Αρχιτεκτονική Παράλληλων και Κατανεμημένων Υπολογιστών

Άσκηση 2

Σπυριδάκης Χρήστος

Εισαγωγή

Σε αυτήν την άσκηση χρειάστηκε να επιταχύνουμε τον υπολογισμό του **ω statistic** αρχικά με την χρήση μόνο του SSE (Streaming SIMD Extensions) και στην συνέχεια συνδυάζοντας το με το MPI (Message Passing Interface). Για να περιγραφτεί όσο πιο αναλυτικά γίνεται η υλοποίηση της άσκησης, η αναφορά έχει χωριστεί σε 3 ενότητες:

1. **Υλοποιήσεις και Σχολιασμοί**
2. **Εκτέλεση**
3. **Συμπεράσματα**

Υλοποιήσεις και Σχολιασμοί

Serial

Για τον σειριακό υπολογισμό πάνω στη λογική του οποίου έγιναν οι παραπάνω υλοποιήσεις χρησιμοποιήθηκε αυτούσιο το sample αρχείο, στο οποίο πραγματοποιήθηκαν οι κατάλληλες αλλαγές. Σε αυτό έχουν προστεθεί μαζί με τις υλοποιήσεις και κάποια μη λειτουργικού χαρακτήρα κομμάτια είτε για την αποφυγή σφαλμάτων είτε για να απλοποιήθουν κάποιες ενέργειες.
Μία από αυτές ως παράδειγμα είναι ότι στην αρχή όλων των υλοποιήσεων έχει προστεθεί το παρακάτω κομμάτι κώδικα ώστε να προβλέπεται η περίπτωση που δεν δόθηκαν ορίσματα και υπήρχαν σφάλματα (παρόλο που προβλέπεται στο run script υπάρχει και σε αυτό το σημείο και ο λόγος είναι ότι έτσι δίνεται η δυνατότητα να τρέξει και μεμονωμένα ο κώδικας χωρίς την χρήση του script).

serial_DNA.c

```
…

if (argc !=2 ){
    printf("\n(ERROR) Wrong number of arguments! Input example:\n$ ./serialR 1000\n");
    return -1;
}

...
```

SSE

Σχετικά με το SSE έχουν υλοποιηθεί και οι δύο μέθοδοι οι οποίες περιγράφονται στις διαλέξεις του μαθήματος (χρήση των εντολών load και με την χρήση pointers) ώστε να μπορέσουμε να πάρουμε περισσότερα συμπεράσματα και να μελετήσουμε διαφορές που υπάρχουν στην απόδοση.
Για αυτό τον λόγο θα υπάρξουν επιγραμματικά κομμάτια κώδικα και για τις δύο υλοποιήσεις καθώς και σύντομοι σχολιασμοί.

1)Στην αρχή του κώδικα πριν ξεκινήσει ο ζητούμενος υπολογισμός δημιουργήθηκαν οι παρακάτω μεταβλητές, οι οποίες έχουν βοηθητικό χαρακτήρα:

Στην υλοποίηση με **pointer**	Στην υλοποίηση με **load**
int alignb = 16; **int** _numV = 4; **int** _sseL = N/4, tail = N - N % 4;	**int** alignb = 16; **int** _numV = 4; **int** _sseL = N - N % 4, tail = _sseL;

Όπου:

- _sseL : Είναι το όριο μέχρι το οποίο θα γίνει ο υπολογισμός με SSE, μπορούμε να δούμε ότι ανάλογα με την υλοποίηση διαφέρουν οι τιμές.

- tail : Είναι το σημείο από το οποίο θα ξεκινήσουμε να κάνουμε με σειριακό τρόπο (για τις εναπομείναντες τιμές) τους υπολογισμούς.

Μπορεί από νωρίς να παρατηρηθεί ότι μεταβλητές οι οποίες ξεκινούν με underscore συσχετίζονται με το καθαρά λειτουργικό κομμάτι της SSE υλοποίησης.

