HBM 513E Parallel and Distributed Computing

Homework 2 Report

Parallel Dense Matrix-Matrix Multiplication with MPI

Kadriye Tuba Turkcan

18.01.2021

Introduction

This report represents the results obtained from a matrix-matrix multiplication (BLAS3 operation) parallel program (in C) based on MPI (Message Passing Interface) for various dense matrix sizes. First, a serial code with optimized matrix multiplication is conducted via C programming language on one processor. Then, a pointwise parallel matrix multiplication is conducted for various matrix sizes ranging from 2000 to 20000, on number of processors ranging from 8 to 128. After pointwise parallel matrix multiplication, blockwise parallel matrix multiplication is conducted for fixed problem sizes on 8, to 128 processor. The results are compared according to wall clock time vs matrix sizes and also wall clock time vs number of cores .

Preparation:

Here, in this study nxn matrices with double entries randomly ranging between 0-1000 are multiplied. To do that a `randBtw(double lowerb, double upperb)` function is used. The matrices are filled with the following code:

```
      double randBtw(double lowerb, double upperb)

{
      double r = upperb - lowerb;
      double d = RAND_MAX / r;
      return lowerb + (rand() / d);
}

void fill1dMat(double *A, int row,int col)
{
      int i,j;
      for (i = 0; i < row; i++)
      {
            for (j = 0; j < col; j++)
            {
                  A[i * col +j] = randBtw(0,1000);
            }
      }
}
```

Since in the C language computer memory is designed for contiguous accesses in rows, one dimensional form of matrix representation with row-major form is used for filling up the matrices.

Serial Matrix Multiplication:

First, a serial optimized matrix matrix multiplication algorithm is implemented. To do that cache blocking and loop unrolling optimization methods are applied on the naive matrix multiplication. Two approaches are compared here to get better results, both in parallel and serial version of code. The first one is blocking and loop unrolling applied on ikj- loop indexing matrix multiplication algorithm and the second one is that applied on ijk-loop indexing algorithm in which the second multiple B is transposed. Then the wall clock times are compared. Analysis show that the second approach gives better results for bigger-sized matrices although first approach seems to be faster for smaller-sized matrices.

C function gettimeofday() ,which returns the current time in the form of number of seconds, is used for time measurement. Since only the multiplication time is needed in this study, this was the most suitable way.

```
 gettimeofday(&t, NULL);

 time1 = t.tv_sec + 1.0e-6*t.tv_usec;

  MatMatMult(A,B,C,SIZE);

  gettimeofday(&t, NULL);
  time2 = t.tv_sec + 1.0e-6*t.tv_usec;

  wct = time2 - time1;
```

In main function, addresses for matrices A,B and C are allocated with the following code:

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

where SIZE is predefined as the matrix size. After the calculations, the allocated space for each one-dimensional matrix is freed with free() function.

Here, block size is also important. Several block sizes are compared on both parallel and serial algorithms . It is calculated by the formula that it should be less than square root of one thirds of the L1 cache size. Here, L1 cache in one core has a size of 32 K, or 32768 Bytes, divided by 3 gives 10922.6. Square root of this is 104.5116 Bytes, which is 13.063 words since each element is a double (8 Bytes). Since loop unrolling is applied in conjunction with cache blocking, a block size which is divisible by the matrix size is needed. The matrix sizes are chosen to be 2048, 4096, 6144, 8192, 10240, 12288, 14336, 16384, 18432, 20480 so that each matrix size could be divisible by the number of cores 8, 16, 32, 64, 128. Thus, block size of 8 comes to mind at the first look. Trials on the block sizes 8, 16, 32 show that indeed block size 8 would be the correct size.

Blocking and loop unrolling on ikj-indexing:

