Parallel Programming using Charm++

EGR 4303 – Applied Research – Fall 2016

Al Akhawayn University

Student: Ali ELABRIDI – a.elabridi@aui.ma

Supervisor: Dr. Naeem Nisar Sheikh

Student: Ali ELABRIDI – a.elabridi@aui.ma

Supervisor: Dr. Naeem Nisar Sheikh

Parallel Programming using Charm++ for the Strassen Algorithm of Matrix Multiplication

Abstract:

Parallel programming paradigms have been developing to answer the high demand for high performance computing and exhaustive scientific computational research that sometimes requires days to complete the computational tasks. Multiples programming languages and paradigms have existed such as OpenMP, Grand Central Dispatch, Cilk++ and so on with each having its strengths and weaknesses. the purpose of this applied research is to explore a new approach that has been implemented and designed in Charm++ by implementing Strassen algorithm for Matrix Multiplication, and analysis of some key aspect of the algorithm.

Overview / introduction

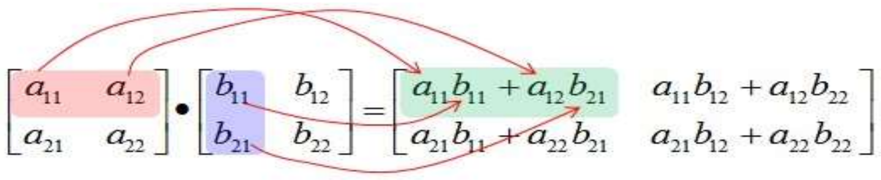
Why using Charm+ as a new parallel computing approach +?

- Charm++ is a parallel object-oriented programming language based on C++ developed in the Parallel Programming Laboratory at the University of Illinois
- Provides a high-level abstraction
- Good performance on many types of hardware
- Strong load balancing capabilities
- Not yet used in the intel challenge

Naïve Matrix Multiplication:

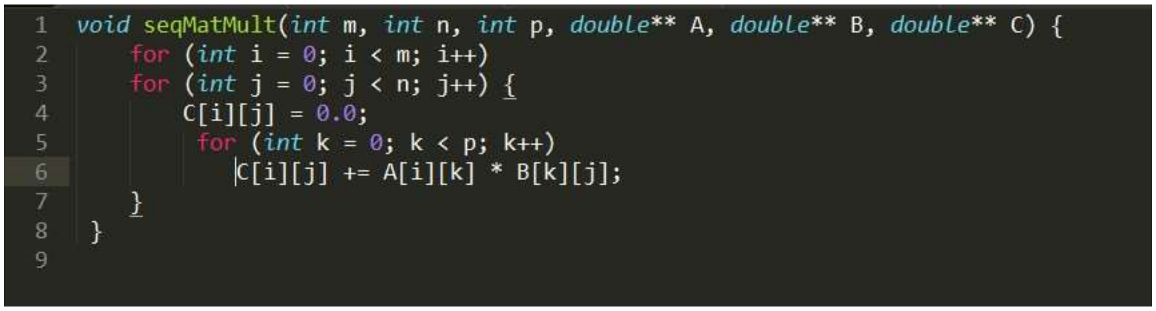
The purpose of the matrix multiplication is to take a matrix A of dimension n*p and another matrix B of dimension k*m and produce a matrix C of dimension n*m such that C = A * B.

The illustration bellow shows a matrix multiplication of n * n matrices with a naïve approach and its implementation with C:



$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \bullet \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

$$\mathbf{C}_{1,1} = \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1}$$
$$\mathbf{C}_{1,2} = \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2}$$
$$\mathbf{C}_{2,1} = \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1}$$
$$\mathbf{C}_{2,2} = \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2}$$



```
void seqMatMult(int m, int n, int p, double** A, double** B, double** C) {
    for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++) {
        C[i][j] = 0.0;
         for (int k = 0; k < p; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
 }
```

The problem with the Naïve Matrix Multiplication is that it has a computational Complexity of O(n^3) and is composed mainly of sequential computations that needs to wait for previous results.

Strassen Algorithm:

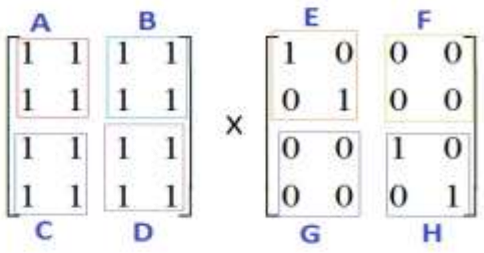
The Strassen is a great algorithm to implement in parallelism as it performs Matrix Multiplication in an independent way for every 4 partitions of a matrix and can be defined recursively for large matrices, and it has a smaller computation complexity of $O([7+o(1)]^n) = O(N^{\log_2 7+o(1)}) \approx O(N^{2.8074})$

Limitation

- Depending on the implementation, the Strassen outperform Naïve matrix multiplication only for extremely large matrices starting of size 2^{15}.
- The matrix multiplication needs to be of two square matrices of size of powers of 2.

