Matrix multiplication in BSP

Name: Adriano
Surname: Cardace
Date: 03/25/2017

Abstract:

This report explains two different parallel algorithm for matrix multiplication of size n, both implemented with BSP. The first solution uses a rows distribution while the second one is an implementation of Cannon's algorithm.

Table of contents:

1. Introduction

The purpose of this assignment is to implement two different parallel algorithm for computing C = A x B where A,B,C are n x n real matrices.
The first algorithm uses a rows distribution where processor i owns rows n/p * i up to row n/p * (i+1) of matrix A and C, while the sets of rows in matrix B are shifted cyclically between the processors in order to avoid waste of memory and useless communication.
The second algorithm is an implementation of Cannon's algorithm, which uses a block distribution.
The presented solutions are written in C and parallelized using BSP; every running test is done at least three times on brake.ii.uib.no.
In order to get the best running time the source files are compiled with GCC using the -O3 flag, furthermore the initial time spent on memory allocation and on generating the random values is not included; the elements of A and B on each processor are initialized on the processor, so we can avoid to distribute them from one processor to the others.
We made the assumption that the number of processor p evenly divides n.
To make sure that the computation is correct, the user can provide coordinates y,x for the matrix C, this element is explicitly computed and compared with the element obtained in position y,x with the parallel algorithm.
The source file are "matrixMul.c" and "cannon.c" for the first and the second algorithm respectively.

2. Matrix multiplication with rows distribution

2.1 Step of the algorithm

a) Initialization
b)Compute partial elements of C
c)Get the set of rows owned by the next processors(repeat steps b and c, p times)
d)Print the requested value
e)Compute explicitly the requested value as a proof of correctness

2.2 Main variables

p: number of processors
s: processor id
A, B, C: local matrices
pA, pB, pC: pointers to the rows of A, B, C matrices
N: matrices dimension
K: is equal to N/p and it represents the number of rows held by each processor
x: column of the element to check, this value is given by the user
y: row of the element to check, this value is given by the user

2.3 algorithm

a)initialization

Each processor allocate the necessary contiguous memory to hold *N*k* elements as an array and fills it up with random values.

```
A = (double *) malloc (N*k*sizeof(double));
B = (double *) malloc (N*k*sizeof(double));
C = (double *) malloc (N*k*sizeof(double));

pA = (double **) malloc(k*sizeof(double *));
pB = (double **) malloc(k*sizeof(double *));
pC = (double **) malloc(k*sizeof(double *));

for(i=0; i<k; i++){
   pA[i]=A+i*N;
   pB[i]=B+i*N;
   pC[i]=C+i*N;
}

gettimeofday(&t1, NULL);

srand(t1.tv_usec * t1.tv_sec * s);
for(i=0; i<k; i++)
   for(j=0; j<N; j++)
      pA[i][j] = (double)rand()/(double)(RAND_MAX);

for(i=0; i<k; i++)
   for(j=0; j<N; j++){
      pB[i][j] = (double)rand()/(double)(RAND_MAX);
      pC[i][j] = 0.0;
   }
```

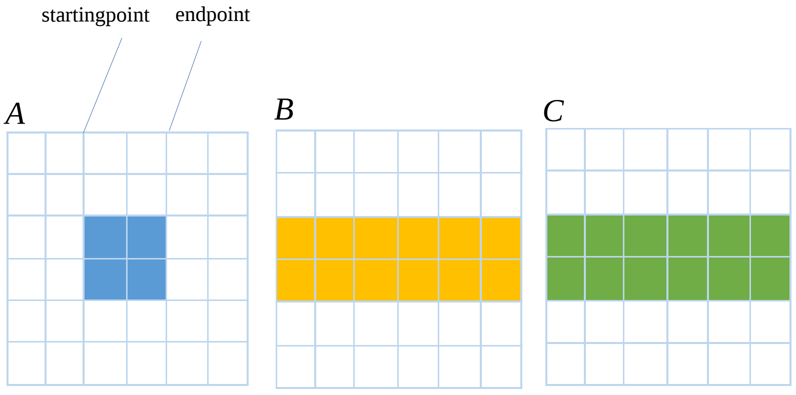
b)Compute partial elements of C

The idea is that to compute the element C[x][y] we have to compute ∑A[x][i]*B[i][y], but the order of the operands is irrelevant.
In addition to this, processor i holds only a portion of column y of B(precisely *N/p* elements), therefore there are *p* supeststeps, in each of these, processor i computes the partial sum for the elements in the rows which is responsible. After that, it receives the next *N/p* rows of the matrix B from the next processor and updates the same element in C.
In this way every processor is able to see the whole matrix B, but in each superstep only *N/p* rows are visible.

