ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΕΠΕΞΕΡΓΑΣΤΩΝ & ΥΛΙΚΟΥ

ΗΡΥ 418 ΑΡΧΙΤΕΚΤΟΝΙΚΗ ΠΑΡΑΛΛΗΛΩΝ ΚΑΙ ΚΑΤΑΝΕΜΗΜΕΝΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΑΡΙΝΟ ΕΞΑΜΗΝΟ 2017-18

Άσκηση 2 : Παραλληλισμός με χρήση SIMD εντολών και MPI

ΟΜΑΔΑ : LAB41835573

ΟΝΟΜΑ : ΖΑΧΑΡΙΑ ΝΕΟΦΥΤΟΣ - 2014030210
ΣΦΥΡΗΣ ΝΕΚΤΑΡΙΟΣ - 2013030058

ΠΕΡΙΓΡΑΦΗ

Στο project αυτό μας ζητήθηκε να χρησιμοποιήσουμε Streaming SIMD Extensions (SSE) εντολές για εύρεση μέγιστης τιμής του ω statistic για καθορισμένο αριθμό DNA θέσεων. Στη συνέχεια χρησιμοποιήσαμε παραλληλισμό με MPI (Message Passing Interface) για P processes για την προηγούμενη υλοποίηση με SSE εντολές.

Streaming SIMD Extensions (SSE)

Έχοντας τον αρχικό κώδικα που υπολογίζει την μέγιστη τιμή του ω statistic δοσμένου του N από το χρήστη, τροποποιήσαμε τον κώδικα στην for loop που μας είχε υποδειχθεί με SSE εντολές των 128 bit. Αυτό σημαίνει ότι με κάθε πράξη μιας μεταβλητής 128 bits θα είχαμε αποτέλεσμα για τέσσερις 32 bit float values.

Αρχίζουμε ορίζοντας τις μεταβλητές που θα χρησιμοποιήσουμε και σετάροντας τις αρχικές τιμές όπου πρέπει.

```
float maxF = 0.0f;
double timeTotal = 0.0f;
__m128 mF=_mm_set_ps1(0.0f);
__m128 num_0,num_1,num_2,num,den_0,den_1,den;
__m128 t_1=_mm_set_ps1(1.0f);
__m128 t_2=_mm_set_ps1(2.0f);
__m128 t_3=_mm_set_ps1(0.01f);
__m128 FV;
__m128 mV,nV,RV,LV,CV;
float xx[4],x1,x2;

for(int j=0;j<iters;j++)
{

    double time0=gettime();

    for(int i=0;i<N;i+=4)
    {
        mV=_mm_load_ps(&mVec[i]);
        nV=_mm_load_ps(&nVec[i]);
        RV=_mm_load_ps(&RVec[i]);
        LV=_mm_load_ps(&LVec[i]);
        CV=_mm_load_ps(&CVec[i]);

        num_0=_mm_add_ps(LV,RV);
        num_1=_mm_div_ps(_mm_mul_ps(mV,_mm_sub_ps(mV,t_1)),t_2);
        num_2=_mm_div_ps(_mm_mul_ps(nV,_mm_sub_ps(nV,t_1)),t_2);
        num=_mm_div_ps(num_0,(_mm_add_ps(num_1,num_2)));
        den_0=_mm_sub_ps(CV,(_mm_add_ps(LV,RV)));
        den_1=_mm_mul_ps(mV,nV);
        den=_mm_div_ps(den_0,den_1);
        FV=_mm_div_ps(num,_mm_add_ps(den,t_3));
        mF =_mm_max_ps(FV,mF);

        if( (i+4>N) & (N%4!=0) )
        {
            for(int i=(N-N%4);i<N;i++)
            {
                float num_0 = LVec[i]+RVec[i];
                float num_1 = mVec[i]*(mVec[i]-1.0)/2.0;
                float num_2 = nVec[i]*(nVec[i]-1.0)/2.0;
                float num = num_0/(num_1+num_2);
                float den_0 = CVec[i]-LVec[i]-RVec[i];
                float den_1 = mVec[i]*nVec[i];
                float den = den_0/den_1;
                FVec[i] = num/(den+0.01);
                maxF = FVec[i]>maxF?FVec[i]:maxF;
            }
        }
    }
    double time1=gettime();
    timeTotal += time1-time0;
}
```