Algorithm Approach

- Partition our matrices into four sub matrices of size (n/2)



$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Compute 7 sub-matrices of these to get the final result:

M1 = A(F-H)

M2 = H(A+B)

M3 + E(C+D)

M4 = D(G-E)

M5 = (A+D)*(E+H)

M6 = (B-D)*(G+H)

M7 = (A-C)*(E+F)

$$M1 = A(F-H) = \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}(\begin{bmatrix}0 & 0\\0 & 0\end{bmatrix} - \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}) = \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}$$

$$M2 = H(A+B) = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}(\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix} + \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}) = 2 * \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}$$

$$M3 + E(C+D) = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}(\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix} + \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}) = 2*\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}$$

$$M4 = D(G-E) = \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}(\begin{bmatrix}0 & 0\\0 & 0\end{bmatrix} + \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}) = \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix})$$

$$M5 = (A+D)*(E+H) = (\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix} + \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}) * (\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} + \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}) = 2 * \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix})$$

$$M6 = (B-D)*(G+H) = (\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix} - \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}) * (\begin{bmatrix}0 & 0\\0 & 0\end{bmatrix} + \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}) = \begin{bmatrix}0 & 0\\0 & 0\end{bmatrix}$$

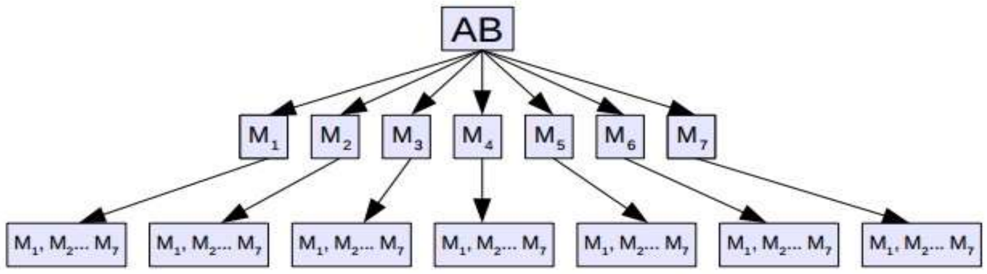
$$M7 = (A-C)*(E+F) = (\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix} - \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}) * (\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} + \begin{bmatrix}0 & 0\\0 & 0\end{bmatrix}) = \begin{bmatrix}0 & 0\\0 & 0\end{bmatrix}$$

- Gather the 7 sub-matrices (n/2) to reconstruct the multiplication of n*n matrices

$$XY = \begin{bmatrix}(M6 + M5 + M4 - M2) & (M1 + M2)\\ (M3 + M4) & (M1 + M5 - M3 - M7)\end{bmatrix} = \begin{bmatrix}1 & 1 & 1 & 1\\1 & 1 & 1 & 1\\1 & 1 & 1 & 1\\1 & 1 & 1 & 1\end{bmatrix}$$

The parallelism approach in the Strassen algorithm of complexity

the evaluation of the 7 submatrices M1,M2,...M7 intermediate matrices are independent, and therefore can be computed independently from each other.



Problem presentation:

This applied research will try to address the Intel Challenge for computing Matrix Multiplications with Strassen Algorithm. The advantage will be to compare performance results of our implementation of charm++ to already implemented solutions with other paradigms.

Research objectives:

- Design and Implement a parallel version of the Algorithm using Charm++
- Design and Implement a mixed version that uses naive and Strassen's Algorithm
- Analysis of performance compared to the already existing submission to the intel challenge using OpenMP, Cilk++...
- Analysis of performance against our own parallel version by tweaking the number of processors involved and the contribution of the naive algorithm
- Draw a conclusion about the effectiveness of charm++

Intel Problem Description:

Write a threaded code to multiply two random matrices using Strassen's Algorithm. The application will generate two matrices A(M,P) and B(P,N), multiply them together using (1) a sequential method and then (2) via Strassen's Algorithm resulting in C(M,N). The application should then compare the results of the two multiplications to ensure that the Strassen's results match the sequential computations.

You are also allowed to change the memory allocation and other code to thread the Strassen's computations.

Timing: The time for execution of the Strassen's Algorithm will be used for scoring. Each submission should include timing code to measure and print this time to stdout. If not, the total execution time will be used.

Procedure/ Methodology

Installation/Environment

The parallelization of Strassen algorithm and its performance analysis needs high performance computing power. We used the Knuth machine of Harvey Mudd College (64 cores, 500GB of RAM).

Running and Testing

To Run the program execute the following command

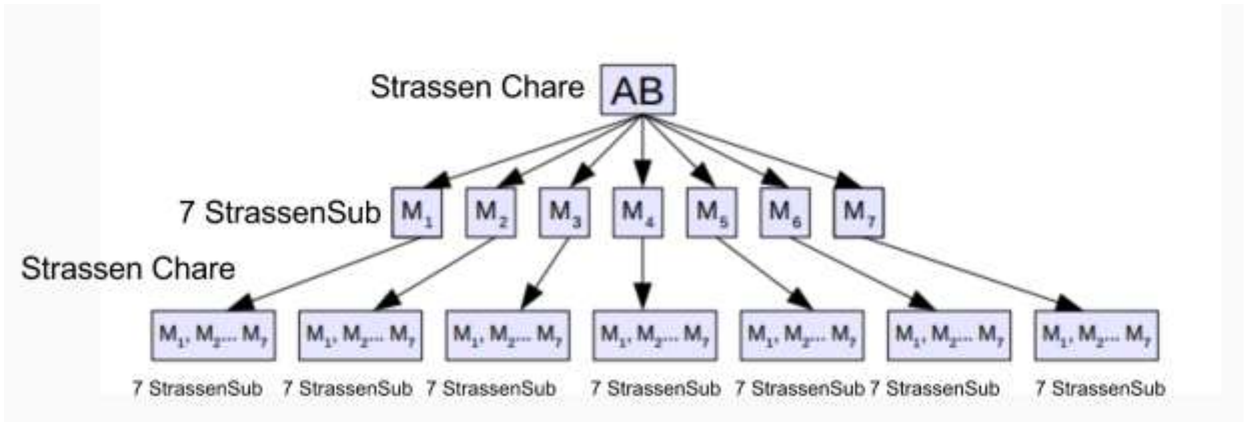
```
./charmrun ./strassen (size) (threshold) (verbosity) +p(# of procs)
```

The output line:

```
Charm++ - Correct: matrix size = 1024, Threshold = 32,# of proc = 2 , Exec time = 11.5419
```