The first loop goes from 0 up to p, so that every processor is able to see the whole matrix B.
The second loop runs from0 up to *k*, since every processor is responsible for *N/p=k* rows.
Now that the row is fixed ,we have to multiply the right elements of A with the corresponding elements in B and update all the selected row in C; after that, the chunk of rows are shifted cyclically and we update the starting point end the endpoint for the next iteration.

```
bsp_push_reg(B, N*k*sizeof(double));
bsp_sync();

double time0= bsp_time();

int l;
int startingpoint = s*k;
int endpoint = startingpoint + k;
int row;
int superstep;
```

```
for(superstep = 0; superstep<p; superstep++){
    for (row=0; row<k ; row++)
        for (j=startingpoint, l=0; j<endpoint; j++, l++)
            for (g=0; g<N; g++)
                pC[row][g] += pA[row][j] * pB[l][g];
    bsp_get((s+1)%p, B, 0, B, N*k*sizeof(double) );
    bsp_sync();
    startingpoint = (startingpoint+k)%N;
    endpoint = startingpoint +k ;
}

 double time1= bsp_time();
```



d)Print the requested value

Now that the C matrix is computed, the processor with *s=y/k* holds the element C[y][x] requested by the user, so it is responsible to print out the value.
Since in each processor local rows start from 0 up to k, we have to select the *y%k* row to pick up the right global row given in input.

```
if (s==y/k){
    printf("---- C[%ld][%ld]=%.2f from row-wise decomposition-----\n", y, x, pC[y%k][x]);
    printf("Totale time : %f\n", time1-time0);
}
```

Now we compute explicitly the value of C[y][x] as a proof of correctness.
Every processor puts his k elements of the column x in B in the right position of bcolumn, so that *s=y/k* can explicitly compute and print C[y][x].

```
double *bcolumn;
bcolumn = (double *) malloc (N*sizeof(double));
bsp_push_reg(bcolumn, N*sizeof(double));
bsp_sync();

for(i=0; i<k; i++){
    bsp_put(y/k, &(pB[i][x]), bcolumn, (s*k+i)*sizeof(double) , sizeof(double) );
    bsp_sync();
}
if(s==y/k){
    double sum=0.0;
    for(i=0; i<N; i++)
        sum += pA[y%k][i]*bcolumn[i];
    printf("---- C[%ld][%ld]=%.2f from test -----\n", y, x, sum);

}
```

3. Cannon’s algorithm

3.1 Step of the algorithm

a)Initialization
b)Test
c)Shifting matrices
d)Compute C

3.2 Main variables

p: number of processors
s: processor id

A, B, C: local matrices
pA, pB, pC: pointer to the rows of *A, B, C* local matrices
N: matrices dimension
K: is equal to *N/p* and it represents the number of row held by each processor
x: column of the element to check, this value is given by the user
y: row of the element to check, this value is given by the user
pside: is the number of processors in each side of the matrices.
row, column: these variables contain the row and column of a processor in the grid.
id: is the processor responsible to print out the value requested by user.
prow, pcomun: this are the values of the row and the columns necessary to compute explicitly C[y][x]; every processor that has these values must participate in the final check.