Το loop το οποίο εξετάζουμε θα έχει βήμα τέσσερα, καθώς για κάθε πέρασμα για τις 128 bit εντολές θα ελέγχουμε τέσσερα floats των 32 bit.

Μεταβλητές των 128 bit κρατάνε τα i, i+1, i+2 και i+3 στοιχεία των πινάκων που χρειαζόμαστε με την _mm_load_ps ώστε να τα χρησιμοποιήσουμε στις πράξεις παρακάτω. Μεταφράσαμε κάθε πράξη με τα instructions που πρέπει για να είναι συμβατά με SSE. Για την επιλογή του ποιας συνάρτησης χρειαζόμαστε για κάθε πράξη, κοιτάξαμε στο Intel Intrinsics Guide, και αν τα instructions τα οποία εκτελούσε η συνάρτηση συμβάδιζαν με τις εντολές που θέλουμε να κάνουμε εμείς, τότε την χρειαζόμασταν. Οι υπόλοιπες εντολές που αφορούν πράξεις, όπως η _mm_div_ps ενκτελούν τις πράξεις μεταξύ των 32 bit floats τα οποία βρίσκονται στις ίδιες θέσεις των 128 bit μεταβλητών. Η _mm_max_ps βρίσκει την max τιμή μεταξύ δύο 128 bit μεταβλητές και την κρατάει.

Η if προς το τέλος της δεύτερης for χρειάζεται για την περίπτωση που ο χρήστης δώσει N το οποίο να μην διαιρείται τέλεια με το 4 (N%4!=0). Στην περίπτωση αυτή μας έχουν μείνει λιγότερες ή ίσες από τρεις φορές που χρειάζεται να ξανακάνουμε τις πράξεις υπολογισμού του ω, αλλά επειδή είναι για τόσες λίγες φορές και μόνο μία φορά για κάθε iter δεν μας ενδιαφέρει να χρησιμοποιήσουμε SSE καθώς η διαφορά του να χρησιμοποιούσαμε θα ήταν αμυδρή.

Αφού στο τέλος έχουμε υπολογίσει το mF για το μέρος των επαναλήψεων που διαιρούνται τέλεια με το 4, και το maxF για ότι μας είχε απομείνει απο τις προηγούμενες επαναλήψεις, τα συγκρίνουμε μεταξύ τους για να βρούμε την μέγιστη τιμή του ω.

Αποτελέσματα

Τρέχοντας τον αρχικά δοσμένο κώδικα αλλά και τον κώδικα που φτιάξαμε “πειράζοντας” τον αρχικό με τις SSE εντολές, παίρνουμε τα παρακάτω αποτελέσματα για το ω statistic και για τον χρόνο που χρειάστηκαν να εκτελεστούν οι πράξεις για να βρεθεί.

```
nektar@nektar-PC:~/Desktop/proj2$ ./simple 1000
Time 0.000020 Max 121.787003
nektar@nektar-PC:~/Desktop/proj2$ ./sse 1000
Time 0.000012 Max 121.787010
nektar@nektar-PC:~/Desktop/proj2$ ./simple 10000
Time 0.000184 Max 510.145172
nektar@nektar-PC:~/Desktop/proj2$ ./sse 10000
Time 0.000086 Max 510.145325
nektar@nektar-PC:~/Desktop/proj2$ ./simple 100000
Time 0.001839 Max 8077.646484
nektar@nektar-PC:~/Desktop/proj2$ ./sse 100000
Time 0.000854 Max 8077.673340
```

Παρατηρούμε ότι για N=1000 έχουμε ταχύτερη εκτέλεση των εντολών κατά 0.000008 seconds, δηλαδή μείωση χρόνου κατά 40%.