Design Structure of Strassen Algorithm using Charm++

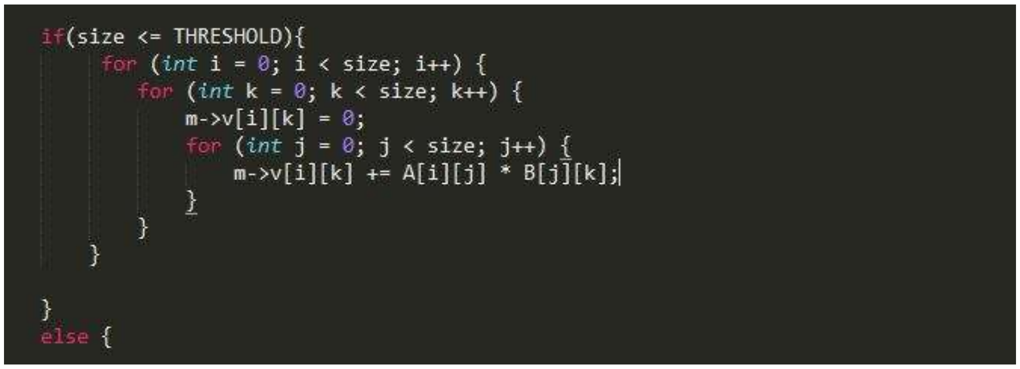
- Strassen Chare fires the StrassenSub to compute the 7 M partitions
- StrassenSub Chare fires Addittition Chare and Substraction Chare in each partition in parrallel
- StrassenSub fires a smaller in size Strassen Chare to perform Multiplication on the result of Addition/Substraction



Implementation

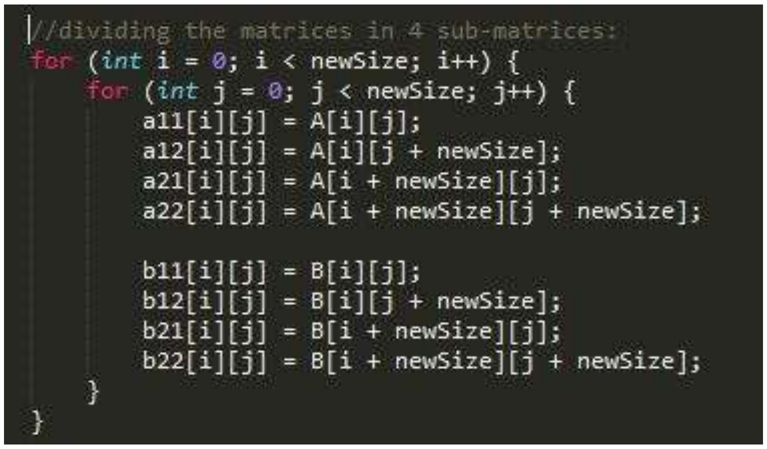
- ***Main Class***
  https://github.com/alielabridi/Strassen-Algorithm-parallelization-charm-/blob/master/strassen.C
  1. Generates a matrix of 1's and an identitiy matrix
  2. Provides Strassen future with the two matrices
  3. Records the waiting time for the Strassen future Matrix computation
  4. Test the resulting matrix return (check if it is a matrix of 1's)
- **Strassen Chare Class**
  https://github.com/alielabridi/Strassen-Algorithm-parallelization-charm-/blob/master/strassen.C

  1. Check if size below a certain threshold to run it naively otherwise using Strassen



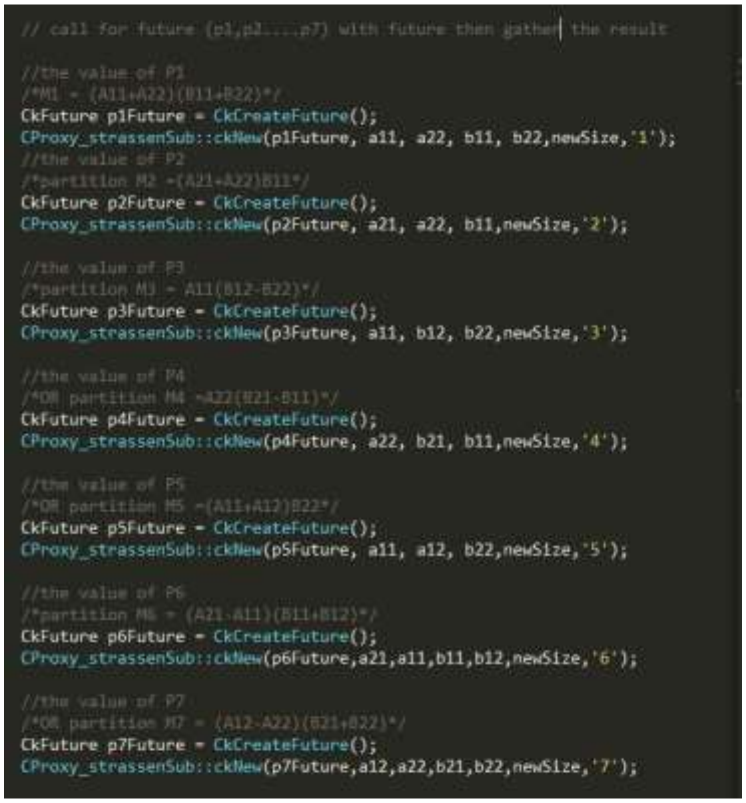
```
if(size <= THRESHOLD){
    for (int i = 0; i < size; i++) {
        for (int k = 0; k < size; k++) {
            m->v[i][k] = 0;
            for (int j = 0; j < size; j++) {
                m->v[i][k] += A[i][j] * B[j][k];
            }
        }
    }

}
else {
```

