Αρχιτεκτονική Παράλληλων και Κατανεμημένων Υπολογιστών

Άρης Ζερβάκης - Στέφανος Καλογεράκης

Πολυτεχνείο Κρήτης — 2 Ιουνίου 2019

Εισαγωγή

Στη δεύτερη εργαστηριακή άσκηση καλεστήκαμε να χρησιμοποιήσουμε συνδιαστικά Streaming SIMD Extensions (SSE), MPI και Pthreads με σκοπό την παραλληλοποίηση του υπολογισμού του ω statistic, το οποίο εφαρμόζεται για ανίχνευση θετικής επιλογής σε ακολουθίες DNA. Ως κώδικας αναφοράς, χρησιμοποιήθηκε που δόθηκε απο τον διδάσκοντα του μαθήματος (φάκελος με όνομα Serial).

Για την υλοποίηση της άσκησης μας προχωρήσαμε σε 4 διαφορετικές υλοποιήσεις:
1. Παραλληλοποίηση με SSE Εντολές
2. Παραλληλοποίηση με SSE Εντολές και Pthreads
3. Παραλληλοποίηση με SSE Εντολές και Pthreads και MPI
4. (Bonus) Παραλληλοποίηση με SSE Εντολές για διαφορετικά Memory Layout

1 Υλοποίηση

Για τον σειριακό υπολογισμό του ω statistic, χρησιμοποιήσαμε αυτούσιο τον reference code χωρίς να πραγματοποίησουμε αλλαγές και για αυτό δεν γίνεται κάποια παραπάνω αναφορά. Παρακάτω γίνεται ανάλυση όλων των μεθόδων παραλληλοποίησης που μελετήθηκαν στα πλαίσια αυτού του πρότζεκτ.

1.1 SSE Εντολές

Η μέθοδος παραλληλοποίησης με τη χρήση SSE εντολών υλοποιήθηκε με τη χρήση pointers όπως είδαμε και στις διαλέξεις. Αξίζει να σχολιάσουμε ότι επιλέξαμε την συγκεκριμένη μέθοδο σε σύγκριση με την υλοποίηση με χρήση load καθώς όπως διδαχτήκαμε είναι πιο γρήγορη. Μετά από δοκιμή στα δικά μας δεδομένα-υπολογισμούς επιβεβαιώσαμε το συγκεκριμένο γεγονός αφού η χρήση των pointers οδήγησε σε λίγο πιο γρήγορο χρόνο εκτέλεσης που είναι και βασικό ζητούμενο της παραλληλοποίησης.

-Όλες οι μεταβλητές οι οποίες ξεκινούν με underscore συσχετίζονται με το λειτουργικό κομμάτι της SSE υλοποίησης.
-Δημιουργήθηκαν οι παρακάτω μεταβλητές πλάτους 128 bits.

```
__m128 *mVec_ptr = (__m128 *) mVec;
__m128 *nVec_ptr = (__m128 *) nVec;
__m128 *LVec_ptr = (__m128 *) LVec;
__m128 *RVec_ptr = (__m128 *) RVec;
__m128 *CVec_ptr = (__m128 *) CVec;
__m128 *FVec_ptr = (__m128 *) FVec;

__m128 avgF_vec = _mm_setzero_ps();
__m128 maxF_vec = _mm_setzero_ps();
__m128 minF_vec = _mm_set_ps1(FLT_MAX);
```

Για την υλοποίηση των υπολογισμών έγιναν οι παρακάτω αλλαγές:
-Αλλαγή των malloc, free με τις __mm_malloc, __mm_free. (εντολές που χρησιμοποιούνται για ευθυγράμμιση των δεδομένων).
-Τροποποίηση εντολών. (Σε σχόλια παρατίθενται οι εντολές στην αρχική μορφή τους και έπειτα η τροποποίηση τους)

```
for(unsigned int i=0; i<N/4 ;i++){

            //float num_0 = LVec[i] + RVec[i];
            temp_num_0 = _mm_add_ps(LVec_ptr[i],RVec_ptr[i]);

            //float num_1 = mVec[i]*(mVec[i]-1.0f)/2.0f;
            temp_num_1 = _mm_sub_ps(mVec_ptr[i], temp_one);
            temp_num_1 = _mm_mul_ps(mVec_ptr[i], temp_num_1);
            temp_num_1 = _mm_div_ps(temp_num_1, temp_two);

            //float num_2 = nVec[i]*(nVec[i]-1.0f)/2.0f;
            temp_num_2 = _mm_sub_ps(nVec_ptr[i], temp_one);
            temp_num_2 = _mm_mul_ps(nVec_ptr[i], temp_num_2);
            temp_num_2 = _mm_div_ps(temp_num_2, temp_two);

            //float num = num_0/(num_1+num_2);
            temp_num = _mm_add_ps(temp_num_1,temp_num_2);
            temp_num = _mm_div_ps(temp_num_0, temp_num);

            //float den_0 = CVec[i]-LVec[i]-RVec[i];
            temp_den_0 = _mm_sub_ps(CVec_ptr[i], LVec_ptr[i]);
            temp_den_0 = _mm_sub_ps(temp_den_0, RVec_ptr[i]);

            //float den_1 = mVec[i]*nVec[i];
            temp_den_1 = _mm_mul_ps(mVec_ptr[i], nVec_ptr[i]);

            //float den = den_0/den_1;
            temp_den = _mm_div_ps(temp_den_0, temp_den_1);

            //FVec[i] = num/(den+0.01f);
            FVec_ptr[i] = _mm_add_ps(temp_den, __temp_one);
            FVec_ptr[i] = _mm_div_ps(temp_num, FVec_ptr[i]);

            //maxF = FVec[i]>maxF?FVec[i]:maxF;
            maxF_vec = _mm_max_ps(FVec_ptr[i], maxF_vec);

            //minF = FVec[i]<minF?FVec[i]:minF;
            minF_vec = _mm_min_ps(FVec_ptr[i], minF_vec);

            //avgF += FVec[i];
            avgF_vec = _mm_add_ps(FVec_ptr[i], avgF_vec );
}
```

