Current Topics in Multimedia Communication: Server, Cluster, and Cloud Computing

Giovanni Liva

May 9, 2016

1 Introduction

All the code was written in C/C++ using only the standard libraries. The compiler used is *clang* version `703.0.31` on OSX `10.10.4`. In order to compile the project, the only operation needed is to run the `make` command in the directory that contains the `Makefile` file. The arguments accepted by the program are the following:

- -row [n] : set the number of rows to n for matrices A, B and C.
- -col [n] : set the number of columns to n for matrices A, B and C.
- -fill [n] : set the filling percentage to n for matrices A and B.
- -t [n] : set the number of thread to use to n, the default value is 1.
- -d : allows the debug mode. This modality prints in the standard output some debug info and print the matrices in csv files.
- -s [n] : set the strategy to use to compute the addiction of the matrices to n, the default value is 0. The possible values are: *0, 1, 2, 3*. Further details about the strategies are given in Section 2.
- -f [filename] : store data about the computation in the specified file in csv format. If the file already exists, the data will be appended to the current one. This parameter is used to store the data used in Section 2.1.

The first two parameters are mandatory. All the tests were done on the following hardware:

- processor: 2,5 GHz Intel Core i7-4870HQ [1]
- number of cores: 4
- number of threads: 8
- ram: 16 GB

[1] http://ark.intel.com/products/83504/Intel-Core-i7-4870HQ-Processor-6M-Cache-up-to-3_70-GHz

2 Implementation

The project was created in two C/C++ files with their own headers: `Main.cpp` and `YSMF.cpp`. In the following section we will discuss the relevant parts of code of these two files. As a requirement for this project, the `pthread` library is used instead of the `std::thread` library.

YSFM.cpp The implementation of the YSMF is coded as C++ class. The whole code is straightforward since it implements some getters to the three vectors that store the information in the Yale Sparse Matrix Format. The filling strategy uses a random distribution to decide where to store a integer that is a randomly generated in the interval $[1, 9]$. The random distribution tends to store more data in the last rows of the matrix. The interesting part of code of this class are following two methods.

```
void YSMF::addElement(int elm, int row, int col) {
    if(elm == 0) return;
    A.push_back(elm);
    JA.push_back(col);
    for(int i = row+1; i <= _nRows; i++){
        IA[i] += 1;
    }
    NNZ++;
}
```

```
void YSMF::addElementThread(int elm, int row, int col, int nElmSkip){
    if(elm == 0) return;
    //create memory for new elements
    if (nElmSkip >= A.size()) {
        A.resize(nElmSkip+1);
    }
    if (nElmSkip >= JA.size()) {
        JA.resize(nElmSkip+1);
    }
    A.insert(A.begin() + nElmSkip, elm);
    JA.insert(JA.begin() + nElmSkip, col);
    for(int i = row+1; i <= _nRows; i++){
        IA[i] += 1;
    }
    NNZ++;
}
```

The `YSMF::addElement` method insert an element in the position specified in the parameters. Since the `YSMF` uses an encoding left to right, top to bottom of the elements, we have to implement a different method if we want to allow an insertion that does not respect that order. This method is mandatory since we would like to compute sum of matrices in parallel. For this reason the `YSMF::addElementThread` method has a parameter that specifies where in A and JA we will insert the new element.

Main.cpp This file, as the name suggests, implements the main function and the logic to parse the parameters and to spawn the threads. After parsing the input parameters, it creates the data to pass to each thread. We refer to this phase of creation of data as *init phase of threads*. After the *init phase* it creates the threads using the specified strategy and then wait until all of them end their computation.

2.1 Strategies

We look at the Yale Sparse Matrix Format and we decided that the computation of the addiction of the two matrices can be performed in different ways. *Strategy 0* and *Strategy 1* use a naïve approach. They go through all the matrices position and check if there is an element stored in that particular position. *Strategy 2* instead pre-process the two matrices in order to go through only the elements stored. Further, if we divide the workload of threads in rows of C to compute, we can avoid to use synchronization method because there are only concurrent reads. *Strategy 1* and *3* implement this approach using a small local matrix to store the results. When all the threads join, all the local matrices are merged to compute the correct C matrix. In each strategy, we give an *id* to each thread. This *id* is unique and in the range $[0, \#thread - 1]$, where $\#thread$ is the number specified as parameter.