  2. If not naively, Partition the two A and B matrices into 4 partitions each (A11,A12,…B11,…B22)



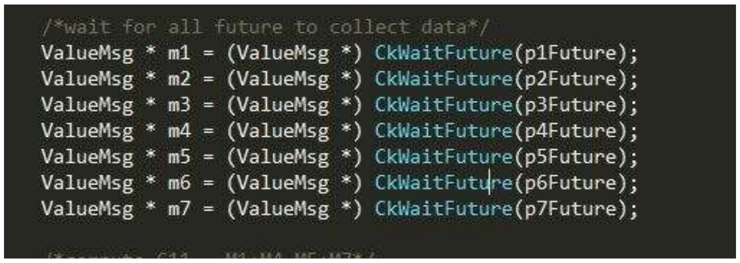
```
//dividing the matrices in 4 sub-matrices:
for (int i = 0; i < newSize; i++) {
    for (int j = 0; j < newSize; j++) {
        a11[i][j] = A[i][j];
        a12[i][j] = A[i][j + newSize];
        a21[i][j] = A[i + newSize][j];
        a22[i][j] = A[i + newSize][j + newSize];

        b11[i][j] = B[i][j];
        b12[i][j] = B[i][j + newSize];
        b21[i][j] = B[i + newSize][j];
        b22[i][j] = B[i + newSize][j + newSize];
    }
}
```

3. Fires the 7 sub-matrices with strassenSub future



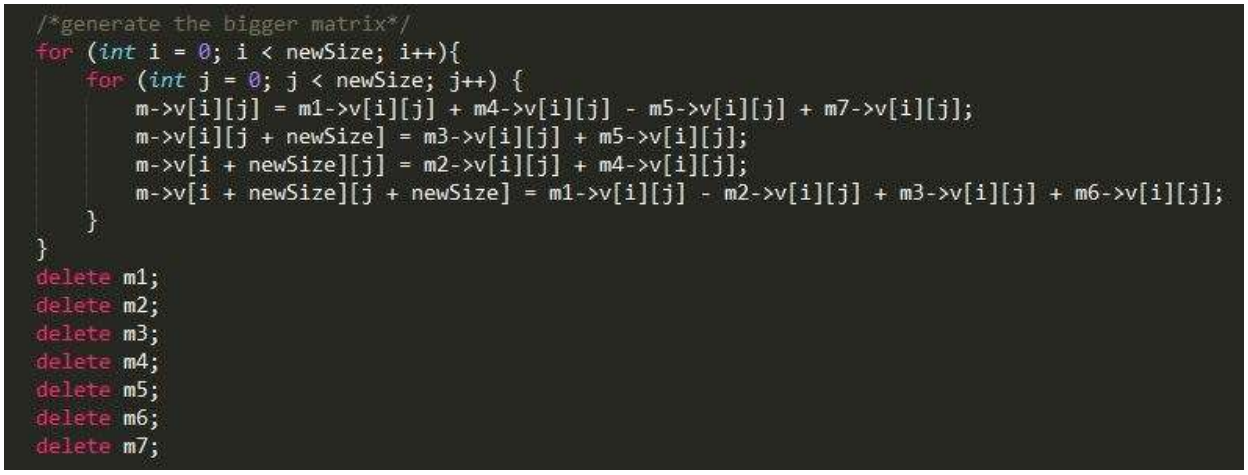
```
// call for future (p1,p2....p7) with future then gather the result

//the value of P1
/*M1 = (A11+A22)(B11+B22)*/
CkFuture p1Future = CkCreateFuture();
CProxy_strassenSub::ckNew(p1Future, a11, a22, b11, b22,newSize,'1');
//the value of P2
/*partition M2 =(A21+A22)B11*/
CkFuture p2Future = CkCreateFuture();
CProxy_strassenSub::ckNew(p2Future, a21, a22, b11,newSize,'2');

//the value of P3
/*partition M3 = A11(B12-B22)*/
CkFuture p3Future = CkCreateFuture();
CProxy_strassenSub::ckNew(p3Future, a11, b12, b22,newSize,'3');

//the value of P4
/*OR partition M4 =A22(B21-B11)*/
CkFuture p4Future = CkCreateFuture();
CProxy_strassenSub::ckNew(p4Future, a22, b21, b11,newSize,'4');

//the value of P5
/*OR partition M5 =(A11+A12)B22*/
CkFuture p5Future = CkCreateFuture();
CProxy_strassenSub::ckNew(p5Future, a11, a12, b22,newSize,'5');

//the value of P6
/*partition M6 = (A21-A11)(B11+B12)*/
CkFuture p6Future = CkCreateFuture();
CProxy_strassenSub::ckNew(p6Future,a21,a11,b11,b12,newSize,'6');

//the value of P7
/*OR partition M7 = (A12-A22)(B21+B22)*/
CkFuture p7Future = CkCreateFuture();
CProxy_strassenSub::ckNew(p7Future,a12,a22,b21,b22,newSize,'7');
```

4. Wait for the different futures of the 7 sub matrices



```
/*wait for all future to collect data*/
ValueMsg * m1 = (ValueMsg *) CkWaitFuture(p1Future);
ValueMsg * m2 = (ValueMsg *) CkWaitFuture(p2Future);
ValueMsg * m3 = (ValueMsg *) CkWaitFuture(p3Future);
ValueMsg * m4 = (ValueMsg *) CkWaitFuture(p4Future);
ValueMsg * m5 = (ValueMsg *) CkWaitFuture(p5Future);
ValueMsg * m6 = (ValueMsg *) CkWaitFuture(p6Future);
ValueMsg * m7 = (ValueMsg *) CkWaitFuture(p7Future);
```