Για N=10000 η μείωση είναι 53,26%.

Για N=100000 η μείωση είναι 53.562%.

Άρα βλέπουμε ότι όσο μεγαλώνει το N τόσο βελτιώνεται και ο χρόνος απόκρισης του προγράμματός μας με την χρήση των SSE εντολών. Μετά όμως από κάποια μεγάλη τιμή (εδώ γύρω στο N=5000 και μετά) η μείωση του χρόνου τείνει στο 53-54% όσο μεγάλο και αν γίνεται το N (ακόμα και για N=1000000).

Οι μικρές διαφορές που παρατηρούνται στα Max για το ω statistic για τα ίδια N είναι λογικές, αφού το SSE αλλάζει την σειρά που εκτελούνται οι πράξεις και εμφανίζονται διαφορετικά rounding errors. Οι τιμές όμως είναι πολύ κοντά μεταξύ τους.

Για να παράξουμε το εκτελέσιμο τρέξαμε την :

$ gcc -msse4.2 -o sse sse.c

και για να το εκτελέσουμε :

$ ./sse N

με N τον αριθμό των DNA θέσεων για τις οποίες θα τρέξουν επαναληπτικά οι πράξεις.

MPI (Message Passing Interface)

Μελετώντας τον SSE κώδικα βρήκαμε σημεία όπου μπορούμε να εκμεταλευτούμε παραλληλισμό με διαφορετικό granularity, και κατά συνέπεια διαφορετικό computation-to-communication ratio. Έτσι καταλήξαμε στο να κάνουμε χρήση του MPI(Message Passing Interface) . Το MPI κάνει "message passing" μέσο μιας ομάδας απο processes(καθε process έχει δικό του ID που λέγεται rank) που έχουν την ιδιότητα να επικοινωνούν μεταξύ τους.

Στο MPI πρόγραμμα κάναμε χρήση των πιο κάτω εντολών:

>Σχηματίζεται μια ομάδα(communicator) απο processes και τα αντιστοιχούμε σε μοναδικά ranks.

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

>Επιστρέφει το μέγεθος της ομάδας(communicator) που δημιουργήθηκε (δηλαδή πόσα process έχω στη διάθεση μου).

MPI_Comm_size(
MPI_Comm communicator,
int* size)

>Επιστρέφει το rank του process που βρίσκεται στον communicator.
MPI_Comm_rank(
MPI_Comm communicator,
int* rank)

>Αποστολή μηνύματος απο σε ένα process (point-to-point communication)
MPI_Send(
void* data,
int count,
MPI_Datatype datatype,
int destination,
int tag,
MPI_Comm communicator)

>Παραλαβή μηνύματος απο ένα process (point-to-point communication)
MPI_Recv(
void* data,
int count,
MPI_Datatype datatype,
int source,
int tag,
MPI_Comm communicator,
MPI_Status* status)

>Αποστολή μηνύματος απο το root process σε όλα τα υπόλοιπα process(collective communication)
MPI_Bcast(
void* data,
int count,
MPI_Datatype datatype,
int root,
MPI_Comm communicator)

>Το root process λαμβάνει συγκεκριμένα στοιχεία απο τα υπόλοιπα process για να εξάγει ένα αποτέλεσμα(sum,min,max...)
MPI_Reduce(
void* send_data,
void* recv_data,
int count,
MPI_Datatype datatype,
MPI_Op op,
int root,
MPI_Comm communicator)

>Τερματίζει το περιβάλλον του MPI.
MPI_Finalize()