Στην υλοποίηση πραγματοποιούμε loop unrolling και jamming στις εντολές του for-loop, με κάθε i να αναλογεί σε 4 στοιχεία. Όσον αφορα την εύρεση του max, min, avg έχοντας ορίσει τις κατάλληλες m128 μεταβλητές πραγματοποιύμε επιμέρους σε συγκρίσεις ανά τετράδες με SSE εντολές και στο τέλος αποθηκεύουμε σε μια global μεταβλητή την σωστή τιμή ανάλογα με την περίπτωση.

```
maxF = maxF_vec[0];
maxF = maxCalc(maxF_vec[1],maxF);
maxF = maxCalc(maxF_vec[2],maxF);
maxF = maxCalc(maxF_vec[3],maxF);

minF = minF_vec[0];
minF = minCalc(minF_vec[1],minF);
minF = minCalc(minF_vec[2],minF);
minF = minCalc(minF_vec[3],minF);

avgF = avgF_vec[0] + avgF_vec[1] + avgF_vec[2] + avgF_vec[3];
```

Στο τέλος, προσθέσαμε ένα κομμάτι κώδικα το οποίο για τα συγκεκριμένα δεδομένα που δοκιμάζουμε δεν πρόκειται να χρησιμοποιηθεί αλλά εισάγεται για λόγους πληρότητας, σε περίπτωση που υπάρξει ανάγκη για δοκιμή σε άλλα δεδομένα.

```
for (int j = (N - N % 4); j < N; j++) {
            float num_0 = LVec[j] + RVec[j];
            float num_1 = mVec[j] * (mVec[j] - 1.0f) / 2.0f;
            float num_2 = nVec[j] * (nVec[j] - 1.0f) / 2.0f;
            float num = num_0 / (num_1 + num_2);
            float den_0 = CVec[j] - LVec[j] - RVec[j];
            float den_1 = mVec[j] * nVec[j];
            float den = den_0 / den_1;

            FVec[j] = num / (den + 0.01f);
            maxF = FVec[j] > maxF ? FVec[j] : maxF;
            minF = FVec[j]<minF?FVec[j]:minF;
            avgF += FVec[j];
}
```

Το παραπάνω κομμάτι κώδικα θα χρησιμοποιηθεί όταν υπάρχουν σαν υπόλοιπο λιγότερα απο 4 στοιχεία όπου δεν μπορεί να εφαρμοστεί το SSE και γίνεται ένας απλός σειριακός υπολογισμός. Σε όλες τις παράλληλες υλοποιήσεις έχει εισαχθεί αντίστοιχο κομμάτι κώδικα που απευθύνεται σε όλα τα στοιχεία που υλοποιούνται σειριακά και δεν μπορούν να παραλληλοποιηθούν με βάση τους διαχωρισμούς που πραγματοποιούμε.

1.2 SSE Εντολές και Pthreads

Στο δέυτερο μέρος έπρεπε να παραλληλοποιήσουμε το reference code συνδιαστικά, με SSE Εντολές και Pthreads. Τα Pthreads βασίζονται στο μοντέλο master-worker , δημιουργούνται στην αρχή της main και γίνονται join πριν την επιστροφή της. Στην υλοποίηση μας όσο το master thread αρχικοποιεί τις μεταβλητές μας, τα worker threads είναι σε κατάσταση busy wait και έπειτα το master μοιράζει στα worker threads τους υπολογισμούς. Ο συγχρονισμός των Pthreads σε κάθε iteration του for loop επιτυγχάνεται με τη χρήση barrier.

Στον υπάρχον κώδικα, πραγματοποιήθηκαν αρκετές αλλαγές προκειμένου να προστεθούν και τα worker threads και να πετύχουμε την λειτουργία που επιθυμούμε.
Συγκεκριμένα, προσθέσαμε τις συναρτήσεις:

- **initializeThreadData**: Ψπεύθυνη για αρχικοποίηση των δεδομένων του κάθε thread.
- **computeOmega**: Υπεύθυνη για την ενεργοποίηση των threads απο busy-wait σε ενεργή κατάσταση.
- **terminateWorkerThreads**: Υπέυθυνη για τον τερματισμό των worker threads και ένωση με το master thread.
- **threads**: Υπεύθυνη για την διατήρηση των worker threads σε busy wait κατάσταση κατά σειρά που

φαίνονται στο αρχείο.