5. Use the 7 sub matrices (n/2) to form the size (n) matrix

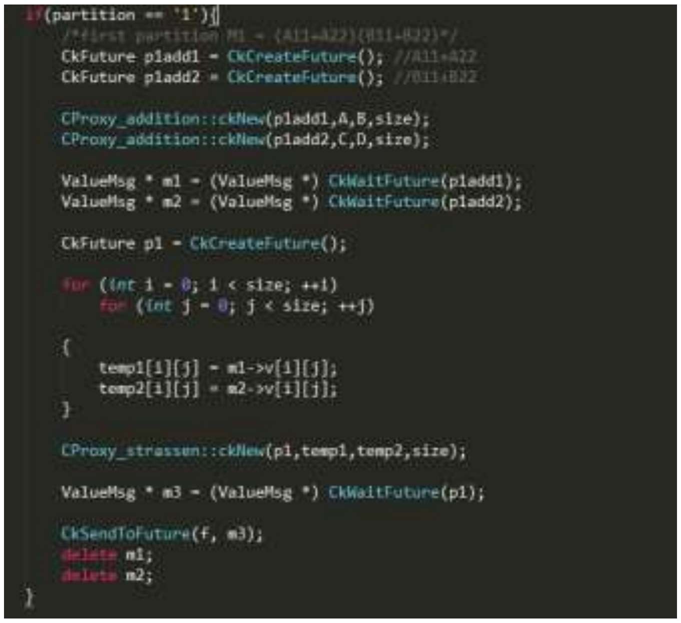


```
/*generate the bigger matrix*/
for (int i = 0; i < newSize; i++){
    for (int j = 0; j < newSize; j++) {
        m->v[i][j] = m1->v[i][j] + m4->v[i][j] - m5->v[i][j] + m7->v[i][j];
        m->v[i][j + newSize] = m3->v[i][j] + m5->v[i][j];
        m->v[i + newSize][j] = m2->v[i][j] + m4->v[i][j];
        m->v[i + newSize][j + newSize] = m1->v[i][j] - m2->v[i][j] + m3->v[i][j] + m6->v[i][j];
    }
}
delete m1;
delete m2;
delete m3;
delete m4;
delete m5;
delete m6;
delete m7;
```

6. Send the result back to the future using a message of size n

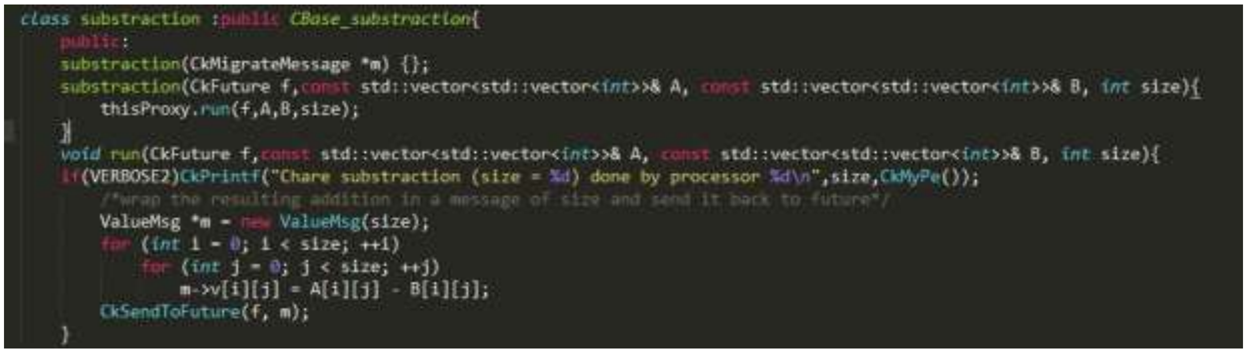
- **StrassenSub Chare Class**
  https://github.com/alielabridi/Strassen-Algorithm-parallelization-charm-/blob/master/StrassenSub.h

  Depending on which of the 7 sub matrices, it will run the appropriate formula by calling addition or subtraction chares, and wrap the result in a message and send it back to the future