Στη συνέχεια βλέπουμε απόσπασματα του κώδικα όπου υλοποιήσαμε το MPI:

```
ierr = MPI_Init(&argc, &argv);
root_process = 0;

MPI_Bcast(LVec, 1, MPI_FLOAT, root_process, MPI_COMM_WORLD);
MPI_Bcast(RVec, 1, MPI_FLOAT, root_process, MPI_COMM_WORLD);
MPI_Bcast(CVec, 1, MPI_FLOAT, root_process, MPI_COMM_WORLD);
MPI_Bcast(mVec, 1, MPI_FLOAT, root_process, MPI_COMM_WORLD);
MPI_Bcast(nVec, 1, MPI_FLOAT, root_process, MPI_COMM_WORLD);
```

Αρχικά ξεκινούμε το MPI περιβάλλον μας με την εντολή **MPI_Init** και στη συνέχεια κάνουμε γνωστά τα vector σε όλα τα process του MPI περιβάλλοντος.

```
/* find out MY process ID, and how many processes were started. */

ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```

Στην συνέχεια με τις εντολές **MPI_Comm_size ,MPI_Comm_rank** βρίσκω το πλήθος των process αλλά και τα rank τους .

```
    if(my_id == root_process) {

        /* I must be the root process. */


        avg_chunk_per_procs = (N / num_procs) ;

        /* distribute a portion of the bector to each child process */

    //printf( "Rank[%i] : | start_chunk = %i | end_chunk=%i  | \n",my_id,0,avg_chunk_per_procs);

        for(an_id = 1; an_id < num_procs; an_id++) {

           start_chunk = an_id*avg_chunk_per_procs + 1;
           end_chunk   = (an_id + 1)*avg_chunk_per_procs;

           if((N - end_chunk) < avg_chunk_per_procs){
                                    end_chunk = N - 1;
             }

    //printf( "Rank[%i] : | start_chunk = %i | end_chunk=%i  | \n",an_id,start_chunk,end_chunk);
           chunk_size_to_send = end_chunk - start_chunk + 1;

           ierr = MPI_Send( &chunk_size_to_send, 1 , MPI_INT,
                 an_id, send_data_tag_chunk_size, MPI_COMM_WORLD);

           ierr = MPI_Send( &start_chunk, 1 , MPI_INT,
                 an_id, send_data_tag_start_chunk, MPI_COMM_WORLD);



        }

        /* and calculate the max of the values in the segment assigned
         * to the root process */

        for(i = 0; i < avg_chunk_per_procs + 1; i+=4) {

                mV= _mm_load_ps(&mVec[i]);
                nV= _mm_load_ps(&nVec[i]);
                RV= _mm_load_ps(&RVec[i]);
                LV= _mm_load_ps(&LVec[i]);
                CV= _mm_load_ps(&CVec[i]);
```

```
                t_num_0= _mm_add_ps(LV,RV);
            //num_1= _mm_mul_ps(mV, _mm_div_ps( _mm_sub_ps(mV,t_1),t_2));
                t_num_1= _mm_div_ps( _mm_mul_ps(mV, _mm_sub_ps(mV,t_1)),t_2);
            //num_2= _mm_mul_ps(nV, _mm_div_ps( _mm_sub_ps(nV,t_1),t_2));
                t_num_2= _mm_div_ps( _mm_mul_ps(nV, _mm_sub_ps(nV,t_1)),t_2);
                t_num= _mm_div_ps(t_num_0,( _mm_add_ps(t_num_1,t_num_2)));
                t_den_0= _mm_sub_ps(CV,( _mm_add_ps(LV,RV)));
                t_den_1= _mm_mul_ps(mV,nV);
                t_den= _mm_div_ps(t_den_0,t_den_1);
                FV= _mm_div_ps(t_num, _mm_add_ps(t_den,t_3));
                mF = _mm_max_ps(FV,mF);
        }