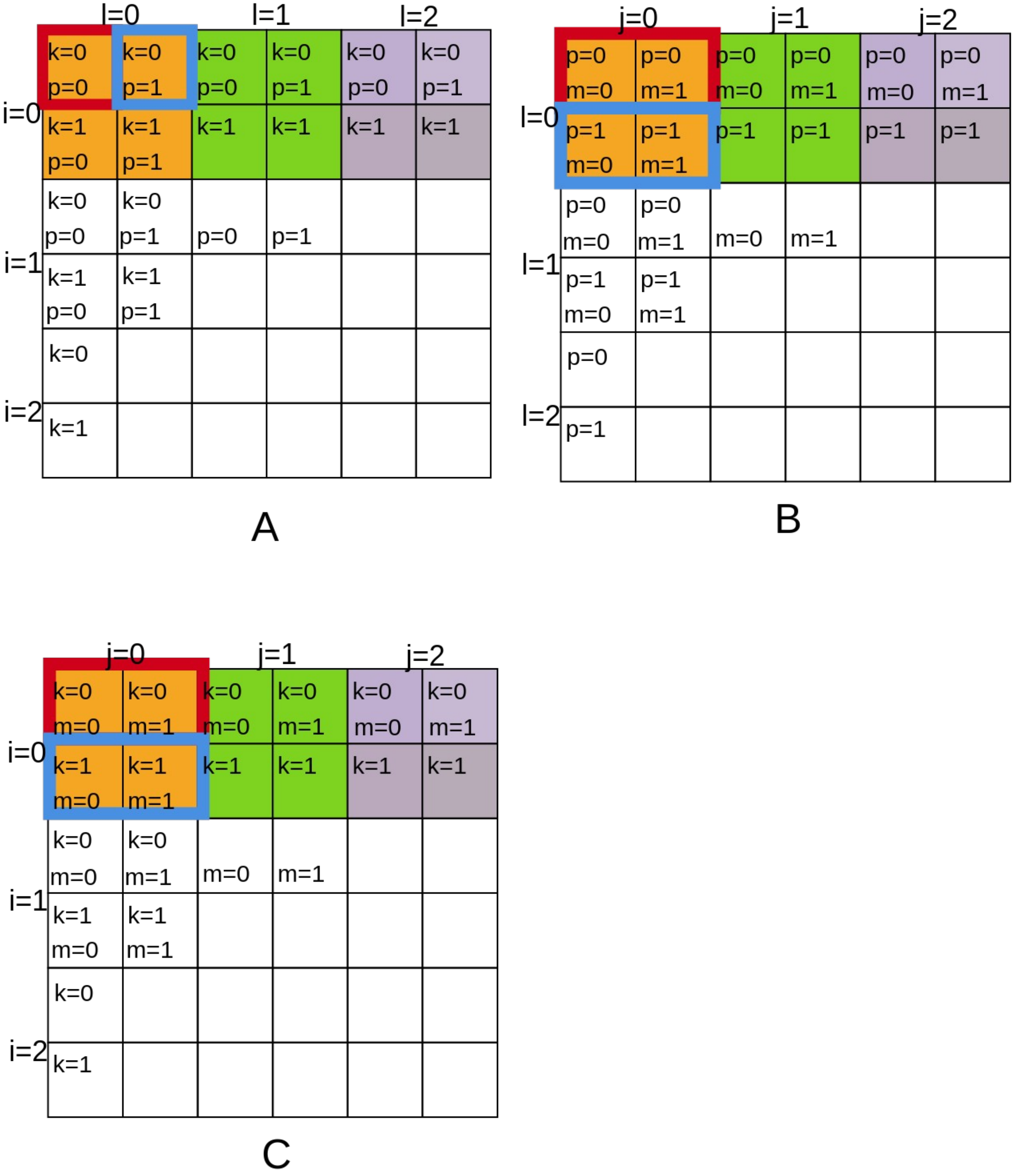
The following code is used for blocking and loop unrolling on ikj-indexing on multiplying two matrices A and B and storing them in matrix C where the matrices are all of dimension n × n.

```
void MatMatMult(double *A, double *B, double *C, int n)
{
      int i,j,k,m,l,p;
      int nb_of_blocks,block_size;
      block_size=8 ;
      nb_of_blocks= n / block_size;
      double r;
```

```
    for(i=0 ; i < nb_of_blocks; i++)
    {
        for (l = 0 ; l < nb_of_blocks; l++)
        {
            for ( k=0; k < block_size ; k++)
            {
                for(p=0; p < block_size; p++)
                {
                r= A[i*block_size*n + l*block_size + k*n +p];
                    for ( j=0 ; j<nb_of_blocks ; j++)
                    {
                        for (m=0; m < block_size ; m+=8)
                        {
                    C[i*block_size*n + j*block_size + k*n + m] += r *
                    B[l*block_size*n + j *block_size + p*n +m];
                    C[i*block_size*n + j*block_size + k*n + m+1] += r *
                    B[l*block_size*n + (j) *block_size + (p)*n + m+1];
                    C[i*block_size*n + j*block_size + k*n + m+2] += r *
                    B[l*block_size*n + (j) *block_size + (p)*n + m+2];
                    C[i*block_size*n + j*block_size + k*n + m+3] += r *
                    B[l*block_size*n + (j) *block_size + (p)*n + m+3];
                    C[i*block_size*n + j*block_size + k*n + m+4] += r *
                    B[l*block_size*n + (j) *block_size + (p)*n + m+4];
                    C[i*block_size*n + j*block_size + k*n + m+5] += r *
                    B[l*block_size*n + (j) *block_size + (p)*n + m+5];
                    C[i*block_size*n + j*block_size + k*n + m+6] += r *
                    B[l*block_size*n + (j) *block_size + (p)*n + m+6];
                    C[i*block_size*n + j*block_size + k*n + m+7] += r *
                    B[l*block_size*n + (j) *block_size + (p)*n + m+7];
                        }
                    }
                }
            }
        }
    }
}
```

Here, all the three matrices, A,B and C are accessed in row major order as one dimensional arrays and calculations are also conducted in that way since in C language, the access into memory is conducted as rows . This way a faster result would be obtained than a column major access. Moreover, r is stored in register which makes calculations faster instead of accessing A matrix for each loop.

A[i*blocksize*n+ l*blocksize + k*n + p] * B[l*blocksize*n+ j*blocksize + p*n + m]
= C[i*blocksize*n+ j*blocksize + k*n + m]



The figure above shows the access of matrices. Here, loop m is unrolled since in both B and C matrices loop m helps us access in row-wise manner which decreases the miss rate and thus makes the calculation faster. The comparison of trials on unrolling loop p with this approach also agrees with this.

Blocking and loop unrolling on ijk-indexing with B transposed:

The following code is used for this second approach.