Ενδεικτικά, παραθέτουμε την συνάρτηση threads που είναι ιδιαίτερα σημαντική για την υλοποίηση των worker threads.

```
void * thread (void * x)
{
    threadData_t * currentThread = (threadData_t *) x;

    int tid = currentThread->threadID;

    int threads = currentThread->threadTOTAL;
    while(1)
    {
        __sync_synchronize();

        /*
         * When we receive exit signal we return to terminate everything
         */
        if(currentThread->threadOPERATION==EXIT)
            return NULL;

        /*
         * All threads except master thread have access here,
         * and when all the calculations are done go back
         * on BUSYWAIT CONDITION
         */
        if(currentThread->threadOPERATION==COMPUTE_OMEGA)
        {
            computeOmega (currentThread);

            currentThread->threadOPERATION=BUSYWAIT;

            pthread_barrier_wait(&barrier);

        }
    }

    return NULL;
}
```

Επιπρόσθετα, στο κομμάτι των υπολογισμών οφείλαμε να προσαρμόσουμε το κάθε thread έτσι ώστε να πραγματοποιεί υπολογισμούς ανεξέρτητα χωρίς να υπάρχουν conflicts και περιττοί υπολογισμοί. Όπως και στο πρώτο ερώτημα, όλα τα δεδομένα, ήταν οργανωμένα σε τετράδες για να λειτουργήσει το μοντέλο του SSE.

```
int tasksPerThread = (N/4)/totalThreads;

int start = tasksPerThread * threadID;
int stop = tasksPerThread * threadID+tasksPerThread-1;
```

Επισήμανση:
Οι εντολές start και stop οριοθετούν το range των υπολογισμών για το εκάστοτε thread.

Τέλος, διαφοροποιήσαμε και τα ορίσματα που δέχεται το εκτελέσιμο αρχείο αφού δίνουμε σαν δεύτερο όρισμα τον αριθμό των threads που επιθυμούμε.

1.3 SSE Εντολές, Pthreads και MPI

Σε συνέχεια του προηγούμενου ερωτήματος, χρειάστηκε να προσθέσουμε κώδικα ώστε να επιτυγχαεται η δημιουργία processes και να ανταλλάσονται πληροφορίες μεταξύ τους με την χρήση του πρωτόκολλου MPI(Message Passing Interface) . Σκοπός μας και πάλι είναι να επιτυγχάνεται ακόμα μεγαλύτερη ταχύτητα στους υπολογισμούς.

Στην αρχή του κώδικα, για την επιτυχή εκτέλεση και αρχικοποίηση του MPI, χρησιμοποιήθηκαν οι παρακάτω εντολές:

```
//Init MPI process
MPI_Init(NULL, NULL);

// Get the number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

// Get the rank of the process
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

Με αυτόν τον τρόπο δημιουργούμε processes και ενημερώνονται οι βασικές μεταβλητές world_size(αριθμός processes) και world_rank(id του εκάστοτε process) που χρειαζόμαστε στην συνέχεια. Να σημειώσουμε ότι το world_size το περνάμε κατά την εκτέλεση του κώδικα με την εντολή mpiexec.

Η δομή του κώδικα, είναι σε πολύ μεγάλο βαθμό ίδια με την προηγούμενη υλοποίηση με λίγες επιμέρους αλλαγές προκειμένου να ενσωματωθεί το MPI. Αυτό οφείλεται και στο γεγονός ότι όλα τα processes μπορούν να αποθηκεύουν όλα τα δεδομένα ένω το κάθε προςεσς χρησιμοποιεί τον ίδιο αριθμό απο threads .Μια σημαντική αλλαγή αφορά και πάλι στον διαχωρισμό των δεδομένων καθώς οφείλαμε να προχωρήσουμε στην ομοιόμορφη κατανομή τους.

```
int MPI_proc = (N / world_size)/4;     //Compute the number of
                                       //processes in a world
int MPI_start = world_rank * MPI_proc;  //Find the starting point
                                         //of each node
int tasksPerThread = MPI_proc/totalThreads;

int start = MPI_start + tasksPerThread * threadID;
int stop = (MPI_start + tasksPerThread * threadID)+tasksPerThread-1;
```

Οι μεταβλητές MPI_proc, MPI_start είναι υπεύθυνες για τον υπολογισμό των στοιχείων προς υπολογισμό βάσει των processes και το σημείο εκκίνησης του κάθε προςεσς. Οι μεταβλητές start και stop λειτουργούν με την λογική του προηγούμενου ερωτήματος με τον διαχωρισμό για pthreads, ενώ προστίθεται και ενα offset(MPI_start) για την κατανομή σε όλα τα processes.

Μετά το τέλος των υπολογισμών, το κάθε process έχει υπολογίσει επιμέρους τις τιμές που μας ενδιαφέρουν(min, max, avg) ενώ σαν τελευταίο βήμα μένει η σύγκριση με τα άλλα processes και ο υπολογισμός global min, max και avg. Δεσμεύουμε λοιπόν, τον απαραίτητο χώρο στην μνήμη και με χρήση της εντολής MPI_Gather συλλέγουμε συγκεντρωτικά δεδομένα απο όλα τα προςεσσες και προχωράμε σε αυτήν την σύγκριση. Να επισημάνουμε ότι οι μέθοδοι υπολογισμού είναι ίδιοι με το προηγούμενο ερώτημα.