Init Phase Strategy 0 and 1 The creation of data for these two strategy is simple, we just have to specify the range of rows where the thread has to compute the sum and initialize the local matrix where to store the results for *Strategy 1*.

```
int batch_size = n_row / n_thread;
for(int i = 0; i < n_thread; i++){
    _data[i].id = i;
    _data[i].start = batch_size * i;
    _data[i].end = (i == n_thread-1) ? n_row :  _data[i].start + batch_size;
    _data[i].c = new YSMF(_data[i].end-_data[i].start, n_col);
}
```

Init Phase Strategy 2 and 3 These two strategies uses the same init phase of the previous ones plus some more work. We store the elements of A and B in two `std::map` data structure. The current pre-process of the data allows the *Strategy 2* and *3* to go through only the elements of A and B and skip the empty zones of the matrices. The workload of threads here is divided in the same manner of the previous strategies.

```
std::vector<std::pair<int, int>> coordinateA = a->getElmCoordinate();
std::vector<int> *aA = a->getA();
for(int i = 0; i < coordinateA.size(); i++){
    mapA[coordinateA[i].first][coordinateA[i].second] = aA->at(i);
}
std::vector<std::pair<int, int>> coordinate = b->getElmCoordinate();
std::vector<int> *bA = b->getA();
for(int i = 0; i < coordinate.size(); i++){
    mapB[coordinate[i].first][coordinate[i].second] = bA->at(i);
}
```

Strategy 0 The code of the strategy is the following:

```
struct threadData *data = (struct threadData *)arg;
int maxColB = b->getCols();
int sum ;
int startLine = data->start;
int endLine = data->end;
int NNZ = 0;
//work only on out batch size
for(int i = startLine; i < endLine; i++){
    for(int j = 0; j < maxColB; j++){
        sum = a->getElement(i,j) + b->getElement(i, j);
        if(sum > 0) {
            //check where store the data
            NNZ = 0;
            pthread_mutex_lock(&lock);
            for(int z = 0; z <= data->id; z++){
                NNZ += nElmWrite[z];
            }
            //write down the result
            c->addElementThread(sum, i, j, NNZ);
            nElmWrite[data->id]++;
            pthread_mutex_unlock(&lock);
        }

    }
}
```

It goes through all the rows and columns of A and B and compute the sum. The vector $nElmWrite$ is a global vector share between each thread that stores the information about how many elements that particular thread has compute. This is mandatory because we have to track how many element the threads with id lower than the id of the current one have written. This assumption of correctness of the algorithm holds since we give horizontal slice of the matrix C to compute to each thread and the slice have the same order of the ids of the threads. This allows the algorithm to fulfill the requirement to write elements in left to right, top to bottom order.

Strategy 1 The code of *Strategy 1* is the following:

```
struct threadData *data = (struct threadData *)arg;
int maxColB = b->getCols();
int sum ;
int startLine = data->start;
int endLine = data->end;
//work only on our batch size
for(int i = startLine; i < endLine; i++){
    for(int j = 0; j < maxColB; j++){
        sum = a->getElement(i,j) + b->getElement(i, j);
        if(sum > 0) {
            data->c->addElement(sum,i - startLine,j);
        }
    }
}
```

The code is similar to the one of the previous strategy. The only difference is that now we do not have any protect regions and we write locally the result of the sum. This strategy has to perform some work for merging the results after the join operation.

```
int sumElms = 0;
int pos = 0;
for(int i = 0; i < n_thread; i++){
    //merge results
    std::vector<int> *tA = _data[i].c->getA();
    std::vector<int> *tIA = _data[i].c->getIA();
    std::vector<int> *tJA = _data[i].c->getJA();
    cA->reserve(cA->size() + tA->size());
    cA->insert(cA->end(), tA->begin(), tA->end());
    cJA->reserve(cJA->size() + tJA->size());
    cJA->insert(cJA->end(), tJA->begin(), tJA->end());
    //full fill spaces
    int max = _data[i].c->getRows();
    for(int k = 1; k <= max; k++){
        cIA->at(++pos) = tIA->at(k) + sumElms;
    }
    sumElms += tIA->at(max);
}
```

Vector A and JA are computed append the ones of each thread. Regarding IA instead we have also to remember the number of element that each thread compute.