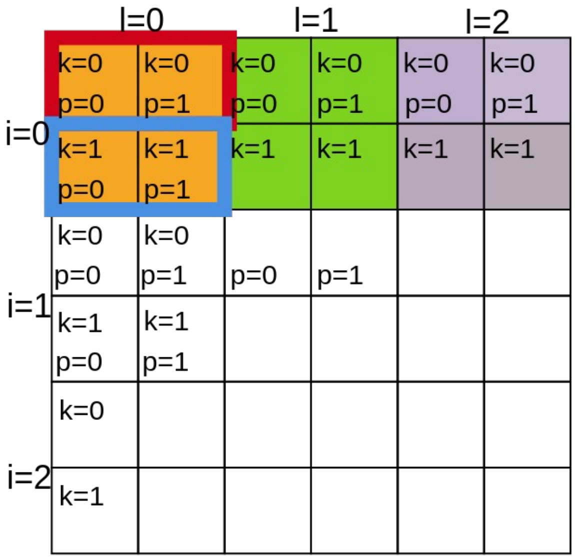
```
void MatMatMult(double *A, double *B, double *C, int n){

      int i,j,k,m,l,p;
      double sum;
      int nb_of_blocks,block_size;
      block_size= 8;
      nb_of_blocks= n / block_size;
      transpose(B,SIZE);

      for(i=0 ; i < nb_of_blocks; i++)
      {
            for ( j=0 ; j<nb_of_blocks ; j++)
            {
                  for ( k=0; k < block_size ; k++)
                  {
                        for (m=0; m < block_size ; m++)
                        {
                              sum =0.0;
                              for (l = 0 ; l < nb_of_blocks; l++)
                              {
                                    for(p=0; p < block_size; p+=8)
                                    {
                        sum += A[i*block_size*n + l*block_size + k*n +p] *
                              B[j*block_size*n + l *block_size + m*n +p];
                        sum += A[i*block_size*n + (l)*block_size + k*n +p+1] *
                              B[j*block_size*n + (l) *block_size + m*n +p+1];
                        sum += A[i*block_size*n + (l)*block_size + k*n +p+2] *
                              B[j*block_size*n + (l) *block_size + m*n +p+2];
                        sum += A[i*block_size*n + (l)*block_size + k*n +p+3] *
                              B[j*block_size*n + (l) *block_size + m*n +p+3];
                        sum += A[i*block_size*n + (l)*block_size + k*n +p+4] *
                              B[j*block_size*n + (l) *block_size + m*n +p+4];
                        sum += A[i*block_size*n + (l)*block_size + k*n +p+5] *
                              B[j*block_size*n + (l) *block_size + m*n +p+5];
                        sum += A[i*block_size*n + (l)*block_size + k*n +p+6] *
                              B[j*block_size*n + (l) *block_size + m*n +p+6];
                        sum += A[i*block_size*n + (l)*block_size + k*n +p+7] *
                              B[j*block_size*n + (l) *block_size + m*n +p+7];
                                    }

                              }
                        C[i*block_size*n + j*block_size + k*n + m] = sum;

                        }
                  }
            }
      }
}
```

Here, all the three matrices, A,B and C are accessed in row major order as one dimensional arrays and calculations are also conducted in that way since in C language, the access into memory is conducted as rows . Moreover, sum is stored in register which makes calculations faster instead of accessing A matrix for each loop. Loop p is unrolled here since inside the p loop the matrices are accessed in row major order.

The following figure shows the access of matrices but here the second matrix is the transpose of B:

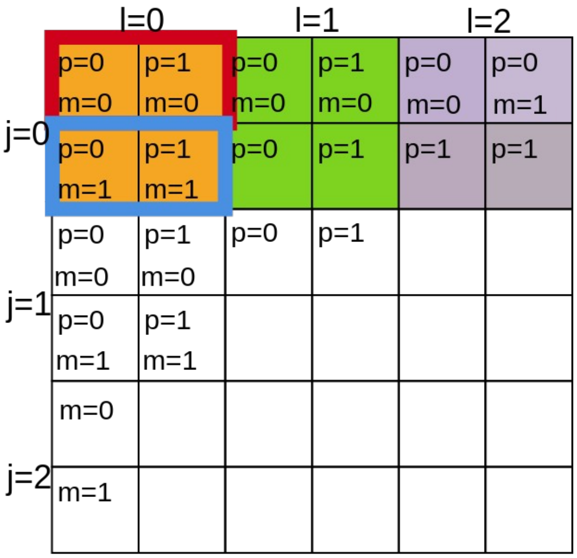
A[i*blocksize*n+ l*blocksize + k*n + p] * B^{T}[j*blocksize*n+ l*blocksize + m*n + p]

= C[i*blocksize*n+ j*blocksize + k*n + m]



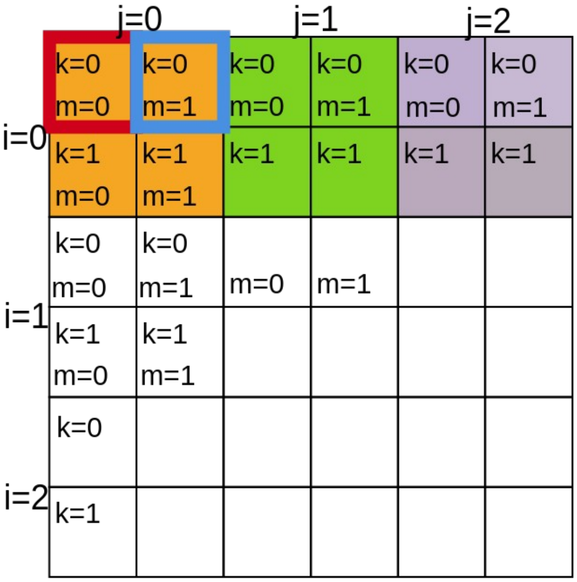
	l=0		l=1		l=2	
i=0	k=0 p=0	k=0 p=1	k=0 p=0	k=0 p=1	k=0 p=0	k=0 p=1
	k=1 p=0	k=1 p=1	k=1	k=1	k=1	k=1
i=1	k=0 p=0	k=0 p=1	p=0	p=1		
	k=1 p=0	k=1 p=1				
i=2	k=0					
	k=1					

A



	l=0		l=1		l=2	
j=0	p=0 m=0	p=1 m=0	p=0 m=0	p=1 m=0	p=0 m=0	p=0 m=1
	p=0 m=1	p=1 m=1	p=0	p=1	p=1	p=1
j=1	p=0 m=0	p=1 m=0	p=0	p=1		
	p=0 m=1	p=1 m=1				
j=2	m=0					
	m=1					

B^{T}



	j=0		j=1		j=2	
i=0	k=0 m=0	k=0 m=1	k=0 m=0	k=0 m=1	k=0 m=0	k=0 m=1
	k=1 m=0	k=1 m=1	k=1	k=1	k=1	k=1
i=1	k=0 m=0	k=0 m=1	m=0	m=1		
	k=1 m=0	k=1 m=1				
i=2	k=0					
	k=1					

C

The following table shows the comparison of block sizes which shows that block size of 8 gives better results for large sized matrices.

Matrix size	Serial ikj b=8	Serial ikj b=16	Serial ijk b=8	Serial ijk b=16
2048	56.91	113.50	32.51	32.50
4096			262.39	
6144			876.78	882.18

From the above table we can see that cache blocking and loop unrolling on ijk-indexing with B transposed gives better results than the other approach.