```
if (world_rank == 0) {
        //Allocate space for everything we will need
        max_results = (float *) malloc(world_size * sizeof(float));
        assert(max_results != NULL);

        min_results = (float *) malloc(world_size * sizeof(float));
        assert(min_results != NULL);

        avg_results = (float *) malloc(world_size * sizeof(float));
        assert(avg_results != NULL);
}

//Gather all max results
MPI_Gather(&GL_maxF, 1, MPI_FLOAT, max_results, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);

//Gather all min results
MPI_Gather(&GL_minF, 1, MPI_FLOAT, min_results, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);

//Gather all avg results
MPI_Gather(&GL_avgF, 1, MPI_FLOAT, avg_results, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

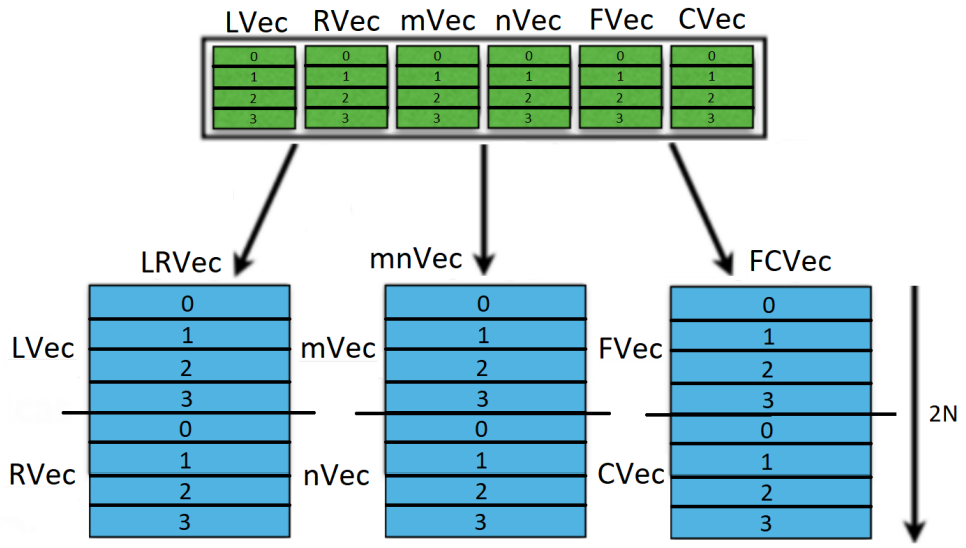
1.4 BONUS: Υλοποίηση διαφορετικών memory layout για τη βελτιστοποίηση της παραλληλοποίησης με SSE Εντολές

Σαν τελευταίο κομμάτι στα πλαίσια του προτζεκτ, ζητήθηκε να μελετήσουμε την χρήση εντολών SSE για διαφορετικό memory layout. Για την καλύτερη μελέτη των διαφορετικών layout, προχωρήσαμε σε τρεις διαφορετικές υλοποιήσεις, οι οποίες βρίσκονται στα αρχεία:
- SSE-MEM-LAYOUT
- SSE-MEM-LAYOUT-3
- SSE_MEMORY_LAYOUT_6

Τα διαφορετικά memory layout αποσκοπούν στην αποφυγή των cache misses που αναπόφευκτα συμβαίνουν όταν έχουμε δεδομένα σε πολλούς διαφορετικούς πίνακες αποθηκευμένους σε διαφορετικές θέσεις μνήμης. Στον αρχικό μας κώδικα, έχουμε 6 διαφορετικά malloc, ένα για τον κάθε πίνακα που χρειαζόμαστε. Σε κάθε μια απο τις υλοποιήσεις, διαφοροποίησαμε τον αριθμό απο πίνακες, με σκοπό να δούμε τι βελτίωση θα επιφέρει η κάθε περίπτωση.

Στην περίπτωση του αρχείου SSE-MEM-LAYOUT χρησιμοποιήσαμε 3 malloc, για να δεσμεύσουμε χώρο στην μνήμη για τους πίνακες LRVec, mnVec, FCVec. Η ονοματολογία των πινάκων είναι ανάλογα με τους πίνακες που έχουν γίνει merge σε κάθε περίπτωση. Αντίστοιχα, στο αρχείο SSE-MEM-LAYOUT-3 χρειαστήκαμε 2 malloc, για τους πίνακες LRCVec, και mnFVec, ενώ στο αρχείο SSE_MEMORY_LAYOUT_6 χρειαστήκαμε 1 malloc, ενώνοντας όλους τους πίνακες σε έναν. Στα πρώτα δύο αρχεία το κριτήριο για την ένωση των πινάκων ήταν η συσχέτιση τους κατά την πραγματοποίηση υπολογισμών.