3.3 algorithm

a)Initialization

```
   A = (double *) malloc (k*k*sizeof(double));
   B = (double *) malloc (k*k*sizeof(double));
   C = (double *) malloc (k*k*sizeof(double));

   pA = (double **) malloc(k*sizeof(double *));
   pB = (double **) malloc(k*sizeof(double *));
   pC = (double **) malloc(k*sizeof(double *));

//row  and column of the processor in the grid
   row = s/pside;
   column = s%pside;

   long prow = y/k;
   long pcol = x/k;
   long id = prow*pside+pcol;

   bsp_push_reg(A, k*k*sizeof(double));
   bsp_push_reg(B, k*k*sizeof(double));
   bsp_push_reg(C, k*k*sizeof(double));
   bsp_sync();

   for(i=0; i<k; i++){
      pA[i]=A+i*k;
      pB[i]=B+i*k;
      pC[i]=C+i*k;
   }

   gettimeofday(&t1, NULL);

   srand(t1.tv_usec * t1.tv_sec * s);
   for(i=0; i<k; i++)
      for(j=0; j<k; j++)
         pA[i][j] = (double)rand()/(double)(RAND_MAX);


 for(i=0; i<k; i++)
      for(j=0; j<k; j++){
         pB[i][j] = (double)rand()/(double)(RAND_MAX);
         pC[i][j] = 0.0;
      }
```

b)Test

For the test every processor that holds parts of the x column or parts of the y row is responsible to send their values to processor 0, so that it cans compute explicitly C[y][x].
Note that we have to put local elements from local row y%k and from local column x%k.

```
double *arow;
double *bcolumn;
arow = (double *) malloc(N*sizeof(double));
bcolumn = (double *) malloc(N*sizeof(double *));
bsp_push_reg(arow,N*sizeof(double *));
bsp_push_reg(bcolumn,N*sizeof(double *));
bsp_sync();

if(row == prow){
        bsp_put(0, pA[y%k] , arow, k*column*sizeof(double), k*sizeof(double));
}
bsp_sync();

if(column == pcol){
    for(i=0; i<k; i++){
        bsp_put(0, &pB[i][x%k] , bcolumn, (k*row+i)*sizeof(double) , sizeof(double));
    }
}
bsp_sync();

if(s==0){
     double sum=0.0;
    for(i=0; i<N; i++)
        sum += arow[i]*bcolumn[i];
    printf("---- C[%ld][%ld]=%.2f from test -----\n", y, x, sum);

}
bsp_pop_reg(arow);
bsp_pop_reg(bcolumn);
free(arow);
free(bcolumn);
```

c)Shifting matrices

This *for* loop shits blocks of matrices A and B according to Cannon’s algorithm:
if i is the last processor in his row, i has to get matrix A from *s-pside+1*, otherwise from *s+1*.
Similarly if *i* is the last on his column, *i* has to get the B matrix from *(s+pside)%pside*, otherwise from the processor below with id equals to *s+side*.

```
for (i=0; i< pside; i++){
    if (row > 0 && i<row){
        if (s%pside == pside-1){
            //i'm the last on my row
            bsp_get(s-pside+1, A, 0, A, k*k*sizeof(double));
        }
        else{
            bsp_get(s+1, A, 0, A, k*k*sizeof(double));
        }
    }
    bsp_sync();
}

for (i=0; i< pside; i++){
    if ( column>0 && i<column){
        if (row == pside-1){
            //i'm the last on my column
            bsp_get((s+pside)%pside, B, 0, B, k*k*sizeof(double));
        }
        else{
            bsp_get(s+pside, B, 0, B, k*k*sizeof(double));
        }
    }
```

```
        bsp_sync();
    }
```

d)Compute C

We run this for loop *pside* times, since this is the number of processor per side, and processor *i* updates his own matrix C calling localMatUpdate().
After that, processor *i* get the block of matrix A and B according to the Cannon’s algorithm.