Parallel Matrix Multiplication:

A parallel algorithm with MPI is written and tested on the matrix sizes 2048, 4096, 6144, 8192, 10240, 12288, 14336, 16,384, 18432 and 20480 , for the core numbers 8 , 16, 32, 64 and 128. The matrix sizes are chosen that way to get an even partitioning of matrices across cores. Pointwise and also block-wise matrix multiplication is done with this parallel algorithm and results are compared.

Here, all matrices are allocated with dynamic memory allocation as one dimensional arrays to decrease the computation cost. First, memory is allocated in each core for B, A_block, B_block, C_block matrices and in core 0 for matrix C. Matrix A and B are partitioned into cores by generating matrices A_block and B_block inside each core with the size of (SIZE / nb of processors). Then B_block matrices are gathered in all cores with MPI's allgatherv() function so each core obtained B matrix. Then inside each core A_block matrices are multiplied by B matrices and stored in C_block matrices. After that, C_block matrices are gathered with gatherv() function inside core 0. Time is calculated with MPI_Wtime() function. All the allocated matrices are freed in the end of main().

Here, collective communication is used instead of point to point communication because it is known that with point to point communication, communication cost increases with the increase of number of cores. To get faster results, allgatherv and gatherv is used. Since allgather and gather functions gather just the first line of a matrix, vector version of these are chosen to be used.

To do the collective communication with algatherv and gatherv and to get each block matrix B_block within each processor, a new array data type is created with `MPI_Type_create_subarray(2, bigsizes, subsizes, starts, MPI_ORDER_C, MPI_DOUBLE, &new_B_block);` function. This subarray represents each row of B_block and C_block. After the creation of this new data type, they are resized back to double to be able to do the calculations inside Algatherv and gatherv with `MPI_Type_create_resized(new_B_block, 0, sizeof(double), &B_blk);` .

```
      MPI_Datatype new_B_block, B_blk;
      int starts[2] = {0,0};
      int subsizes[2] = {1, SIZE};
      int bigsizes[2] = {block_size, SIZE};
      MPI_Type_create_subarray(2, bigsizes, subsizes, starts, MPI_ORDER_C,
MPI_DOUBLE, &new_B_block);
      MPI_Type_create_resized(new_B_block, 0, sizeof(double), &B_blk);
      MPI_Type_commit(&B_blk);
```

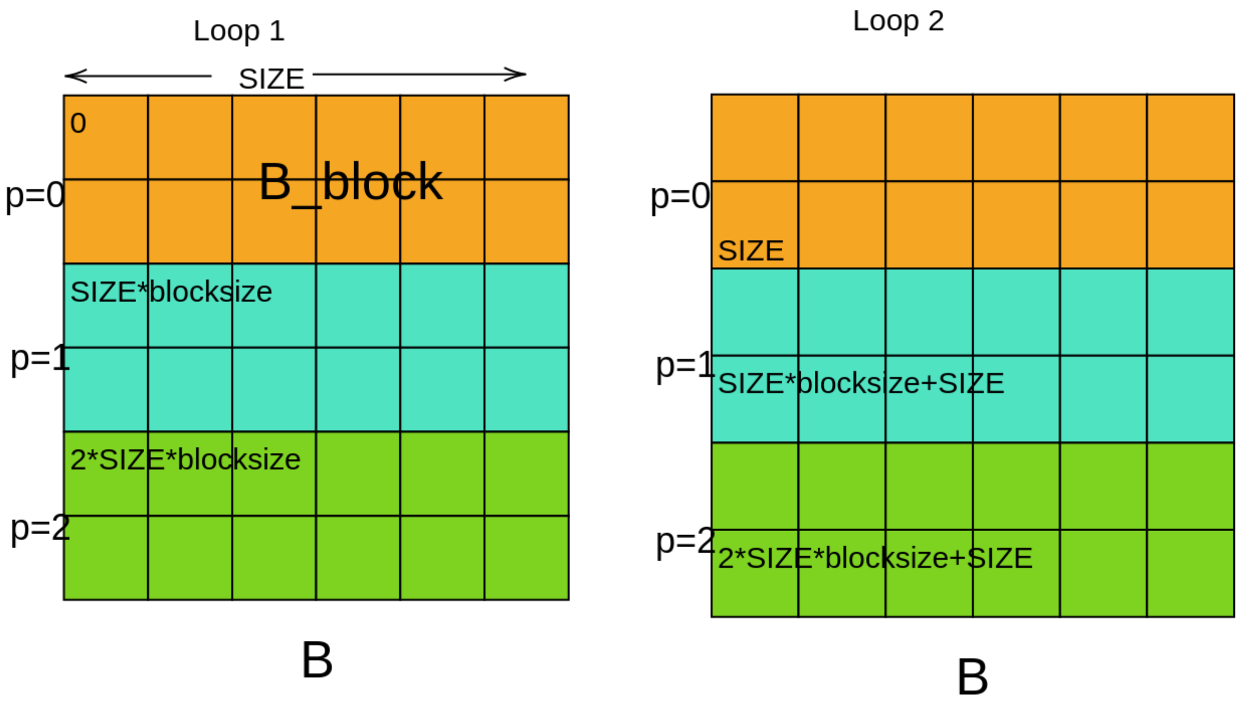
This new datatype is gathered within a loop to obtain the whole matrix B and C.

```
for (i=0; i< block_size; i++)

            MPI_Allgatherv(&B_block[SIZE*i],1, B_blk, B, rcv_counts, disp[i],
B_blk, MPI_COMM_WORLD );
```

The displacements are given as a two dimensional array to be able to fit within this loop. The displacements are put at the beginning of each row in B.

```
      for (j=0; j<block_size; j++)

      {
            for (i=0; i< np; i++)
                  {
                        disp[j][i]= i * block_size*SIZE+SIZE*j;
                  }
      }
```