Strategy 2 The code of *Strategy 2* is the following:

```
struct threadData *data = (struct threadData *)arg;
int start = data->start;
int end = data->end;
int maxColB = b->getCols();
int row,col,sum;
for (row = start; row < end; row++) {
    if(mapA.count(row) < 1 && mapB.count(row) < 1){
        continue;
    }
    //for each col of B
    for(col = 0; col < maxColB; col++){
        sum = 0;
        if(mapA.count(row) > 0 && mapA[row].count(col) > 0){
            sum += mapA[row][col];
        }
        if(mapB.count(row) > 0 && mapB[row].count(col) > 0){
            sum += mapB[row][col];
        }
        if(sum > 0){
            int z = 0;
            int NNZ = 0;
            pthread_mutex_lock(&lock);
            for(; z <= data->id; z++){
                NNZ += nElmWrite[z];
            }
            //write down the result
            c->addElementThread(sum, row, col, NNZ);
            nElmWrite[data->id]++;
            pthread_mutex_unlock(&lock);
        }
    }
}
```

It is similar to the first strategy, except for the using of the map. The motivation for introducing such strategy is given looking through the code of `YSMF::getElement`.

```
int YSMF::getElement(int i, int j){
    int d = IA[i+1] - IA[i];
    if(d > 0){
        std::pair<std::vector<int>, std::vector<int>> row = getRow(i);
        for(int k = 0; k < row.first.size(); k++){
            if(j == row.second[k])
            return row.first[k];
        }
    }
    return 0;
}
```

Since in the previous strategies call it many times it makes the code go through the for many times and for each row always looking just one more element of the column. To avoid this, we decided to pre-process the matrices in order to do the work only once.

Strategy 3 The code of *Strategy 3* is the following:

```
struct threadData *data = (struct threadData *)arg;
int start = data->start;
int end = data->end;
int maxColB = b->getCols();
int row,col,sum;
for (row = start; row < end; row++) {
    if(mapA.count(row) < 1 && mapB.count(row) < 1){
        continue;
    }
    //for each col of B
    for(col = 0; col < maxColB; col++){
        sum = 0;
        if(mapA.count(row) > 0 && mapA[row].count(col) > 0){
            sum += mapA[row][col];
        }
        if(mapB.count(row) > 0 && mapB[row].count(col) > 0){
            sum += mapB[row][col];
        }
        if(sum > 0){
            data->c->addElement(sum,row - start,col);
        }
    }
}
```

It combines all the purpose of *Strategy 2* with the idea of using a local matrix to avoid synchronization of *Strategy 1*.

3 Evaluation

All the data is computed running the `./evaluation.sh` script. That script is divided in three sections and each one computes only square matrices. In the first part we execute the program multiple time varying the number of rows and columns using only one thread. In the second section the size of the matrix is fixed to 3000×3000, but we change the number of threads to use, from 2 to 80 with a step of 2. In the latter part, we execute strategy number one and three in the same settings of the second evaluation but with a matrix size of 15000×15000. The last evaluation was necessary to completely check how much time with *Strategy 3* the *init phase* of the data takes with a matrix that requires some time to be process.

Single Thread Figure 1 shows the time necessary to compute the addiction of the matrices for all the four strategies. The graph confirms out expectation, pre-process the data with *Strategy 2* and *3* reduces the time to compute the addiction. Moreover, it confirms that using locking strategies or not, does not impact the result since there is not any race to a common resource.