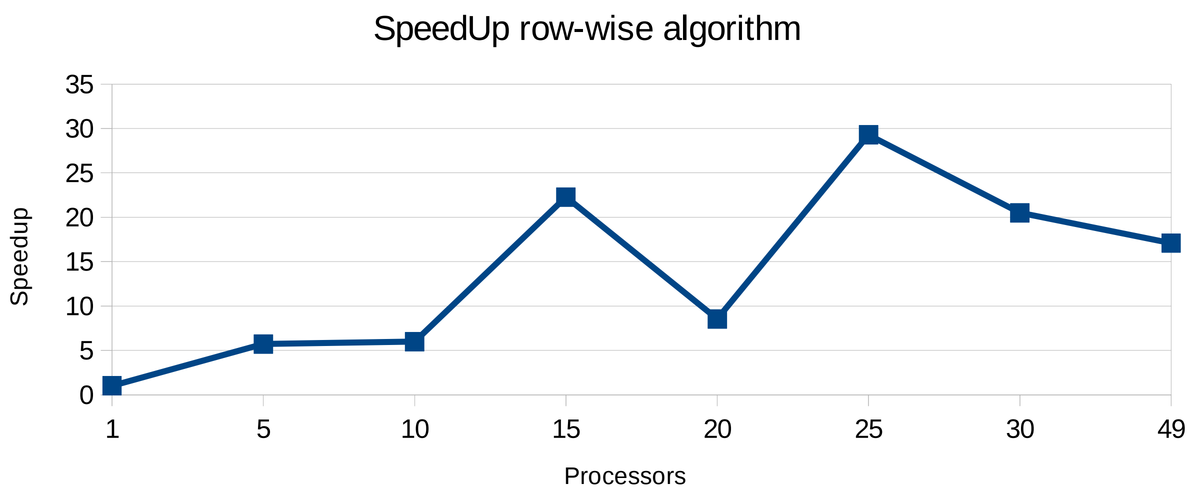
```
   double start = bsp_time();
    for(i=0; i<pside; i++){
        localMatUpdate(pA, pB, pC, k);

        //get matrix A from right
        if (s%pside == pside-1){
            //i'm the last on my row
            bsp_get(s-pside+1, A, 0, A, k*k*sizeof(double));
        }
        else{
            bsp_get(s+1, A, 0, A, k*k*sizeof(double));
        }
        //get matrix B from down
        if (row == pside-1){
                //i'm the last on my column
                bsp_get((s+pside)%pside, B, 0, B, k*k*sizeof(double));
            }
            else{
                bsp_get(s+pside, B, 0, B, k*k*sizeof(double));
            }
        bsp_sync();
    }

    double finish = bsp_time();
```

At the end of the previous loop the responsible processor print out C[y][x].
Note that *id=prow*pside+pcol*.

```
    if (s==id){
        printf("element C[%ld][%ld]=%.2f in %f\n", y, x, pC[y%k][x%k], finish-start);
    }
```

4. Strong scaling

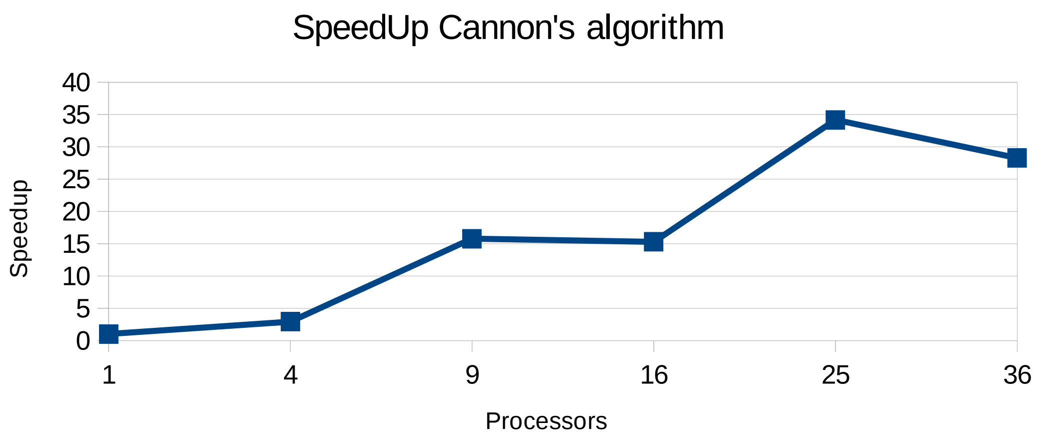
The following chart shows the speedup with matrices of size n=1800.



Here there is another representation of the same data where we include also the running time for the communication and the computation parts.