        _mm_store_ps(local_max, mF);
```

Ακολούθως αν βρίσκομαι στο root_process τότε καθορίζω το μέγεθος (avg_chunk_per_procs) που του αναλογεί .Επίσης καθορίζω στα υπόλοιπα process την αρχή(start_chunk) και το τέλος(end_chunk) τους για να υπολογίζω το μέγεθος που τους αναλογεί(chunk_size to send).Ακολούθως για την αποστολή του chunk_size_to_send , start_chunk απο το root_process προς το κάθε process ξεχωριστά, χρησιμοποιώ την εντολή MPI_Send όπου περιλαμανονται τα ανάλογα tag και rank.Τέλος υπολογίζω το κομμάτι που αναλογεί στο root process και αποθηκεύω το αποτέλεσμα στην πίνακα local_max.

```
    else {

        /* I must be a slave process, so I must receive my array segment,
         * storing it in a "local" array, array1. */
          sender1 = status.MPI_SOURCE;
        ierr = MPI_Recv( &chunk_size_to_receive, 1, MPI_INT,
              root_process, send_data_tag_chunk_size, MPI_COMM_WORLD, &status);


        ierr = MPI_Recv( &start_chunk, 1, MPI_FLOAT,
              root_process, send_data_tag_start_chunk, MPI_COMM_WORLD, &status);


        chunk_size_received = chunk_size_to_receive;

    /* for (int w=start_chunk;w<start_chunk +chunk_size_received;w++)
          printf("Lvec[%i] = %f\n",i,LVec[w]);*/
        for(i = start_chunk; i < start_chunk + chunk_size_received; i+=4) {

              LV= _mm_loadu_ps(&LVec[i]);
              mV= _mm_loadu_ps(&mVec[i]);
              nV= _mm_loadu_ps(&nVec[i]);
              RV= _mm_loadu_ps(&RVec[i]);
               CV= _mm_loadu_ps(&CVec[i]);

            /*  how to print __m128
              uint16_t *val = (uint16_t*) &LV;
              printf(" || LV[%i] = %f || , || LV[%i] = %f || , || LV[%i] = %f ||,|| LV[%i] = %f ||\n",
                     i,val[0],i+1,val[1],i+2 ,val[2],i+3, val[3],i+4,val[4]);*/
                num_0= _mm_add_ps(LV,RV);
                num_1= _mm_div_ps( _mm_mul_ps(mV, _mm_sub_ps(mV,t_1)),t_2);
                num_2= _mm_div_ps( _mm_mul_ps(nV, _mm_sub_ps(nV,t_1)),t_2);
                num= _mm_div_ps(num_0,( _mm_add_ps(num_1,num_2)));
                den_0= _mm_sub_ps(CV,( _mm_add_ps(LV,RV)));
                den_1= _mm_mul_ps(mV,nV);
                den= _mm_div_ps(den_0,den_1);
                FV= _mm_div_ps(num, _mm_add_ps(den,t_3));
                mF = _mm_max_ps(FV,mF);
        }

        _mm_store_ps(local_max, mF);

    /* printf("RANK[%i] -> local_max[0]=%f,local_max[1]=%f,local_max[2]=%f,local_max[3]=%f \n",my_id
    }
```

Πιο πάνω βλέπουμε τη περίπτωση όπου δεν είμαστε στο root process αλλά σε κάποιο άλλο.Το πρώτο πράγμα που κάνω εδώ είναι να λάβω τα μηνύματα που έστειλα απο το root_process(chunk_size_to_send,start_chunk) μέσο της εντολής MPI_Recv.Στη συνέχεια υπολογίζω το κομμάτι που αναλογεί σε κάθε ένα process και αποθηκεύω το αποτέλεσμα στην πίνακα local_max.