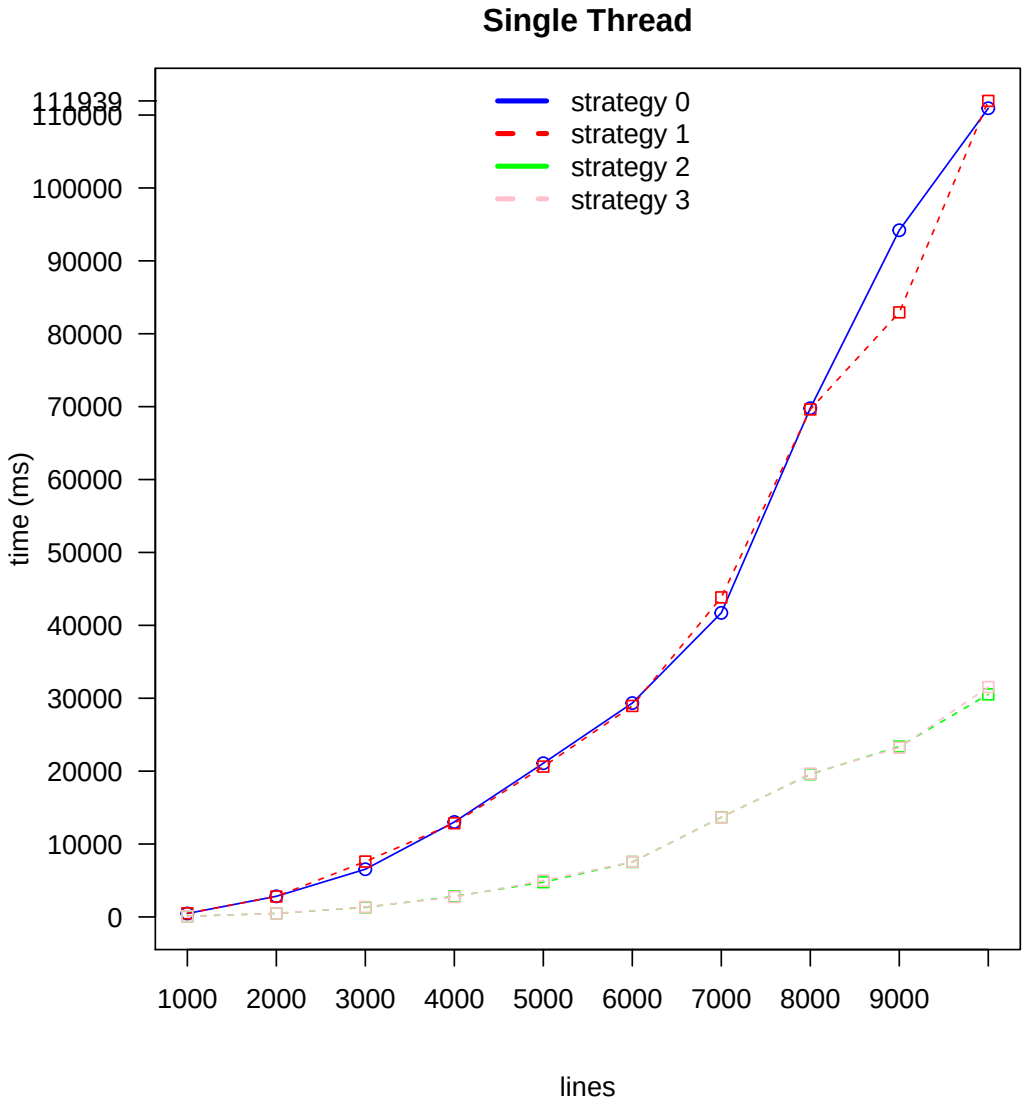


Figure 1: Time to compute the addiction with a single thread.