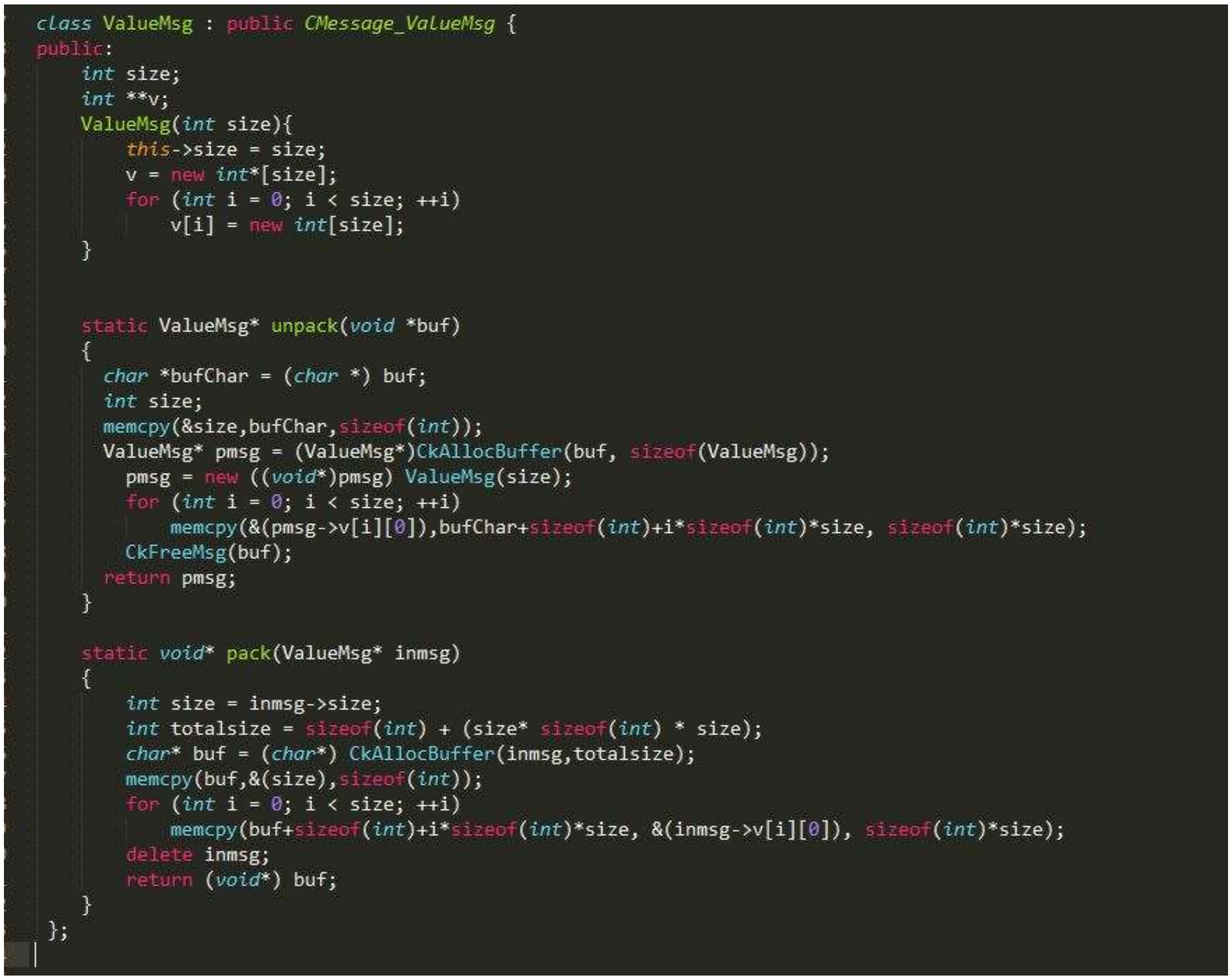
```
if(partition == '1'){
    /*first partition M1 = (A11+A22)(B11+B22)*/
    CkFuture p1add1 = CkCreateFuture(); //A11+A22
    CkFuture p1add2 = CkCreateFuture(); //B11+B22

    CProxy_addition::ckNew(p1add1,A,B,size);
    CProxy_addition::ckNew(p1add2,C,D,size);

    ValueMsg * m1 = (ValueMsg *) CkWaitFuture(p1add1);
    ValueMsg * m2 = (ValueMsg *) CkWaitFuture(p1add2);

    CkFuture p1 = CkCreateFuture();

    for (int i = 0; i < size; ++i)
        for (int j = 0; j < size; ++j)

    {
        temp1[i][j] = m1->v[i][j];
        temp2[i][j] = m2->v[i][j];
    }

    CProxy_strassen::ckNew(p1,temp1,temp2,size);

    ValueMsg * m3 = (ValueMsg *) CkWaitFuture(p1);

    CkSendToFuture(f, m3);
    delete m1;
    delete m2;
}
```

- **Addition/Subtraction Chare class**
  https://github.com/alielabridi/Strassen-Algorithm-parallelization-charm-/blob/master/AddSubMat.h
  Perform the addition or subtraction of two matrices, and wrap the result in a message of size n and send it back to the future



```
class substraction :public CBase_substraction{
    public:
    substraction(CkMigrateMessage *m) {};
    substraction(CkFuture f,const std::vector<std::vector<int>>& A, const std::vector<std::vector<int>>& B, int size){
        thisProxy.run(f,A,B,size);
    }
    void run(CkFuture f,const std::vector<std::vector<int>>& A, const std::vector<std::vector<int>>& B, int size){
    if(VERBOSE2)CkPrintf("Chare substraction (size = %d) done by processor %d\n",size,CkMyPe());
        /*wrap the resulting addition in a message of size and send it back to future*/
        ValueMsg *m = new ValueMsg(size);
        for (int i = 0; i < size; ++i)
            for (int j = 0; j < size; ++j)
                m->v[i][j] = A[i][j] - B[i][j];
        CkSendToFuture(f, m);
    }
```

- ***Message Class***
  https://github.com/alielabridi/Strassen-Algorithm-parallelization-charm-/blob/master/ValueMsg.h
  The message is dynamically allocated at run time. It also contains the definition of the static methods: packing & unpacking. The purpose of packing and unpacking is to be able to move the message across multiple processors by marshalling it and reconstructing it in another memory space.



```
class ValueMsg : public CMessage_ValueMsg {
public:
    int size;
    int **v;
    ValueMsg(int size){
        this->size = size;
        v = new int*[size];
        for (int i = 0; i < size; ++i)
            v[i] = new int[size];
    }


    static ValueMsg* unpack(void *buf)
    {
      char *bufChar = (char *) buf;
      int size;
      memcpy(&size,bufChar,sizeof(int));
      ValueMsg* pmsg = (ValueMsg*)CkAllocBuffer(buf, sizeof(ValueMsg));
        pmsg = new ((void*)pmsg) ValueMsg(size);
        for (int i = 0; i < size; ++i)
            memcpy(&(pmsg->v[i][0]),bufChar+sizeof(int)+i*sizeof(int)*size, sizeof(int)*size);
        CkFreeMsg(buf);
      return pmsg;
    }

    static void* pack(ValueMsg* inmsg)
    {
        int size = inmsg->size;
        int totalsize = sizeof(int) + (size* sizeof(int) * size);
        char* buf = (char*) CkAllocBuffer(inmsg,totalsize);
        memcpy(buf,&(size),sizeof(int));
        for (int i = 0; i < size; ++i)
            memcpy(buf+sizeof(int)+i*sizeof(int)*size, &(inmsg->v[i][0]), sizeof(int)*size);
        delete inmsg;
        return (void*) buf;
    }
 };
```