Μετά την δέσμευση μνήμης σε κάθε περίπτωση, γεμίζουμε τους πίνακες με δεδομένα ανά τετράδες όπως φαίνεται παρακάτω στην εικόνα για μπορέσουμε να αξιοποιήσουμε τις εντολές τύπου SSE.



Σχήμα 1: Σχέδιο Υλοποίησης

Στην συνέχεια, ανάλογα με τον πίνακα και το στοιχείο που επιθυμούμε πρόσβαση αλλάζουμε θέση στον στοιχείο του πίνακα. Παρακάτω δείχνουμε ένα ενδεικτικό κομμάτι κώδικα με τους υπολογισμούς απο την υλοποίηση με 3 malloc(SSE-MEM-LAYOUT).

```
for(unsigned int i=0; i< 2*N/4 ;i+=2)
{
            //Replace each statement step by step
            //using the instructions as mentioned before

            //float num_0 = LVec[i] + RVec[i];
            temp_num_0 = _mm_add_ps(LRVec_ptr[i],LRVec_ptr[i+1]);

            //float num_1 = mVec[i]*(mVec[i]-1.0f)/2.0f;
            temp_num_1 = _mm_sub_ps(mnVec_ptr[i], temp_one);
            temp_num_1 = _mm_mul_ps(mnVec_ptr[i], temp_num_1);
            temp_num_1 = _mm_div_ps(temp_num_1, temp_two);


            //float num_2 = nVec[i]*(nVec[i]-1.0f)/2.0f;
            temp_num_2 = _mm_sub_ps(mnVec_ptr[i+1], temp_one);
            temp_num_2 = _mm_mul_ps(mnVec_ptr[i+1], temp_num_2);
            temp_num_2 = _mm_div_ps(temp_num_2, temp_two);


            //float num = num_0/(num_1+num_2);
            temp_num = _mm_add_ps(temp_num_1,temp_num_2);
            temp_num = _mm_div_ps(temp_num_0, temp_num);

            //float den_0 = CVec[i]-LVec[i]-RVec[i];
            temp_den_0 = _mm_sub_ps(FCVec_ptr[i+1], LRVec_ptr[i]);
            temp_den_0 = _mm_sub_ps(temp_den_0, LRVec_ptr[i+1]);


            //float den_1 = mVec[i]*nVec[i];
            temp_den_1 = _mm_mul_ps(mnVec_ptr[i], mnVec_ptr[i+1]);

            //float den = den_0/den_1;
            temp_den = _mm_div_ps(temp_den_0, temp_den_1);

            //FVec[i] = num/(den+0.01f);
            temp_fvec = _mm_add_ps(temp_den, __temp_one);
            FCVec_ptr[i] = _mm_div_ps(temp_num, temp_fvec);

            //maxF = FVec[i]>maxF?FVec[i]:maxF;
            maxF_vec = _mm_max_ps(FCVec_ptr[i], maxF_vec);

            //minF = FVec[i]<minF?FVec[i]:minF;
            minF_vec = _mm_min_ps(FCVec_ptr[i], minF_vec);

            //avgF += FVec[i];
            avgF_vec = _mm_add_ps(FCVec_ptr[i], avgF_vec );


}
```

2 Εκτέλεση

Για την εκτέλεση, απλά καλούμαστε να τρέξουμε στο command line την παρακάτω εντολή, παίρνοντας τα ακόλουθα αποτελέσματα:

Command Line

```
$ ./run.sh

---------------------- Building everything ------------------------------


---------------------- Giving permissions -------------------------------


---------------------- Reference Code N:10000000-------------------------
Omega time 0.177147s - Total time 2.829882s - Min -2.711918e+06 -
Max 6.048419e+05 - Avg -6.529043e-01



---------------------- SSE N:10000000-----------------------------------
Omega time 0.106534s - Total time 2.149127s - Min -2.711918e+06 -
Max 6.048419e+05 - Avg -6.348448e-01



-------------- SSE-PTHREADS N:10000000, PTHREADS:2----------------------
Omega time 0.053595s - Total time 2.226116s - Min -2.711918e+06 -
Max 6.048419e+05 - Avg -6.332115e-01


--------------- SSE-PTHREADS N:10000000, PTHREADS:4---------------------
Omega time 0.044613s - Total time 2.552894s - Min -2.711918e+06 -
Max 6.048419e+05 - Avg -6.323332e-01
```