The following figure shows this:



The following code is used for the parallel algorithm:

```
      int main(int argc, char ** argv)

{
      int my_rank, np,i,j;
      int block_size, send_count;
      double *B, *A_block, *B_block, *C_block, *C;
      double start, finish;

      MPI_Init(&argc, &argv);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    block_size = SIZE / (np);
    int disp[block_size][np];
    int rcv_counts[np];



    A_block = (double*)malloc(block_size*SIZE* sizeof(double));
    B_block = (double*)malloc(block_size* SIZE*sizeof(double));
    C_block = (double*)malloc(block_size* SIZE*sizeof(double));

    B = (double*)calloc(SIZE * SIZE , sizeof(double));

    if (my_rank == 0)
        C = (double*)calloc(SIZE * SIZE , sizeof(double));

    fill1dMat(A_block, block_size, SIZE);
    fill1dMat(B_block, block_size, SIZE);

    MPI_Barrier(MPI_COMM_WORLD);

    for (i=0; i< np; i++)
        rcv_counts[i]=1;


    for (j=0; j<block_size; j++)
    {
        for (i=0; i< np; i++)
            {
                disp[j][i]= i * block_size*SIZE+SIZE*j;
            }
    }

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Datatype new_B_block, B_blk;

    int starts[2] = {0,0};
    int subsizes[2] = {1, SIZE};
    int bigsizes[2] = {block_size, SIZE};
    MPI_Type_create_subarray(2, bigsizes, subsizes, starts, MPI_ORDER_C,
MPI_DOUBLE, &new_B_block);
    MPI_Type_create_resized(new_B_block, 0, sizeof(double), &B_blk);
    MPI_Type_commit(&B_blk);

    for (i=0; i< block_size; i++)

        MPI_Allgatherv(&B_block[SIZE*i],1, B_blk, B, rcv_counts, disp[i],
B_blk, MPI_COMM_WORLD );

    MPI_Barrier(MPI_COMM_WORLD);
    start = MPI_Wtime();

    MatMatMult(A_block, B, C_block, block_size, SIZE,  SIZE, SIZE );

    for (i=0; i< block_size; i++)
```

```
            MPI_Gatherv(&C_block[SIZE*i],1, B_blk, C, rcv_counts, disp[i],
B_blk,0, MPI_COMM_WORLD );

      finish = MPI_Wtime();
      if (my_rank==0)
      {
            printf("Proc %d > Elapsed time = %6.4lf seconds\n", my_rank, finish-
start);
            free(C);}

      MPI_Barrier(MPI_COMM_WORLD);
      MPI_Type_free (&B_blk);

      free(A_block);
      free(B_block);
      free(C_block);
      free(B);
      MPI_Finalize();
      return 0;
}
```

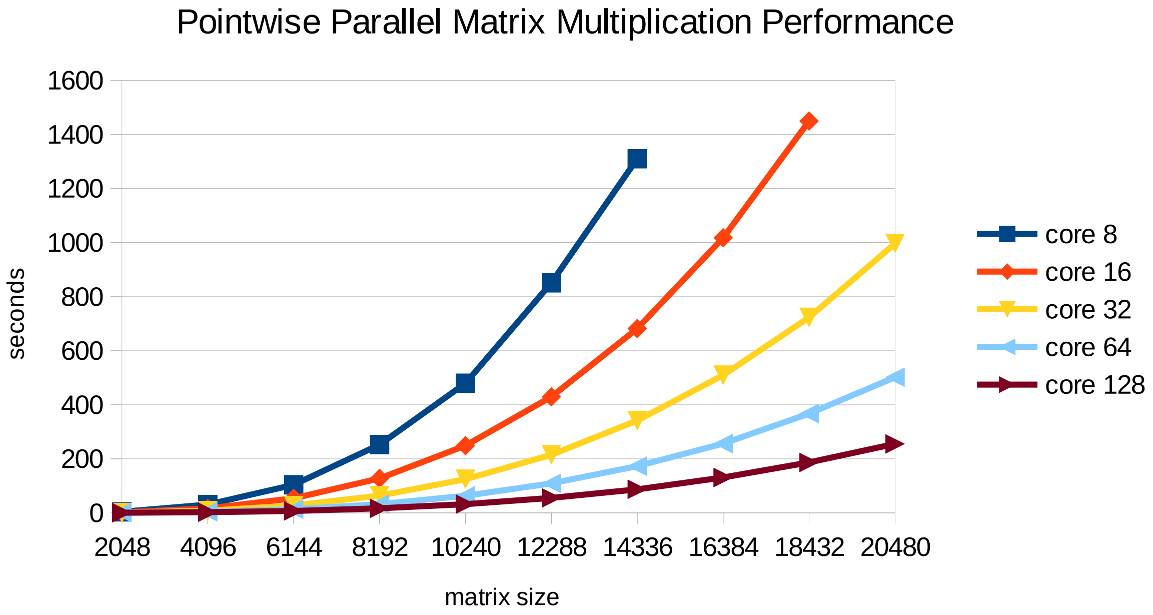
Pointwise Parallel Matrix Multiplication:

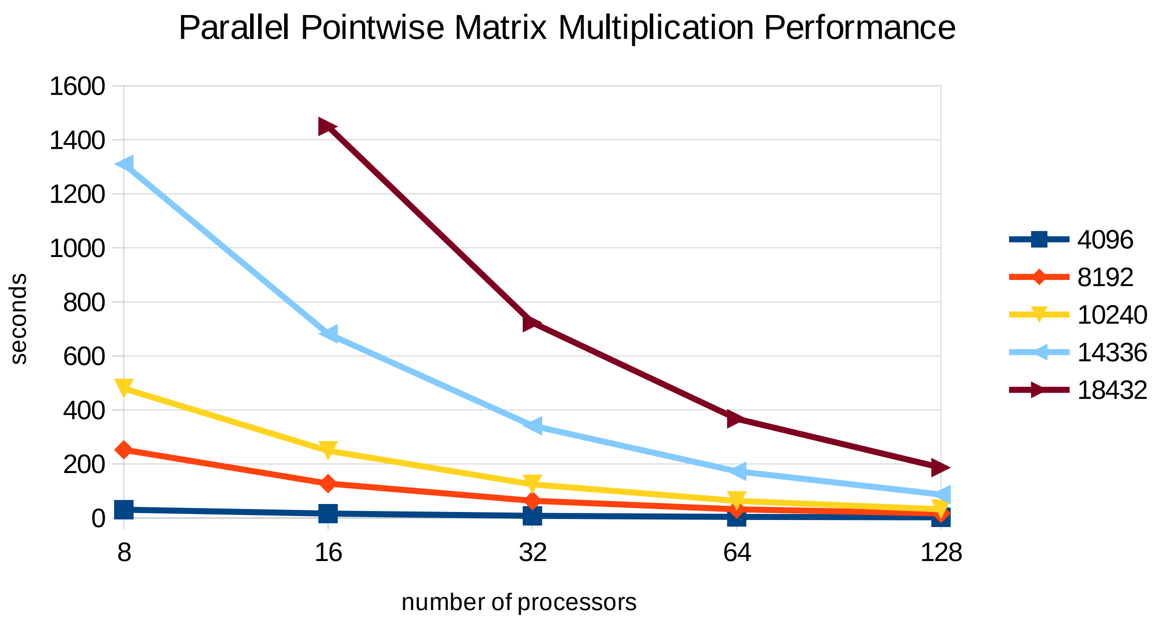
For the pointwise matrix multiplication, ikj- indexing is used:

```
      void MatMatMult(double *A, double *B, double *C, int row1, int col1, int
row2, int col2)

{
      int i,j,k;
      double sum;
      for(i = 0; i < row1; i++)
      {
            for (k = 0; k < col1; k++)
            {
                  r = A[i * col1 + k];
                  for( j = 0; j < col2; j++)
                  {
                        C[i * col2 + j] += r * B[k * col2 + j];

                  }
            }
      }
}
```

Matrix size	Core 8	Core 16	Core 32	Core 64	Core 128
2048	3.83	1.99	1.03	0.59	0.47
4096	30.68	15.96	8.02	4.12	2.28
6144	103.62	53.85	26.98	13.68	7.12
8192	252.54	127.37	63.9	32.44	16.52
10240	479.22	248.85	124.64	63.04	32.19
12288	850.73	429.39	215	109.2	55.31
14336	1309.81	681.78	341.28	172.78	87.05
16384	time limit	1017.37	509.3	256.4	130.64
18432	time limit	1449.47	723.35	367.05	186.35
20480	time limit	time limit	996.75	501.63	255.04

The results are compared in the following graph. As it can be seen from the graph that, with the increase of matrix size the wall clock time increases in all the number of processors. Partitioning the matrices into more processors gives better results than less number of processors since collective communication is used. We can relate this to the architecture of the memory that as the data increases within a processor, calculations take more time although collective communication is used and communication cost is decreased.





The above graph shows number of processors vs time for pointwise parallel matrix multiplication for fixed matrix sizes. It can be seen that as the number of processors increase the performance of the algorithm also increases. The best performance is obtained with 128 processors.

Blockwise Matrix Multiplication

The matrix multiplication is conducted in a blockwise manner with loop unrolling which is the same approach used for serial computation. Here again, the block size for various matrix sizes are compared and it is observed that block size of 8 gives better performance. Also blockwise ikj-indexing and blockwise ijk-indexing with B transposed are also compared and it is seen that the second approach gives better results for larger size matrices. The following table shows this for 8 cores, time is given in seconds.

Matrix Size	ijk-indexing (B transposed) b=32	ijk-indexing (B transposed) b=8	ijk-indexing (B transposed) b=16	ikj-indexing b=8
2048		4.6	4.58	23.0075
4096	36.5575	37.5	36.29	
6144		122.12	122.26	
8192		298.96		
10240		563.79	564.1	

The following code is used for blockwise parallel matrix multiplication:

```
void MatMatMult(double *A, double *B, double *C, int row1, int col1, int col2)

{
      int i,j,k,m,l,p;
      double sum;
      int nb_of_blocks1,nb_of_blocks2,blk_size;
      blk_size= 8;
      nb_of_blocks1= row1 / blk_size;
      nb_of_blocks2 = col1 /blk_size;
      transpose(B,col1,col2);

      for(i=0 ; i < nb_of_blocks1; i++)
      {
            for ( j=0 ; j<nb_of_blocks2 ; j++)
            {
                  for ( k=0; k < blk_size ; k++)
                  {
                        for (m=0; m < blk_size ; m++)
                        {
                              sum =0.0;
                              for (l = 0 ; l < nb_of_blocks2; l++)
                              {
                                    for(p=0; p < blk_size; p+=8)
                                    {
                        sum += A[i*blk_size*col1 + l*blk_size + k*col1 +p] *
                              B[j*blk_size*col2 + l *blk_size + m*col2 +p];
                        sum += A[i*blk_size*col1 + (l)*blk_size + k*col1 +p+1] *
                              B[j*blk_size*col2 + (l) *blk_size + m*col2 +p+1];
                        sum += A[i*blk_size*col1 + (l)*blk_size + k*col1 +p+2] *
                              B[j*blk_size*col2 + (l) *blk_size + m*col2 +p+2];
                        sum += A[i*blk_size*col1 + (l)*blk_size + k*col1 +p+3] *
                              B[j*blk_size*col2 + (l) *blk_size + m*col2 +p+3];
                        sum += A[i*blk_size*col1 + (l)*blk_size + k*col1 +p+4] *
                              B[j*blk_size*col2 + (l) *blk_size + m*col2 +p+4];
                        sum += A[i*blk_size*col1 + (l)*blk_size + k*col1 +p+5] *
                              B[j*blk_size*col2 + (l) *blk_size + m*col2 +p+5];
                        sum += A[i*blk_size*col1 + (l)*blk_size + k*col1 +p+6] *
                              B[j*blk_size*col2 + (l) *blk_size + m*col2 +p+6];
                        sum += A[i*blk_size*col1 + (l)*blk_size + k*col1 +p+7] *
                              B[j*blk_size*col2 + (l) *blk_size + m*col2 +p+7];
                                    }

                              }
                        C[i*blk_size*col2 + j*blk_size + k*col2 + m] = sum;

                        }
                  }
            }
      }
}
```

Here, since the multiplication is done between B_Block and A_Block matrix within each processor, matrix row and column values are not equal, so there are two number of blocks, nb_of_blocks1 and nb_of_blocks2. The loops and multiplication is conducted accordingly.