2)Δημιουργία Μεταβλητών για το SSE κομμάτι:

Στην υλοποίηση με **pointer**	Στην υλοποίηση με **load**

__m128 * _mVec = (__m128 *)mVec;
__m128 * _nVec = (__m128 *)nVec;
__m128 * _LVec = (__m128 *)LVec;
__m128 * _RVec = (__m128 *)RVec;
__m128 * _CVec = (__m128 *)CVec;
__m128 * _FVec = (__m128 *)FVec;
	__m128 _mVec;
	__m128 _nVec;
	__m128 _LVec;
	__m128 _RVec;
	__m128 _CVec;
	__m128 _FVec;

Μπορούμε να δούμε ότι η μόνη διαφορά και στις δύο υλοποιήσεις είναι ότι στην μία υλοποίηση δημιουργούμε απλές μεταβλητές τύπου __m128 ενώ στην άλλη pointer σε μεταβλητές αυτού του τύπου, που δείχνουν αμέσως στους πίνακες που έχουμε δημιουργήσει.

3) Υλοποίηση των υπολογισμών

Στην υλοποίηση με **pointer**

```
...

for (int i = 0; i < _sseL; i++) {

  _num_0 = _mm_add_ps(_LVec[i], _RVec[i]);
  _tmp   = _mm_sub_ps(_mVec[i], _1);
  _num_1 = _mm_div_ps(_mm_mul_ps(_mVec[i], _tmp), _2);
  _tmp   = _mm_sub_ps(_nVec[i], _1);
  _num_2 = _mm_div_ps(_mm_mul_ps(_nVec[i], _tmp), _2);
  _tmp   = _mm_add_ps(_num_1, _num_2);
  _num   = _mm_div_ps(_num_0, _tmp);

  _tmp   = _mm_sub_ps(_CVec[i], _LVec[i]);
  _den_0 = _mm_sub_ps(_tmp, _RVec[i]);
  _den_1 = _mm_mul_ps(_mVec[i], _nVec[i]);
  _den   = _mm_div_ps(_den_0, _den_1);

  _tmp   = _mm_add_ps(_den, _001);
  _FVec[i] = _mm_div_ps(_num, _tmp);

  _maxF  = _mm_max_ps(_FVec[i], _maxF);

}

...
```

Στην υλοποίηση με **load**

```
…

for (int i = 0; i < _sseL; i += 4) {
  _mVec  = _mm_load_ps(mVec + i);
  _nVec  = _mm_load_ps(nVec + i);
  _LVec  = _mm_load_ps(LVec + i);
  _RVec  = _mm_load_ps(RVec + i);
  _CVec  = _mm_load_ps(CVec + i);
  _FVec  = _mm_load_ps(FVec + i);

  _num_0 = _mm_add_ps(_LVec, _RVec);
  _tmp   = _mm_sub_ps(_mVec, _1);
  _num_1 = _mm_div_ps(_mm_mul_ps(_mVec, _tmp), _2);
  _tmp   = _mm_sub_ps(_nVec, _1);
  _num_2 = _mm_div_ps(_mm_mul_ps(_nVec,  _tmp), _2);
  _tmp   = _mm_add_ps(_num_1, _num_2);
  _num   = _mm_div_ps(_num_0, _tmp);

  _tmp   = _mm_sub_ps(_CVec, _LVec);
  _den_0 = _mm_sub_ps(_tmp, _RVec);
  _den_1 = _mm_mul_ps(_mVec, _nVec);
  _den   = _mm_div_ps(_den_0, _den_1);

  _tmp   = _mm_add_ps(_den, _001);
  _FVec  = _mm_div_ps(_num, _tmp);

  _maxF  = _mm_max_ps(_FVec, _maxF);
}
...
```

Στην υλοποίηση με load χρειάζεται να αυξάνουμε ανά 4 το i καθώς “διαβάζουμε” (εδώ είναι και ένα σημείο που μας βοηθάει το align για αυτόν τον λόγο αλλάξαμε την malloc των πινάκων σε _mm_malloc()) τέσσερις float από τους πίνακες ανά φορά και τους αποθηκεύουμε σε μία μεταβλητή τύπου __m128, ενώ στην υλοποίηση με pointer μπορούμε να το φανταστούμε ως ένα πίνακα από __m128 όπου έχει ήδη γίνει αυτό για αυτό αυξάνουμε απλά το i κατά ένα.
Χρησιμοποιούμε την μεταβλητή _tmp για ενδιάμεσους υπολογισμούς, την οποία χρησιμοποιούμε στην συνέχεια για την πραγματοποίηση των υπολογισμών ακριβώς στην λογική της σειριακής υλοποίησης με μόνη διαφορά ότι όλες οι πράξεις γίνονται με την χρήση των SSE Intrinsics πλέον.