Command Line

```
-------- SSE-PTHREADS-MPI N:10000000, PTHREADS:2, PROCESSES:2------------
Omega time 0.041026s - Total time 2.108915s - Min -2.711918e+06 -
Max 6.048419e+05 - Avg -6.323332e-01


-------- SSE-PTHREADS-MPI N:10000000, PTHREADS:4, PROCESSES:2------------
Omega time 0.040200s - Total time 4.056555s - Min -2.711918e+06 -
Max 6.048419e+05 - Avg -6.323262e-01


-------- SSE-PTHREADS-MPI N:10000000, PTHREADS:2, PROCESSES:4------------
Omega time 0.028752s - Total time 4.942250s - Min -2.711918e+06 -
Max 6.048419e+05 - Avg -6.323262e-01


-------- SSE-PTHREADS-MPI N:10000000, PTHREADS:4, PROCESSES:4------------
Omega time 0.033914s - Total time 8.615657s - Min -2.711918e+06 -
Max 6.048419e+05 - Avg -6.321101e-01


----------------------- Bonus -----------------------------------------


---------------------- SSE N:10000000(AGAIN)-----------------------------
Omega time 0.108413s - Total time 2.189199s - Min -2.711918e+06 -
Max 6.048419e+05 - Avg -6.348448e-01


------------ SSE N:10000000, SSE_MEM_LAYOUT 3 vectors--------------------
Omega time 0.105147s - Total time 2.193838s - Min -2.711918e+06 -
Max 6.048419e+05 - Avg -6.348448e-01


------------ SSE N:10000000, SSE_MEM_LAYOUT 2 vectors--------------------
Omega time 0.104894s - Total time 2.180221s - Min -2.711918e+06 -
Max 6.048419e+05 - Avg -6.348448e-01


------------ SSE N:10000000, SSE_MEM_LAYOUT 1 vector --------------------
Omega time 0.104966s - Total time 2.203330s - Min -2.711918e+06 -
Max 6.048419e+05 - Avg -6.348448e-01
```

Σημείωση: Για να τρέξουμε το run script είναι απαραίτητο να βρισκόμαστε στο directory όπου έχουμε τοποθετήσει τα αρχεία μας και να έχουμε ήδη εγκατεστημένα το gcc, make και mpich ώστε να μπορεί να γίνει compile και να πάρουμε τα αποτελέσματα.

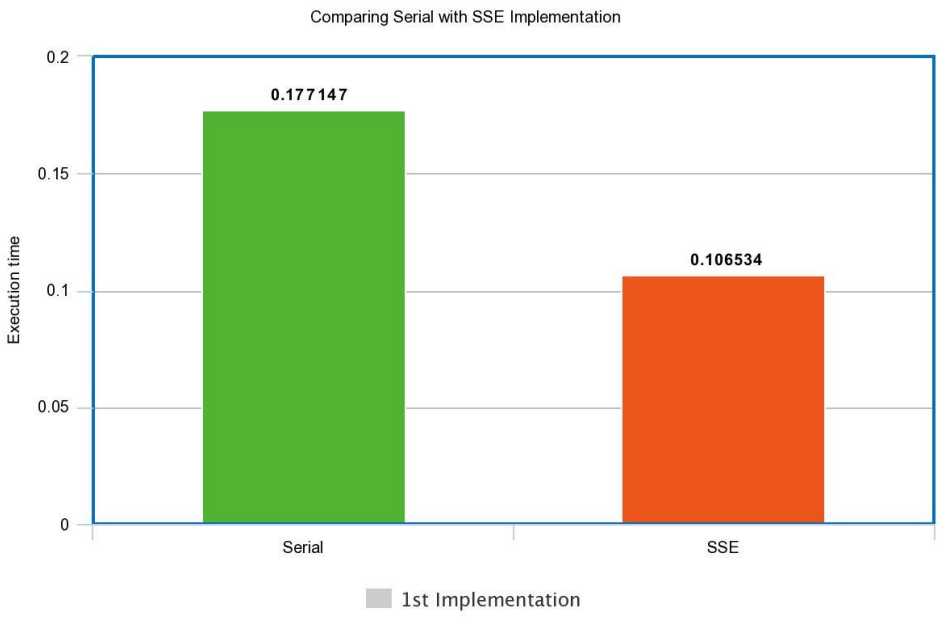
3 Συμπεράσματα

Απεικονίζοντας τους παραπάνω χρόνους μπορούμε να εξάγουμε πολύτιμα συμπεράσματα για τις υλοποιήσεις μας, υπολογίζοντας το speedup που έχουμε στην εκάστοτε περίπτωση.
Μπορούμε να υπολογίσουμε το speedup με τον ακόλουθο τύπο:

$$Speedup = \frac{SerialCodeExecutionTime}{ParallelizedCodeExecutionTime} \tag{1}$$