As we know that the data is stored as arrays of rows in the cache, it would be better to transpose the column vectors in B to get a row vectors. The transpose function is given in the following code block:

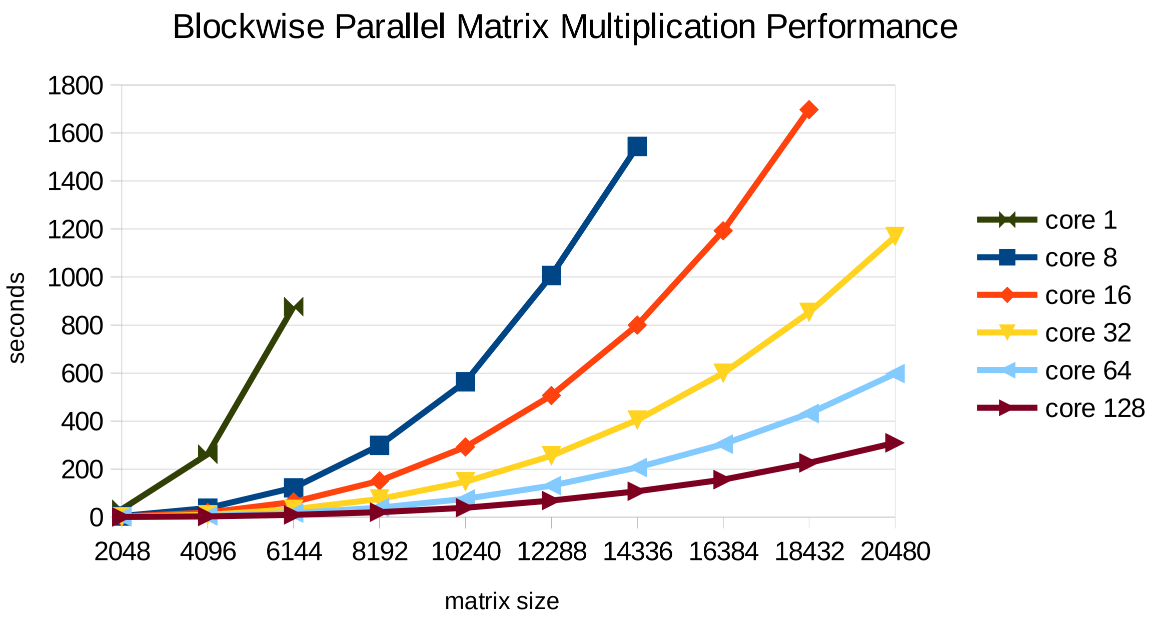
```
void transpose(double *A,int row, int col)
{
      int i,j;
      double tmp;
      for(i = 0; i < row; i++)
      {
            for ( j = i+1; j < col; j++)
            {
                  tmp = A[i * col + j];
                  A[i * col + j]= A[j * row + i];
                  A[j * row + i]=tmp;
            }
      }
}
```

Results

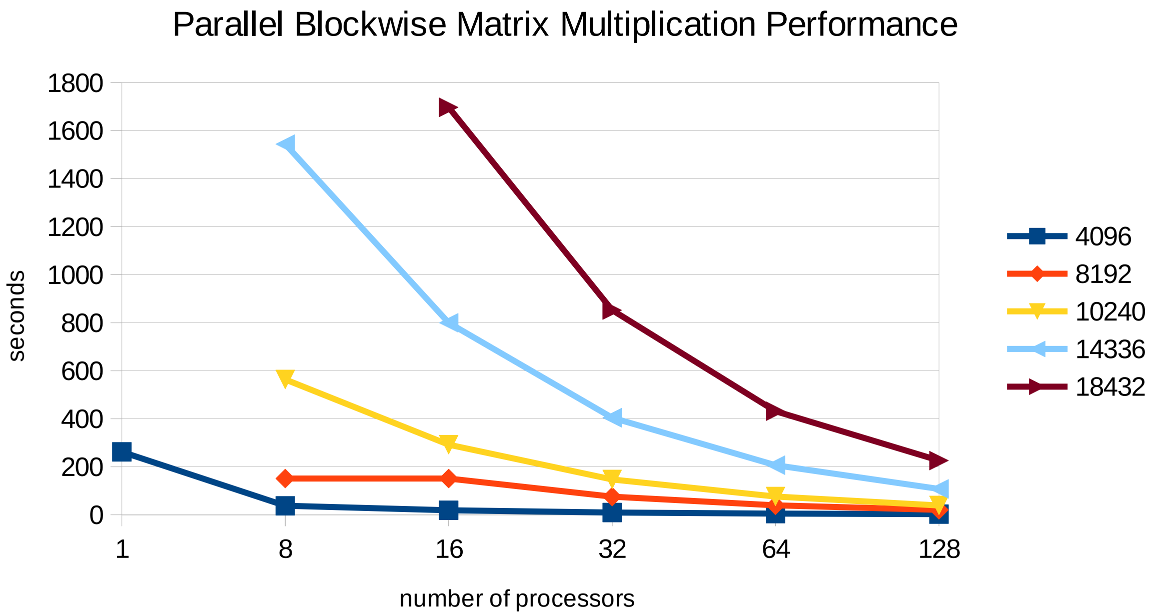
The results for the blockwise matrix multiplication is given in the following table.