Processors	1	5	10	15	20	25	30	40
Sequential	10.25	10.25	10.25	10.25	10.25	10.25	10.25	10.25
Parallel	10.11	1.79	1.71	0.46	1.20	0.35	0.50	0.60
Communication time	0.04	0.18	0.40	0.08	0.49	0.11	0.14	0.3
Computation time	10.09	1.61	1.21	0.38	0.71	0.24	0.36	0.30
Speedup	1.01	5.72	5.99	22.28	8.54	29.28	20.5	17.08

Oddly, both communication and computation time are high with 10 and 20 processors,.



Here there is a table representation of the same data.

Processors	1	4	9	16	25	36
Sequential	10.25	10.25	10.25	10.25	10.25	10.25
Parallel	10.22	3.49	0.65	0.67	0.30	0.36
Speedup	1.00	2.93	15.77	15.29	34.17	28.28

5. Running time analysis

The cost of a BSP algorithm is an expression of the form $a+bg+cl$, where a is the number of flops overall the supersteps, b determines the communication time, and finally c is the number of supersteps.
g and l are function of the number of processors p , the former represents the necessary flops to send one data word, the latter is the cost of the synchronization.
r is the computing rate and it depends also from the machine.
In the first algorithm we have the following values:

$a=\frac{n^3}{p}$, since there are p steps, where for each of the $\frac{n}{p}$ rows, we have $\frac{n^2}{p}$ computations.

$b=n^2$, since there are p communications steps of lengths $\frac{n^2}{p}$.

$c=p$, it takes p step to see the whole B matrix, because every processor holds $\frac{n^2}{p}$ elements.

g and l are computed with a benchmarking program for BSP executed on brake.ii.uib.no; with p fixed to 30, they are respectively 29.2 and 39825.7 flop.

The computing rate r is equal to 529.225 Mflop/s.

The final formula is $\frac{n^3}{p}+n^2\cdot g+p\cdot l$, and inserting the corresponding values we get:

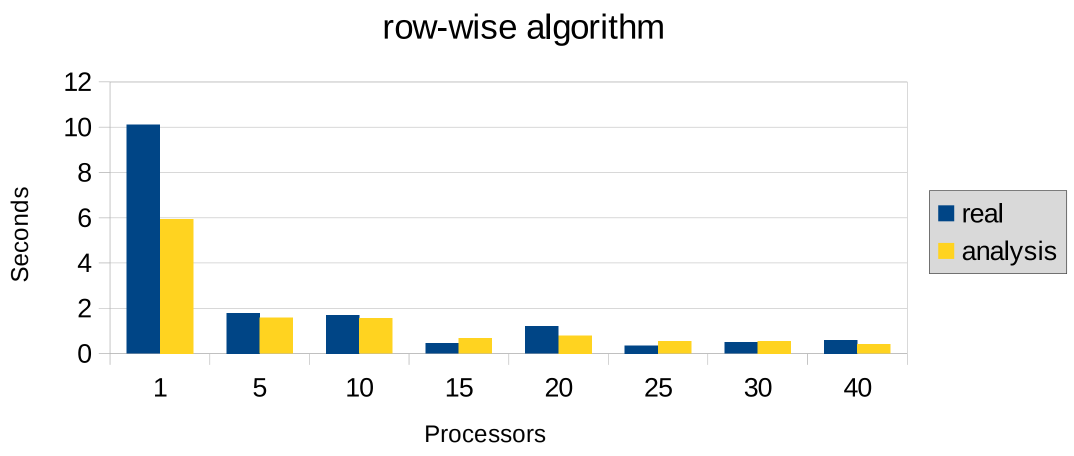
$$\frac{\frac{1800^3}{30}+1800^2\cdot 29.2+30\cdot 39825.7}{529.255\,x\,10^6}=0.55\,s$$

Considering instead $p=25$, we have $g=101.7\,flop$, $l=85036.5\,flop$ $r=1036.617\,Mflop/s$, so we get:

$$\frac{\frac{1800^3}{25}+1800^2\cdot 101.7+25\cdot 85036.5}{1036.617\,x\,10^6}=0.54\,s$$

When $p=30$ we got 0.55 seconds, while in test the running time was 0.50 seconds, so that means the theoretical analysis is very precise; however with $p=25$ we got the best speedup with a parallel running time equals to 0.35 seconds, while according to theoretical analysis we should get roughly 0.54 seconds.
Probably the better result obtained in the real experiment is due to the cache.