3.1 SSE Εντολές

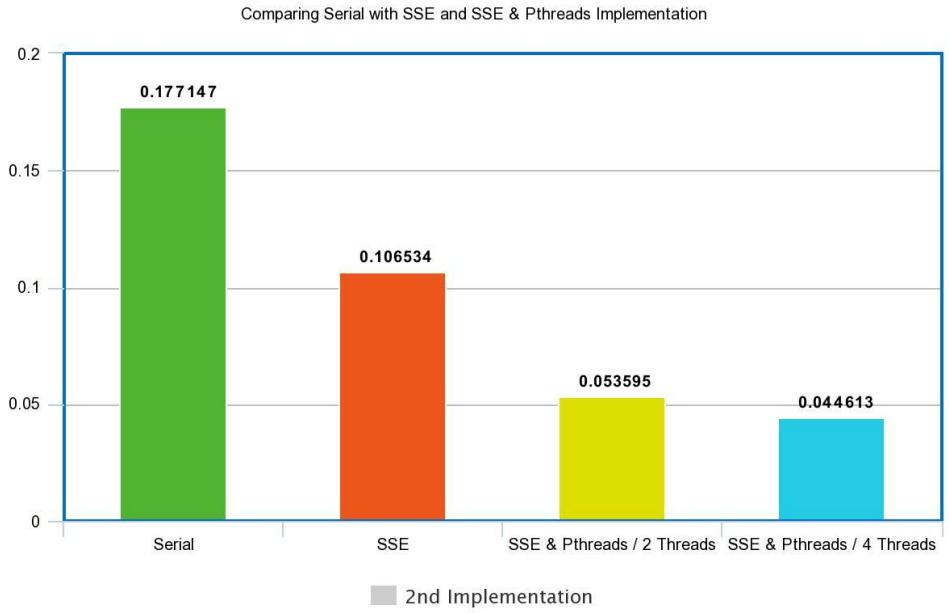
Στην πρώτη περίπτωση, στην παραλληλοποίηση του reference code με SSE εντολές παρατηρούμε αμέσως καλύτερη απόδοση, αρκετά χαμηλότερο όμως απο το θεωρητικό speedup το οποίο είναι ίσο με 4. Πιο συγκεκριμένα έχουμε $Speedup = 1.66$.



Σχήμα 2: Comparing Serial with SSE Implementation

3.2 SSE Εντολές, Pthreads

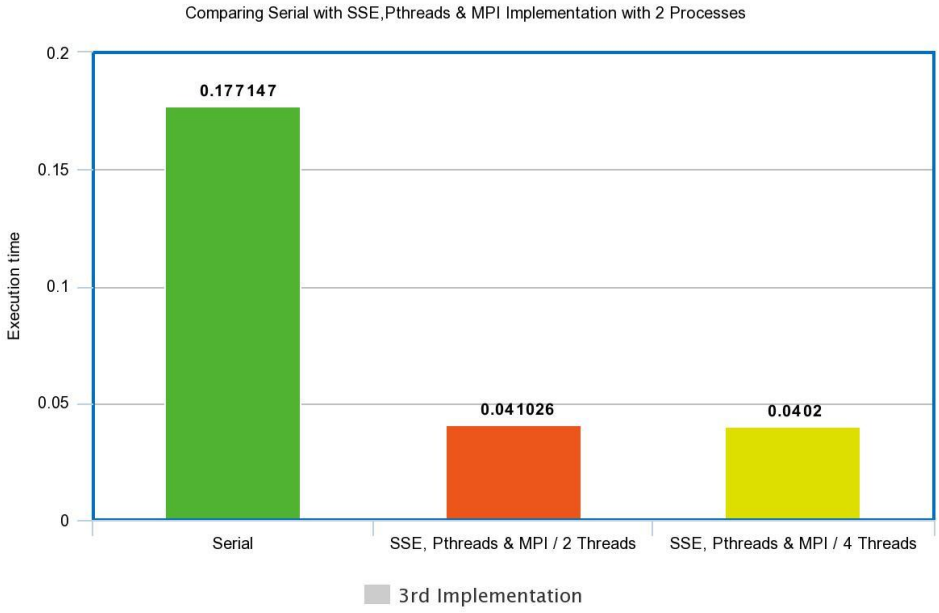
Στην δεύτερη περίπτωση, στην παραλληλοποίηση του reference code με SSE εντολές σε συνδιασμό με Pthreads παρατηρούμε ακόμα καλύτερη απόδοση σε σχέση τόσο με το σειριακό, όσο και με την παραλληλοποίηση μόνο με SSE εντολές, με την περίπτωση των τεσσάρων threads να είναι καλύτερη απο την περίπτωση των δύο.
Πιο συγκεκριμένα, στην περίπτωση των 2 threads έχουμε $Speedup = 3.3$, και στην περίπτωση των 4 threads $Speedup = 3.97$