Matrix size	Core 8	Core 16	Core 32	Core 64	Core 128
2048	4.6	2.42	1.27	0.75	0.56
4096	37.5	18.89	9.75	5.23	2.93
6144	122.12	63.42	32.44	16.98	9.3
8192	298.96	150.84	75.77	39.61	20.52
10240	563.79	292.17	147.2	76.25	39.06
12288	1006.37	506.39	255.47	131.02	69.08
14336	1543.95	799.66	404.34	206.51	107.64
16384	time limit	1192.58	599.51	303.93	156.29
18432	time limit	1697.23	852.81	431.81	225.91
20480	time limit	time limit	1168.77	597.84	309.3

The following graph shows matrix size vs time in seconds on parallel blockwise matrix multiplication, with number of cores from 8 to 128. It can be seen from the graph that as matrix size increases the wall clock time increases. The best performance is seen with 128 cores. The time doesn’t increase much since the number of rows generated in each core is still very few compared to others. On the other hand, with 8 cores, since the size of A_Block and B_Block are greater, it is slower than the others.

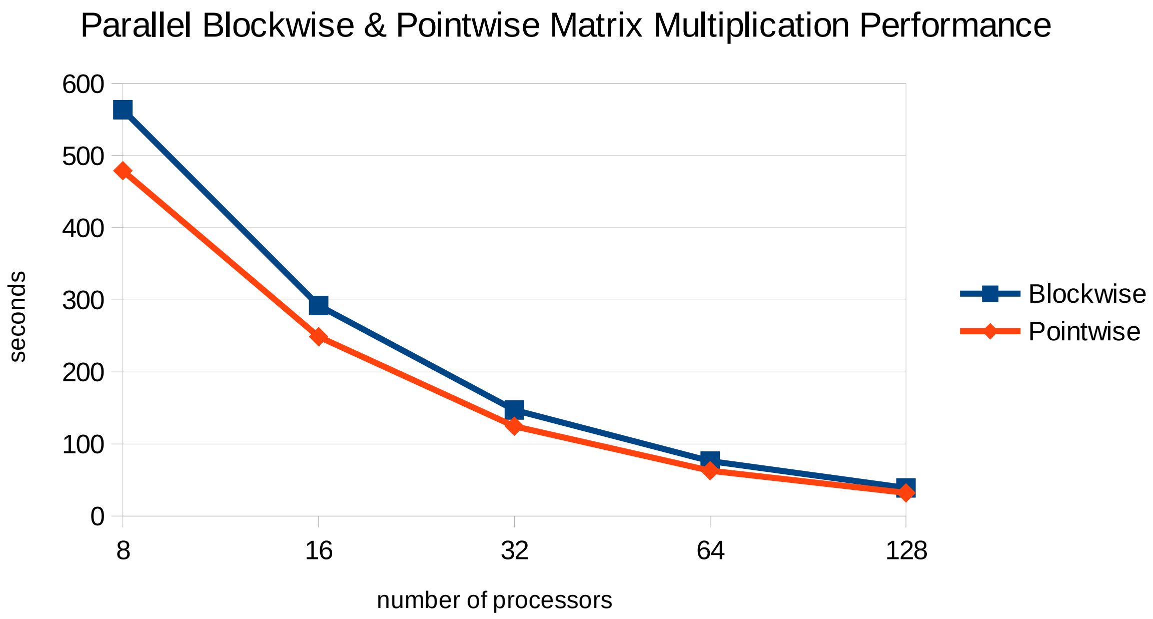


The following graph shows number of cores vs time in seconds on fixed matrix size for parallel block matrix multiplication.



The graph shows that with the increase of number of processors, the wall clock time decreases. Since collective communication is used instead of point to point send and receive, the communication time does not increase as expected. When the number of processors increase the size of matrix partitioned into gets smaller and thus calculations done faster.

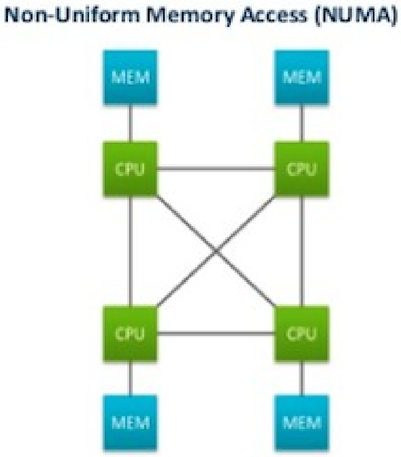
The following graph shows the comparison of Pointwise and Blockwise parallel matrix multiplication with matrix size 10240. Although we would expect blockwise to be faster, pointwise gives faster results.



The machine used is Intel Xeon E5-2680. The following table shows the details of the machine architecture.

Architecture	X86_64
Memory Bandwidth	76.8 GB/s
L1_i Cache Size	32768
L1_i Cache Line Size	64
L1_d Cache Size	32768
L1_d Cache Line Size	64
L2 Cache Size	262144
L2 Cache Line Size	64
L3 Cache Size	36700160
L3 Cache Line Size	64
CPUs	28
CPU MHz	2900.343
BogoMPIS	4804.82

It has high bandwith, low latency. We try mostly to fit the matrices within L1 cache since it is the fastest memory unit. Since C programming language accesses memory row-wise manner, we tried to optimize this with both blockwise and pointwise parallel multiplication. Here, NUMA architecture is used where memory is not shared, it is distributed memory.



Conclusion

In this report, we analyzed parallel matrix multiplication with sizes changing from 2000 to 20000, with an increment of 2000. Serial matrix multiplication algorithm is applied first in a blockwise manner. Parallel matrix multiplication algorithm is written using collective communication to decrease communication cost. Both pointwise and blockwise algorithms are used for parallel multiplication. For various matrix sizes the number of cores vs time in seconds is observed.