Multiple Thread Figure 2 presents some behaviors that we do not await. All the strategies compute a matrix 3000×3000 with different number of threads. If we compare the result for each strategy in a singular thread, *Strategy 0* and *2* perform always worst with multi-thread. We do not foresee such behavior and we dig into the code to find some bugs that can cause the reduction of performance, but we did not manage to discover anything. Nevertheless, the Figure shows a confirmed that increasing too much the number of threads we decrease the performances. This attitude does not appear in *Strategy 1* and *3* because the matrix to compute is too small.

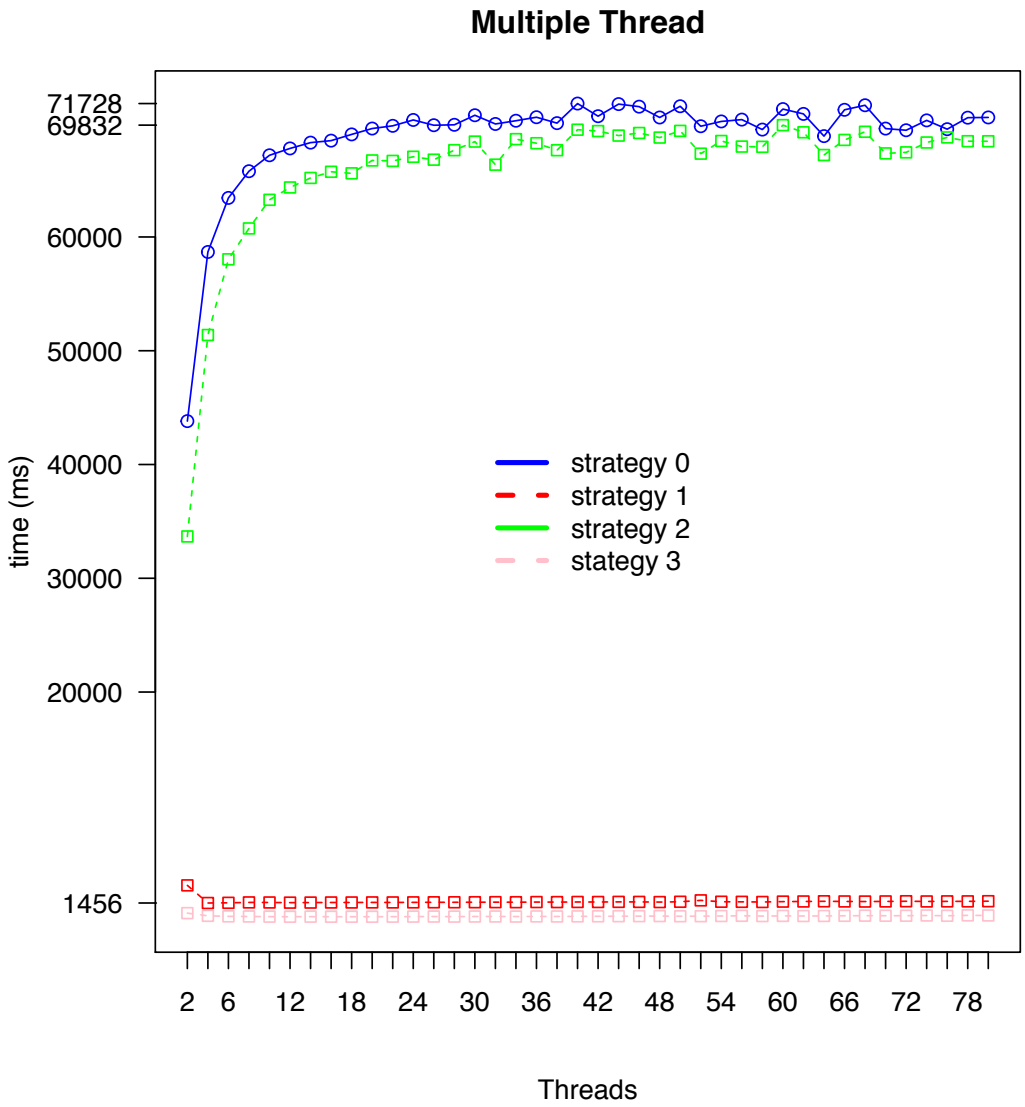


Figure 2: Time required to compute a matrix 3000×3000 with different number of threads.