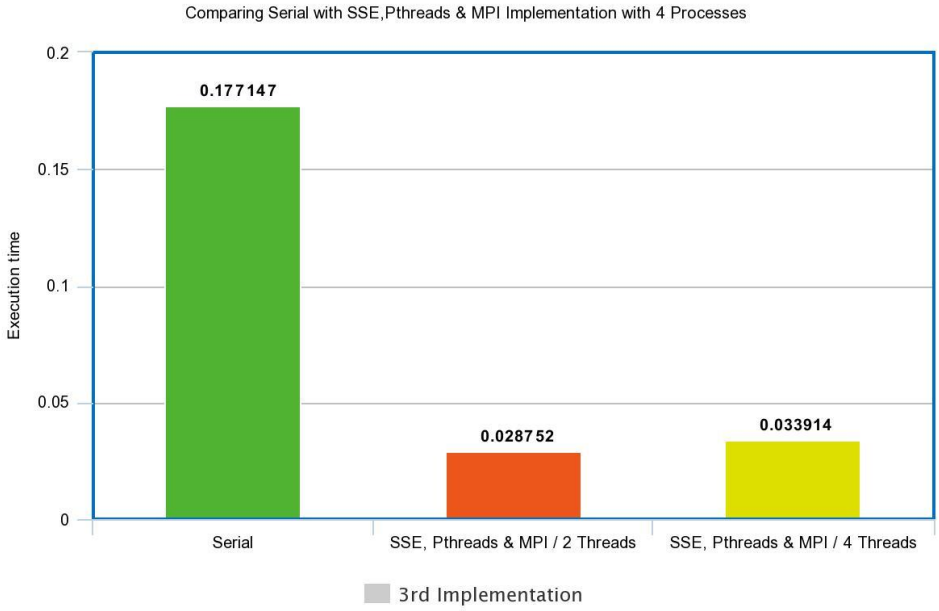
Σχήμα 3: Comparing Serial with SSE and Pthreads Implementation

3.3 SSE Εντολές, Pthreads και MPI

Στην περίπτωση παραλληλοποίησης του κώδικα με MPI δε μας ζητήθηκε να αξιολογήσουμε την απόδοση και το speedup. Άλλωστε δεν θα είχε ιδιαίτερο νόημα να αξιολογηθεί διότι τρέχουμε τον κώδικα τοπικά και όχι σε κάποιο cluster. Ακολουθούν ενδεικτικά τα διαγράμματα με τους χρόνους που λάβαμε:



Σχήμα 4: Comparing Serial with SSE,Pthreads and MPI Implementation with 2 Processes

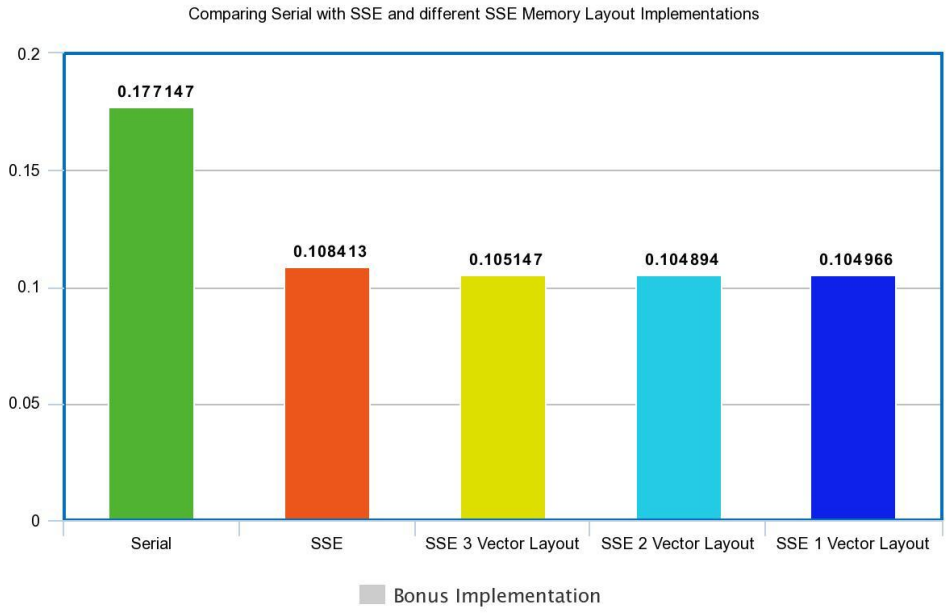


Σχήμα 5: Comparing Serial with SSE,Pthreads and MPI Implementation with 4 Processes

3.4 BONUS: SSE Memory Layouts

Στην τελευταία περίπτωση, στην παραλληλοποίηση του reference code με διαφορετικά SSE Memory Layouts παρατηρούμε καλύτερη απόδοση σε σύγκριση με την αρχική SSE υλοποίηση , κάτι το οποίο περιμέναμε διότι με τη συγκεκριμένη υλοποίηση αποφεύγουμε τα cache misses. Όσον αφορά τα 3 Memory Layouts, δεν παρατηρούμε κάποια σημαντική διαφορά στην απόδοση τους. Στο γράφημα που ακολουθεί, καλύτερη απόδοση φαίνεται να έχει η υλοποίηση με τα 2 vectors, συμπέρασμα βέβαια το οποίο αλλάζει απο εκτέλεση σε εκτέλεση του κώδικα.

Πιο συγκεκριμένα, στην περίπτωση των 3 vectors έχουμε $Speedup = 1.031$, στην περίπτωση των 2 vectors $Speedup = 1.033$ και όταν έχουμε 1 vector $Speedup = 1.032$, σε σχέση πάντα με την αρχική υλοποίηση του SSE.



Σχήμα 6: Comparing different SSE Memory Layouts