```
MPI_Reduce(local_max,global_max , 4, MPI_FLOAT, MPI_MAX, 0,MPI_COMM_WORLD);

 if(my_id==0){

            x1 = global_max[0]>global_max[1]?global_max[0]:global_max[1];
            x2 = global_max[2]>global_max[3]?global_max[2]:global_max[3];
            x2 = x1>x2?x1:x2;
   /* printf("Time %f global_max[0]=%f,global_max[1]=%f,global_max[2]=%f,global_max[
             printf("Time %f || Max %f \n", timeTotal/iters,x2);
    }

 ierr = MPI_Finalize();
```

Στο τέλος με την εντολή MPI_Reduce λαμβάνω όλα τα local_max που έχει υπολογίζει κάθε ένα ξεχωριστά και τα αποθηκεύω στον πίνακα global_max. Έτσι το root_process βρίσκει απο το πίνακα global_max το τελικό max και το εκτυπώνει.

Αποτελέσματα

Τρέχοντας τον κώδικα που φτιάξαμε με τις SSE εντολές , μαζί με τον κώδικα όπου στο SSE+MPI που υλοποιήσαμε πιο πάνω , παίρνουμε τα παρακάτω αποτελέσματα για το ω statistic και για τον χρόνο που χρειάστηκαν να εκτελεστούν οι πράξεις για να βρεθεί.

(P=2 - N=100,1000,10000,100000)

```
******************** SSE (N=100) **********************
Time 0.000002 Max 15.967343
******************** SSE+MPI (N=100)(P=2) **********************
Time 0.000003 || Max 15.967343
```

```
******************** SSE (N=1000) **********************
Time 0.000023 Max 121.787010
******************** SSE+MPI (N=1000)(P=2) **********************
Time 0.000011 || Max 121.787010
```

```
******************** SSE (N=10000) **********************
Time 0.000183 Max 510.145325
******************** SSE+MPI (N=10000)(P=2) **********************
Time 0.000098 || Max 510.145325
```

```
******************** SSE (N=100000) **********************
Time 0.001775 Max 8077.673340
******************** SSE+MPI (N=100000)(P=2) **********************
Time 0.001044 || Max 8077.673340
```

(P=4 - N=100,1000,10000,100000)

```
******************** SSE (N=100) **********************
Time 0.000002 Max 15.967343
******************** SSE+MPI (N=100)(P=4) **********************
Time 0.000010 || Max 15.967343
```

```
******************** SSE (N=1000) **********************
Time 0.000019 Max 121.787010
******************** SSE+MPI (N=1000)(P=4) **********************
Time 0.000016 || Max 121.787010
```

```
******************** SSE (N=10000) **********************
Time 0.000172 Max 510.145325
******************** SSE+MPI (N=10000)(P=4) **********************
Time 0.000103 || Max 510.145325
```