Speed Up We compute the P value comparing the results in multiple-threading evaluation with the equivalent in single-thread settings in terms of matrix size. Figure 3 presents the values. As we expected, for *Strategy 1* and *3*, the P value is greater than 1 and increasing the number of threads to use, it slowly decreases. Instead, *Strategy 0* and *2* have a stable value around 0.018. This means that the two strategies are not suitable at all for multi-thread execution.

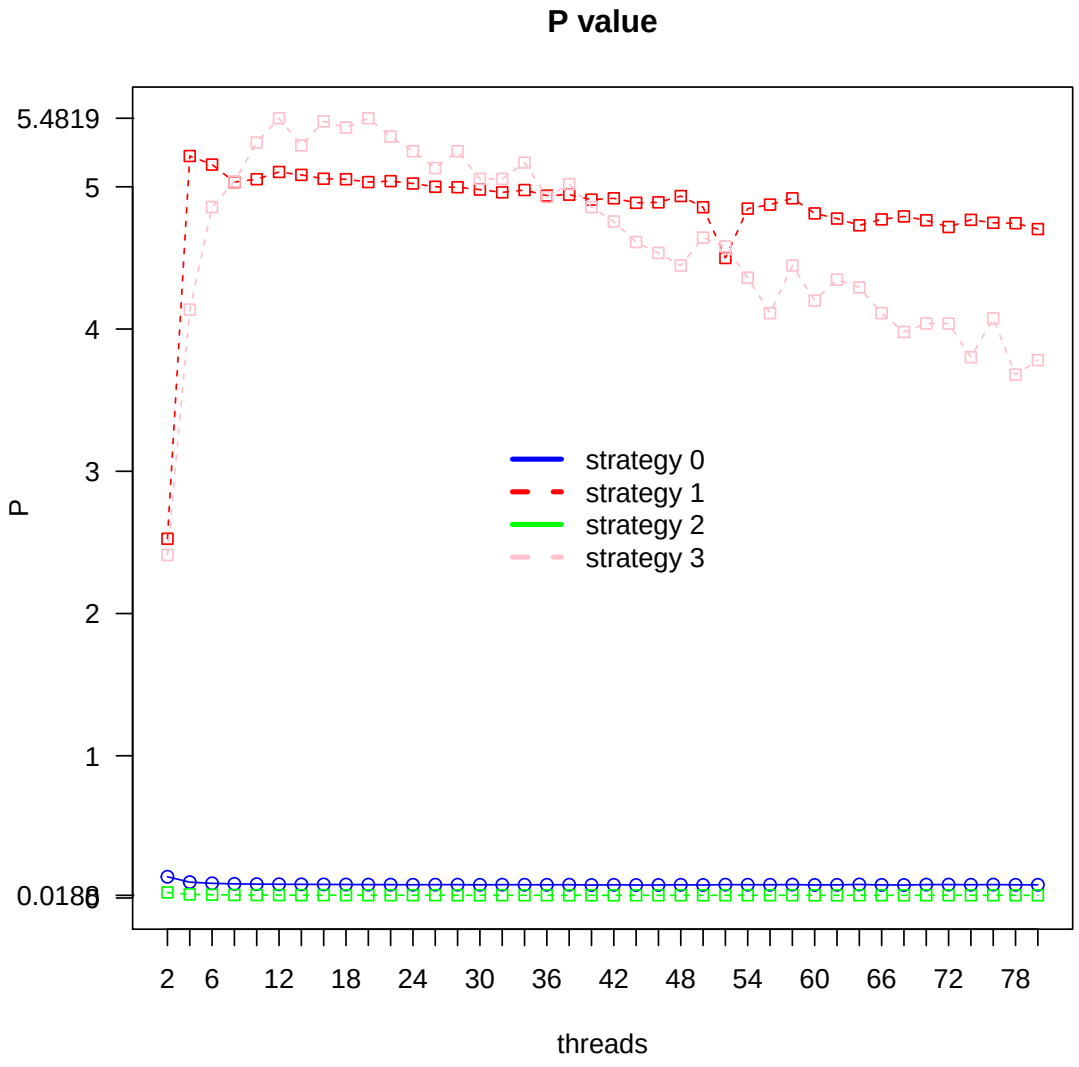
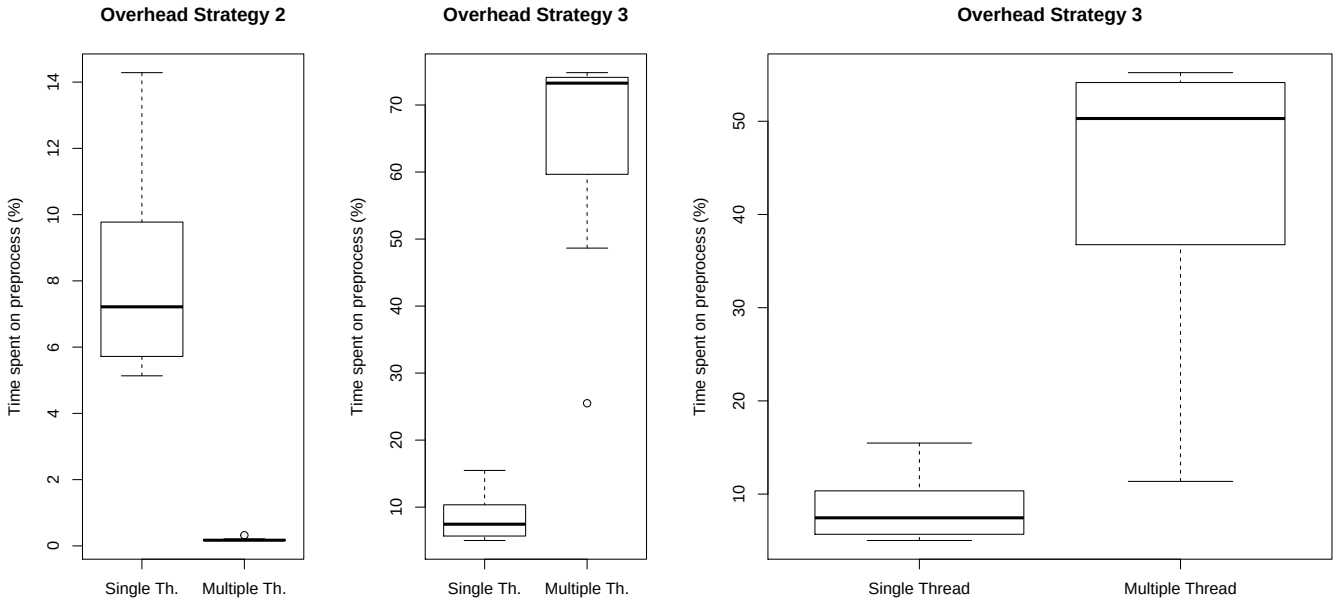


