the block distribution requires, for any $p \leq n^2$, computation time $C_{mul}(n)$ that is given by*

$$C_{mul}(n) \leq \max\{L, 2n^3/p\} + \max\{L, n^2/p^{2/3}\},$$

*and communication time $M_{mul}(n)$ that is given by the expression*

$$M_{mul}(n) = \max\{L, 2g\frac{n^2}{p^{2/3}}\} + \max\{L, g\frac{n^2}{p^{2/3}}\}.$$

The optimality in communication of the algorithm is established by the following result.

**Theorem 1** *On a model of computation that allows the operations $\{+, *\}$ only, if any processor reads $s$ elements of $A$ and $B$ and computes at most $s$ partial sums of $C$, it can compute at most $O(s^{3/2})$ multiplicative terms for these sums.*

This way, if a processor reads at most $s$ elements of $A$ and $B$ it can compute at most $O(s^{3/2})$ multiplicative terms of $C$. Combined, all $p$ processors can compute $p\, O(s^{3/2})$ such terms which must be $\Omega(n^3)$. Therefore $s = \Omega(n^2/p^{2/3})$ and thus algorithm MULT_C is communication optimal.

# 1 A BSP program in ANSI-C

Below, the source code for matrix multiplication algorithm MULTT_B tested is given. Initially, two global $n \times n$ matrices $A_g$ and $B_g$ are distributed among $p$ processors. The $p$ processors are divided into $\sqrt{p}$ groups of $\sqrt{p}$ processors each. Element $(i, j)$ of $A_g$ or $B_g$ is stored in the $i/\sqrt{p}$-th processor of the $j/\sqrt{p}$ processor group, that is, processor $(j/\sqrt{p}) * \sqrt{p} + i/\sqrt{p}$.

Function `_multiply_par` requires five arguments, the two input matrices $A$ and $B$ which contain the $n^2/p$ elements of $A_g$ and $B_g$ that are local to a particular processor, the result $C$ that will hold $n^2/p$ elements of the product $C_g$, the dimension $n$, and $\sqrt{p}$ (which must be an integer).

As we have previously mentioned, $A$, $B$ and $C$ are ANSI-C pointers to a `double` data type. We store matrices in the form of an on dimensional array. This way, element $(i, j)$ of a two-dimension matrix is stored in position $j * n + i$. All indices are in the range $0, \ldots, n-1$.

The BSPlib allows for SPMD (Single Program Multiple Data) programming. The program that calls `_multiply_par` spawns $p$ processes each executing the same code. Each such process is assigned a unique identifier that can be accessed by `bsp_pid()`. The number of available processes is available through a call to `bsp_nprocs()`. Among the variables used, `nprocs` and `pid` hold the number of available processors $p$ and the processor identifier of a process.

```
1.  void _multiply_par(double *A,double *B,double *C,int n,int p_sqrt )
2.  {
3.   register int    nprocs=bsp_nprocs(); /* # of processors */
4.   register int    pid=bsp_pid();       /* processor identifier */
```

Variables $a$ and $b$ will hold local copies of $A$ and $B$ that will be fetched during the course of the algorithm (lines 4 and 5 of Figure 2). Variable `ni` holds $n/\sqrt{p}$ and variables `ppi` and `ppj` hold respectively, the index of processor `pid` within processor group `ppj`, and the processor group to which processor `pid` belongs.

```
5.   double          *a,*b;                /* local storage  */
6.   register double tmp;                  /* temporary variable */
7.   register int    i,j,k,l;              /* indices */
8.   register int    t1,t2,                /* temporary indices */
9.                   ppi,ppj,              /* index and processor group*/
                                           /* for processor  pid */
10.                  ni;                   /* n/sqrt(p) */
```

In the following code segment `ni`, `ppi` and `ppj` are computed.

```
11.  ni=n/p_sqrt;
12.  ppi=pid % p_sqrt; /*index of pid within its (column block) group */
13.  ppj=pid / p_sqrt; /* (column-block) group pid belongs to */
```

In lines 14-18, allocation of memory space for `a` and `b` is performed. Function `check_if_null()` checks whether a returned pointer by `malloc()` is empty or not and in the former case prints an informative diagnostic message.

```
14.  /* Allocation and checking */
15.  a = (double *)malloc(ni*ni*sizeof(double));
16.  check_if_null((void *)a,"in multiply_par","a");
17.  b = (double *)malloc(ni*ni*sizeof(double));
18.  check_if_null((void *)b,"in multiply_par","b");
```

A registration of variables that hold data to be communicated is required in BSPlib. This is performed in lines 19-21.

```
     /* Registration */
19.  bsp_push_reg(A,ni*ni*sizeof(double));
20.  bsp_push_reg(B,ni*ni*sizeof(double));
21.  bsp_sync();
```

An initialization of the result matrix $C$ is performed in lines 22-24.

```
     /* Initialization */
22.  for (j=0;j<ni;j++)
23.    for (i=0,t1=j*ni;i<ni;i++)
24.      C[t1 + i]= 0.0;
```

The number of communication rounds (supersteps) for this matrix multiplication routine is $\sqrt{p}$; line 25 initiates the loop of line 3 or Figure 2. In round $l$ the required submatrices stored in processor $((p_i + p_j + l) \bmod \sqrt{p}) * \sqrt{p} + p_i$ (for $A$) and $((p_i + p_j + l) \bmod \sqrt{p}) + p_j * \sqrt{p}$ (for $B$) are fetched. These operations correspond to lines 4 and 5 of Figure 2. The syntax of `bsp_get(processor, source, offset, destination, size)` is as follows: `processor` identifies the processor from whom data will be obtained, `source` is a pointer to the structure that will provide the data from `processor`, `offset` is an offset in bytes of the first address of the data to be transferred from `source`, `destination` is the destination address in `pid` of the data that will be transferred and `size` is the size of transferred data in bytes.

```
25. for (l=0;l<p_sqrt;l++) {
     /* Efficient non-conflicting communication */
     /* A 2n^2/p relation is communicated */
26.  bsp_get(((ppi+ppj+l)%p_sqrt)*p_sqrt+ppi,A,0,a,ni*ni*sizeof(double));
27.  bsp_get(((ppi+ppj+l)%p_sqrt)+ppj*p_sqrt,B,0,b,ni*ni*sizeof(double));
28.  bsp_sync();
```

In lines 29-34 array $a$ is transposed so that the following multiplication loop be performed more efficiently.

```
      /* Transpose a */
29.    for (i=0,t2=0;i<ni;i++,t2+=ni)
30.       for (j=i,t1=i*ni;j<ni;j++,t1+=ni)  {
31.         tmp=a[t2+j];
32.         a[t2+j]=a[t1+i];
33.         a[t1+i]=tmp;
34.       }
```

Lines 35-44 perform the local multiplication of line 6 of Figure 2.

```
      /* Multiplication */
35.      for (j=0;j<ni;j++) {
36.        t1=j*ni;
37.        for (i=0;i<ni;i++) {
38.            t2=i*ni;tmp=0.0;
39.            for (k=0;k<ni;k++)
40.              tmp += (b[t1+k]*a[t2+k]);
41.            C[t1+i] += tmp;
42.          }
43.      }
44.    } /* end of (loop) l-th superstep */
```

A de-registration of variables is performed, followed by a deallocation of variables.

```
45.  bsp_pop_reg(B);
46.  bsp_pop_reg(A);

47.  free((void *)b);
48.  free((void *)a);
}
```

## References

[1] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103-111, August 1990.