```
******************** SSE (N=100000) **********************
Time 0.001880 Max 8077.673340
------------------------------------------------------------
******************** SSE+MPI (N=100000)(P=4) **********************
Time 0.000986 || Max 8077.673340
```

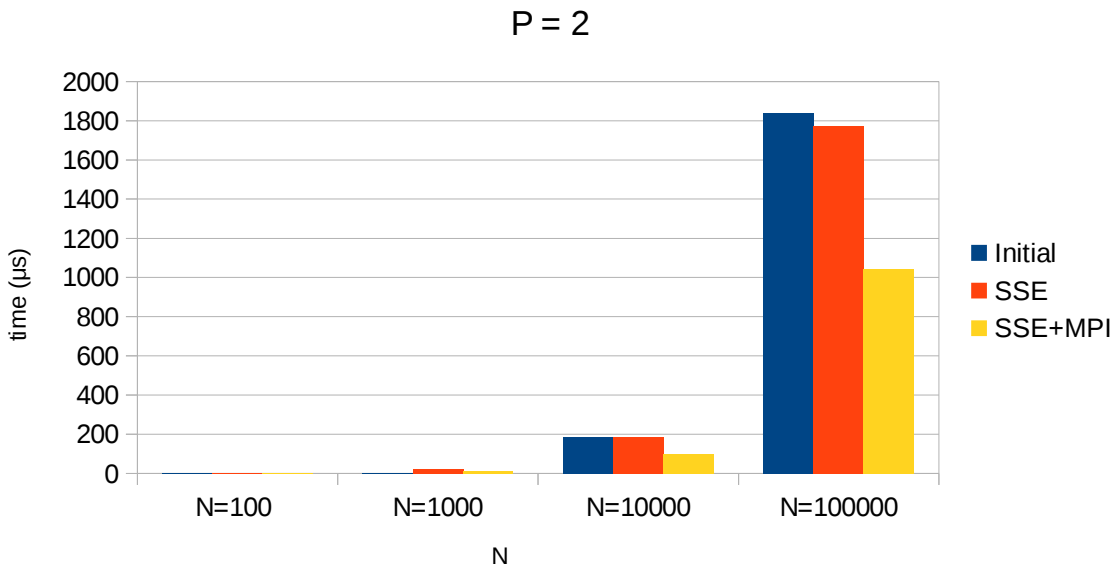
(P=8 - N=100,1000,10000,100000)

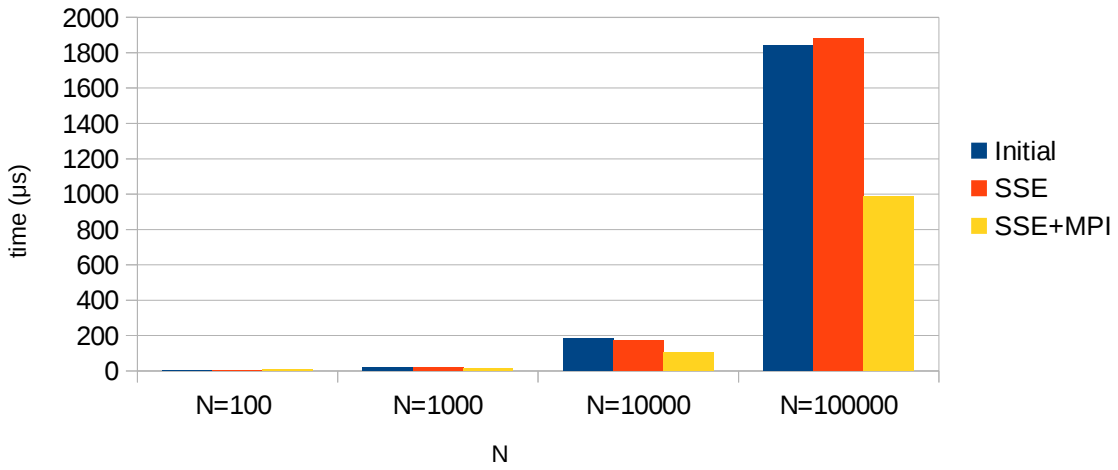
```
******************** SSE (N=100) **********************
Time 0.000002 Max 15.967343
******************** SSE+MPI (N=100)(P=8) **********************
Time 0.001329 || Max 15.967343
```

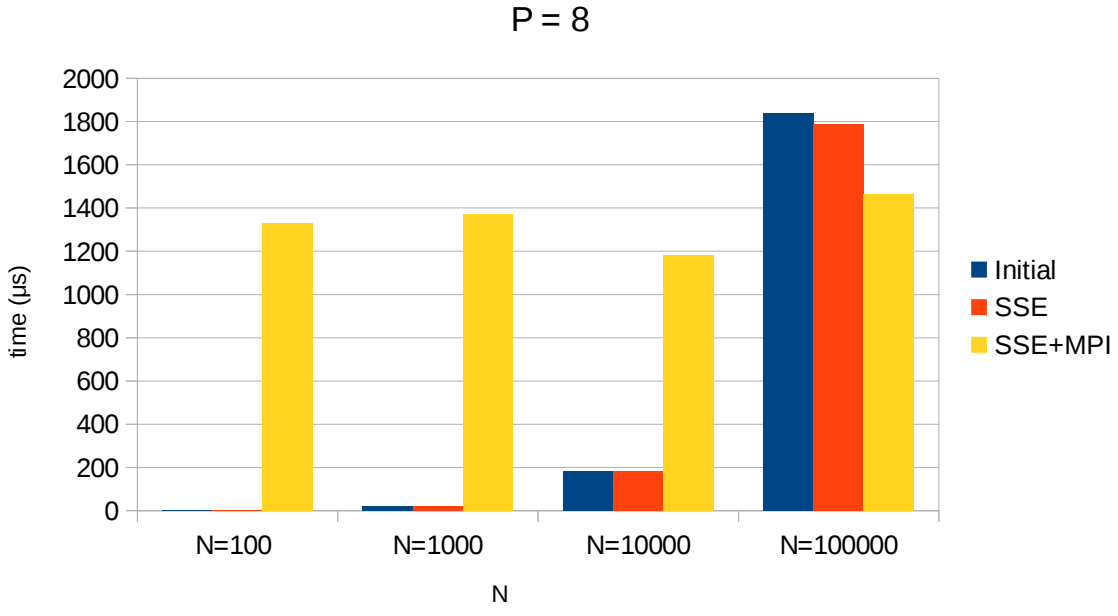
```
******************** SSE (N=1000) **********************
Time 0.000022 Max 121.787010
******************** SSE+MPI (N=1000)(P=8) **********************
Time 0.001371 || Max 121.787010
```