Figure 3: P value for each number of thread used.

Overhead of Init Phase Figure 4a displays how much time *Strategy 2* and *3* spend on pre-process the data in the map structure. *Strategy 2* spent in average roughly 7.5% of its time on the init phase for singular thread and almost 0 in multi-thread. This trend is confirmed looking to the graphs for its execution time in singular and multi thread. In the singular thread, *Strategy 2* has a good performance that decrease rapidly in the multi-thread experiment.

Regarding *Strategy 3*, it uses more than 70% on preparing the data in the multi-thread setting. This high value is not surprising because the size of the matrix is too small for *Strategy 3* to compute. Figure 4b shows the time spent on preparing the data for a matrix of 15000. Also in this setting the time spent on the init phase takes considerably amount of time, 50%, but shows that preparing the data for an efficient process of it improve the performance.



(a) Time spent of pre-process the data.

(b) Time spent of pre-process the data with bigger matrix.

Blow the Memory Up In order to check the size limit that our hardware can handle we manually run the program increasing the value of cols and rows in the parameters with the fill parameter set to 5%. The maximum value achieved is: $rows = 100000, cols = 1000000$. That means the program creates two matrices of 100000×1000000 with 5000000000 non zero elements. The total memory used is presented in Figure 5. If we had to store the two matrices as two dimensional array, it requires 1.6 TB of memory.

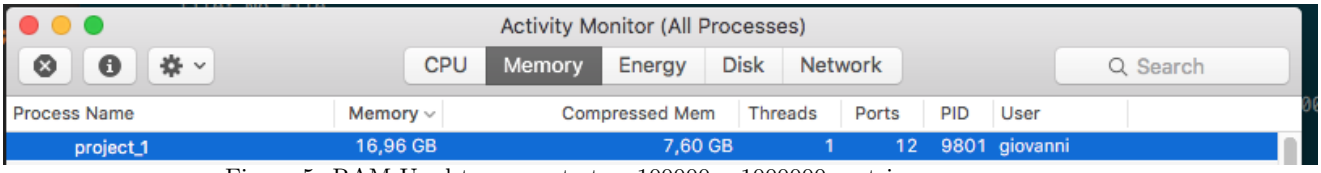


Figure 5: RAM Used to compute two 100000×1000000 matrices

Comparison of Best Strategies Since in Figure 2 is not possible to completely check the performance of *Strategy 1* and *3*, we run these two strategies with a bigger matrix. Figure 6 displays the expected fact of increasing the number of thread after a threshold, it decreases the performance.

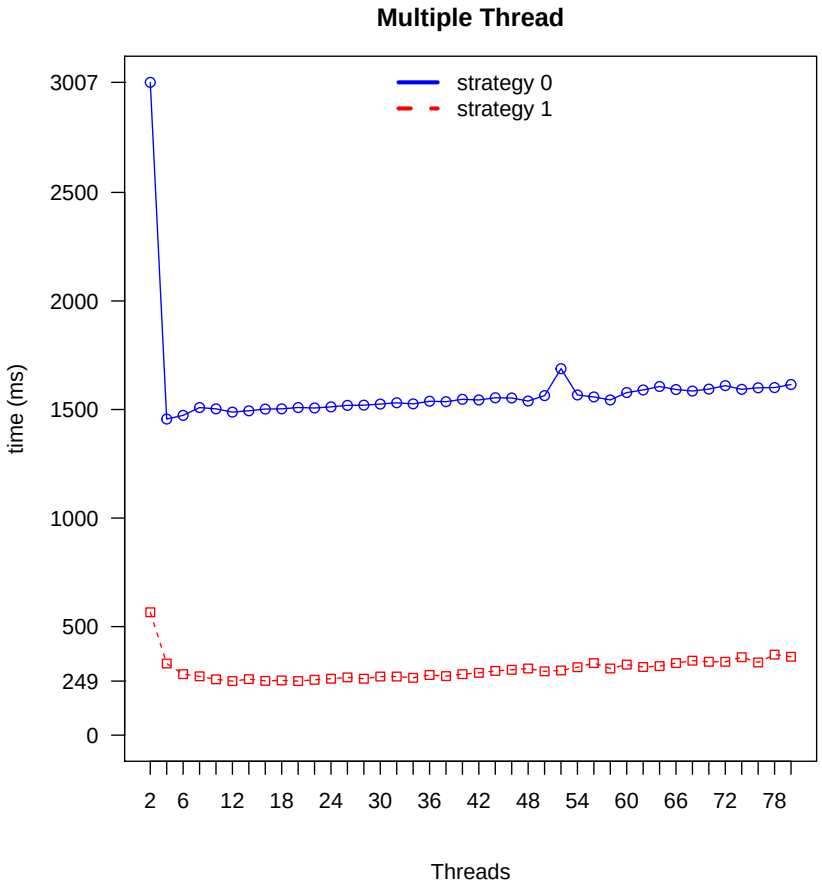


Figure 6: Time to compute addiction of 15000×15000 size matrices.