In the bar chart below we can see a full comparison between the expected running time and the real running time.



In Cannon’s algorithm we have the following values:

$a=\frac{n^3}{p}$, because there are $\sqrt{p}$ steps, each one with $\left(\frac{n}{\sqrt{p}}\right)^3$ operations.

$b=2\frac{n^2}{\sqrt{(p)}}$, since in each of the $\sqrt{p}$ steps processor i sends and receives two matrices of size $\left(\frac{n}{\sqrt{p}}\right)^2$.

$c=\sqrt{(p)}$.

So the final formula in Cannon's algorithm is $\frac{n^3}{p}+2\frac{n^2}{\sqrt{(p)}}\cdot g+\sqrt{(p)}\cdot l$ and when $p=30$, we get:

$$\frac{\frac{1800^3}{25}+\frac{2\cdot 1800^2}{5}\cdot 101.7+5\cdot 85036.5}{1036.617\,x\,10^6}=0.352\,s$$

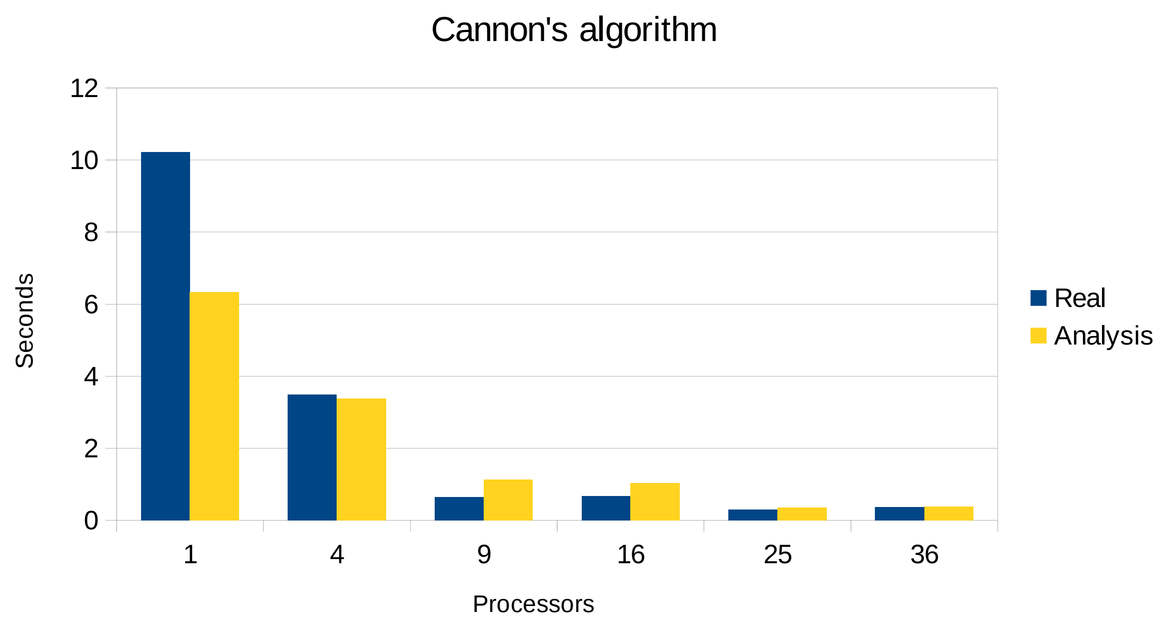
which is roughly the same result we got in the practical experiment(0.30 seconds).

Whit $p=36$ according to the benchmarking program we have $g=36.9$, $l=57395.1$, $r=529.619\,Mflop/s$, so the final result is:

$$\frac{\frac{1800^3}{36}+\frac{2\cdot 1800^2}{6}\cdot 36.9+6\cdot 57395.1}{529.619\,x\,10^6}=0.38\,s$$

Also in this case the theoretical analysis confirms the real parallel running time, which is 0.36 s.

In the bar chart below we can see a full comparison between the expected running time and the real running time for Cannon's algorithm.


Cannon's algorithm
Seconds
Processors
Real
Analysis