```
******************** SSE (N=10000) **********************
Time 0.000183 Max 510.145325
******************** SSE+MPI (N=10000)(P=8) **********************
Time 0.001181 || Max 510.145325
```

```
******************** SSE (N=100000) **********************
Time 0.001789 Max 8077.673340
******************** SSE+MPI (N=100000)(P=8) **********************
Time 0.001464 || Max 8077.673340
```


P = 2
time (µs)
2000
1800
1600
1400
1200
1000
800
600
400
200
0
N=100
N=1000
N=10000
N=100000
N
Initial
SSE
SSE+MPI


P = 4
time (µs)
2000
1800
1600
1400
1200
1000
800
600
400
200
0
N=100
N=1000
N=10000
N=100000
N
Initial
SSE
SSE+MPI


P = 8
time (µs)
2000
1800
1600
1400
1200
1000
800
600
400
200
0
N=100
N=1000
N=10000
N=100000
N
Initial
SSE
SSE+MPI

Παρατηρούμε ότι για N=100000 και για P=4 έχουμε ταχύτερη εκτέλεση των εντολών σε σχέση με τον αρχικό κώδικα κατά 0.000853 seconds, δηλαδή μείωση χρόνου κατά 46% .Άρα βλέπουμε ότι και εδώ όσο μεγαλώνει το N τόσο βελτιώνεται και ο χρόνος απόκρισης του προγράμματός μας με την χρήση των SSE+MPI εντολών.

Για μεγάλο αριθμό απο P παρατηρούμε ότι , το SSE+MPI για να γίνει πιο αποδοτικό πρέπει να έχουμε πάρα πολύ μεγάλο N (π.χ. 100000).Αν έχουμε μικρό αριθμό απο N γίνεται μεγάλη αλλαγή πληροφορίας ανάμεσα στα process ενώ μπορούσαν να εκτελεστούν πιο γρήγορα χωρίς το MPI.

Για να παράξουμε το εκτελέσιμο τρέξαμε την :

$ mpicc mpi.c -o mpi_ex

και για να το εκτελέσουμε :

$ $ mpiexec -n P ./mpi_ex N

όπου N τον αριθμό των DNA θέσεων για τις οποίες θα τρέξουν επαναληπτικά οι πράξεις και P ο αριμός των process που θέλουμε.