Interpretation of results

Analysis

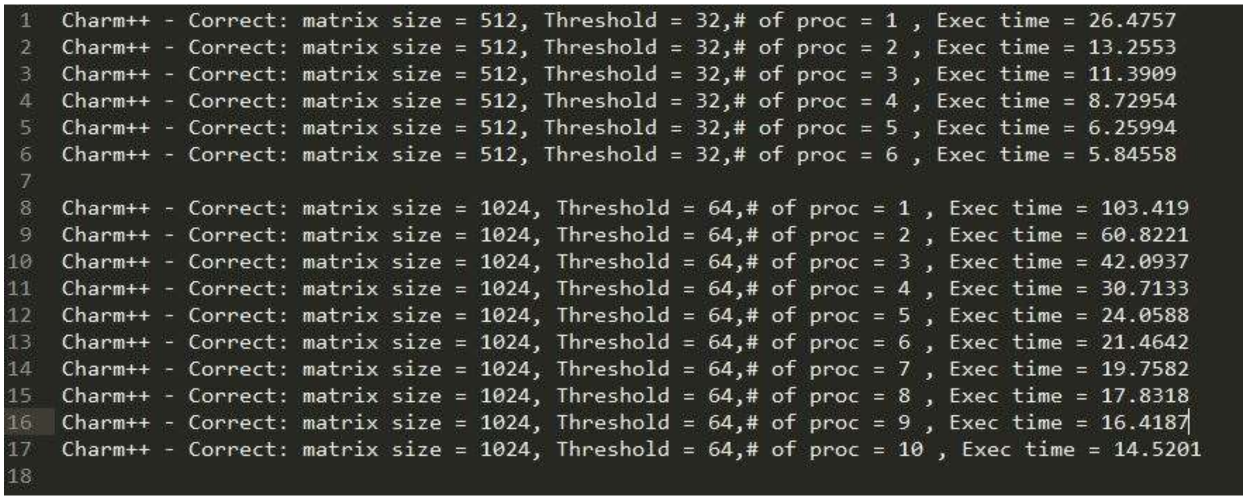
The Strassen algorithm has been run with different combination of matrix size, threshold and numbers of processors (processing units). These data are gathering in the following link:
https://github.com/alielabridi/Strassen-Algorithm-parallelization-charm-/blob/master/results.txt

Key aspects

- Proving that Strassen Algorithm of Matrix Multiplication is perfectly parallel (Not rigorous, but getting the right data)
- Necessity of a threshold to switch to the Naive Algorithm (Hybrid Strassen)
- Comparing paradigms: OpenMP vs Charm++ results

Different numbers of processors

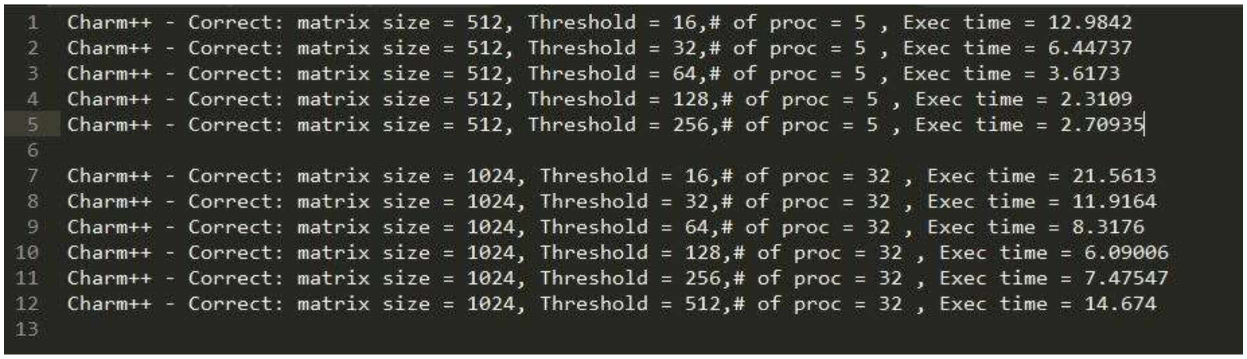
From the data, we can observe that the first two lines of each matrix size has its execution time dropped by two as the number of processors doubled. However, as we increase the number of processors in each case, we can observe that the linear correlation between numbers of processors and executions time is disturbed as the overhead of the parallelization takes over it.



```
Charm++ - Correct: matrix size = 512, Threshold = 32,# of proc = 1 , Exec time = 26.4757
Charm++ - Correct: matrix size = 512, Threshold = 32,# of proc = 2 , Exec time = 13.2553
Charm++ - Correct: matrix size = 512, Threshold = 32,# of proc = 3 , Exec time = 11.3909
Charm++ - Correct: matrix size = 512, Threshold = 32,# of proc = 4 , Exec time = 8.72954
Charm++ - Correct: matrix size = 512, Threshold = 32,# of proc = 5 , Exec time = 6.25994
Charm++ - Correct: matrix size = 512, Threshold = 32,# of proc = 6 , Exec time = 5.84558

Charm++ - Correct: matrix size = 1024, Threshold = 64,# of proc = 1 , Exec time = 103.419
Charm++ - Correct: matrix size = 1024, Threshold = 64,# of proc = 2 , Exec time = 60.8221
Charm++ - Correct: matrix size = 1024, Threshold = 64,# of proc = 3 , Exec time = 42.0937
Charm++ - Correct: matrix size = 1024, Threshold = 64,# of proc = 4 , Exec time = 30.7133
Charm++ - Correct: matrix size = 1024, Threshold = 64,# of proc = 5 , Exec time = 24.0588
Charm++ - Correct: matrix size = 1024, Threshold = 64,# of proc = 6 , Exec time = 21.4642
Charm++ - Correct: matrix size = 1024, Threshold = 64,# of proc = 7 , Exec time = 19.7582
Charm++ - Correct: matrix size = 1024, Threshold = 64,# of proc = 8 , Exec time = 17.8318
Charm++ - Correct: matrix size = 1024, Threshold = 64,# of proc = 9 , Exec time = 16.4187
Charm++ - Correct: matrix size = 1024, Threshold = 64,# of proc = 10 , Exec time = 14.5201
```

Different thresholds:

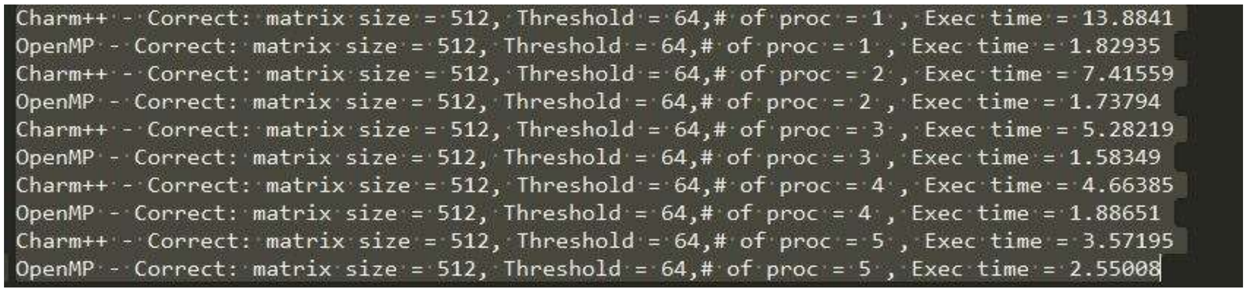
As we can see from the data, the threshold impacts heavily the execution time as it went in the computation of matrix size 512 from a 16 to 32 threshold dropped the execution time by two and so on for threshold 64. However, the benefits of increasing the threshold stopped in 128. The idea is that as we increase the threshold we limit the number of recursions, and thus the overhead of parallelization, but we can hit a limit where the difference between the naïve approach and the Strassen approach kick in.



```
Charm++ - Correct: matrix size = 512, Threshold = 16,# of proc = 5 , Exec time = 12.9842
Charm++ - Correct: matrix size = 512, Threshold = 32,# of proc = 5 , Exec time = 6.44737
Charm++ - Correct: matrix size = 512, Threshold = 64,# of proc = 5 , Exec time = 3.6173
Charm++ - Correct: matrix size = 512, Threshold = 128,# of proc = 5 , Exec time = 2.3109
Charm++ - Correct: matrix size = 512, Threshold = 256,# of proc = 5 , Exec time = 2.70935

Charm++ - Correct: matrix size = 1024, Threshold = 16,# of proc = 32 , Exec time = 21.5613
Charm++ - Correct: matrix size = 1024, Threshold = 32,# of proc = 32 , Exec time = 11.9164
Charm++ - Correct: matrix size = 1024, Threshold = 64,# of proc = 32 , Exec time = 8.3176
Charm++ - Correct: matrix size = 1024, Threshold = 128,# of proc = 32 , Exec time = 6.09006
Charm++ - Correct: matrix size = 1024, Threshold = 256,# of proc = 32 , Exec time = 7.47547
Charm++ - Correct: matrix size = 1024, Threshold = 512,# of proc = 32 , Exec time = 14.674
```

Comparing two different paradigms (OpenMP vs Strassen)

An attempt has been made to compare two different paradigms, but failed as the OpenMP version couldn't limit itself to the number of processors, and seemed to take advantage of the whole number of processors in the server. This will be explored further in the future work.



```
Charm++ - Correct: matrix size = 512, Threshold = 64,# of proc = 1 , Exec time = 13.8841
OpenMP - Correct: matrix size = 512, Threshold = 64,# of proc = 1 , Exec time = 1.82935
Charm++ - Correct: matrix size = 512, Threshold = 64,# of proc = 2 , Exec time = 7.41559
OpenMP - Correct: matrix size = 512, Threshold = 64,# of proc = 2 , Exec time = 1.73794
Charm++ - Correct: matrix size = 512, Threshold = 64,# of proc = 3 , Exec time = 5.28219
OpenMP - Correct: matrix size = 512, Threshold = 64,# of proc = 3 , Exec time = 1.58349
Charm++ - Correct: matrix size = 512, Threshold = 64,# of proc = 4 , Exec time = 4.66385
OpenMP - Correct: matrix size = 512, Threshold = 64,# of proc = 4 , Exec time = 1.88651
Charm++ - Correct: matrix size = 512, Threshold = 64,# of proc = 5 , Exec time = 3.57195
OpenMP - Correct: matrix size = 512, Threshold = 64,# of proc = 5 , Exec time = 2.55008
```

Future Work

- More rigorous testing with larger sizes and more hardware resources
- Compare it to more parallelization paradigms
- Deeper analysis of the fundamental difference with other paradigm, and draw conclusion on what kind of problem is suited for Charm++
- Submission of the code to the Charm++ as to be added to the examples repository

Conclusions

The applied research of exploring the realm of parallelization using Strassen algorithm and charm++ was extremely interesting. I had the chance to deepen my understanding of it, and explore new idea and new ways of apprehending parallelization. Moreover, the biggest challenge was the Message passing. I spent countless hours trying to understand how it works, and the lack of documentation and examples from Charm++ certainly did not help. I am willing to explore further Charm++ as it ended up being an extremely interesting paradigm to work on for future projects.

References

J. Choi. A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. Concurrency: Practice and Experience, 10(8):655–670, 1998

D. Coppersmith and S. Winograd. On the asymptotic complexity of matrix multiplication. SIAM Journal on Computing, 11(3):472–492, 1982.

B. Grayson, A. Shah, and R. van de Geijn. A high performance parallel Strassen implementation. In Parallel Processing Letters, volume 6, pages 3–12, 1995

D. Coppersmith and S. Winograd. On the asymptotic complexity of matrix multiplication. SIAM Journal on Computing, 11(3):472–492, 1982.

G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds, 2012. Manuscript, submitted to SPAA 2012.