4) Μέγιστο SSE και εναπομείναντες τιμές

Αφού γίνουν τα παραπάνω βρίσκουμε το max ανάμεσα στις τιμές των float που υπάρχουν στη __m128 μεταβλητή (_maxF) και έχοντας αυτό υπόψιν με σειριακό τρόπο πλέον το συγκρίνουμε με τα max των τελευταίων στοιχείων που δεν μπόρεσαν να υπολογιστούν με SSE.

MPI-SSE

Αφού είχε υλοποιηθεί ο υπολογισμός με SSE αυτό που χρειάστηκε να κάνουμε είναι να προσθέσουμε στον κώδικα κομμάτια ώστε να μπορεί να δημιουργεί processes και να ανταλλάσσει πληροφορίες μεταξύ τους ώστε να επιταχύνει ακόμα περισσότερο τον υπολογισμό του max (χρησιμοποιήθηκε η υλοποίηση του SSE με το load).

0)Εγκατάσταση απαραίτητων πακέτων.

```
$ sudo apt-get install mpich
```

Για να μπορούμε να κάνουμε compile, να δημιουργήσουμε και να κάνουμε τους υπολογισμούς με MPI χρειάστηκε να εγκαταστήσουμε το mpich, σε Debian-Based σύστημα χρειάστηκε απλά να εκτελεστεί η παραπάνω εντολή.

1)Δημιουργία των processes

```
…

MPI_Init(NULL, NULL);
int wSize,wRank;
MPI_Comm_size(MPI_COMM_WORLD, &wSize);
MPI_Comm_rank(MPI_COMM_WORLD, &wRank);

...
```

Με αυτόν τον τρόπο (όταν χρησιμοποιήσουμε την εντολή mpiexec περνώντας ως όρισμα των αριθμό των processes που θέλουμε να δημιουργήσουμε) δημιουργούμε τα processes και παίρνουμε για αυτά βασικές πληροφορίες που μας χρειάζονται στην συνέχεια (**wSize** είναι ο αριθμός των processes που δημιουργήθηκαν και το **wRank** είναι ο σειριακός αύξων αριθμός του συγκεκριμένου process).

2)Όρια για τους υπολογισμούς του συγκεκριμένου Process

```
…

int Np     =  N / wSize - (N / wSize) % 4;
int pStart =  Np * wRank;
int pEND   =  Np * (wRank+1)-1;

if(wSize==wRank+1){
  Np=Np+(N-pEND-1);
  pEND=N-1;
}

int alignb =  16;
int iters  =  1000;
int _numV  =  4;
int _PsseL =  pEND - Np % 4, Ptail = _PsseL+1;

...
```

Για να επιταχύνουμε τον υπολογισμό, παραλληλοποιούμε τους υπολογισμούς ώστε κάθε process να υπολογίσει ένα μέρος των πινάκων. Σε αυτό το σημείο υπολογίζουμε τα όρια στα οποία θα έχει πρόσβαση το κάθε process με γνώμονα το id που έχει το καθένα.

3) Υπολογισμοί

Στην συνέχεια το κάθε Process ακολουθεί την διαδικασία του SSE ώστε να υπολογίσει το δικό του max, μόνο όμως στα όρια που του επιτρέπεται.

4) Υπολογισμός χρόνου και Max

```
…

MPI_Barrier(MPI_COMM_WORLD);
time1 = gettime();
timeTotal += time1 - time0;


float *Pmaxs=NULL;
if (wRank == FATHER) {
  Pmaxs = malloc(sizeof(float) * wSize);
}

MPI_Gather(&maxF, 1, MPI_FLOAT, Pmaxs, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);

...
```

Αφού έχουν γίνει οι απαραίτητοι υπολογισμοί, το κάθε process έχει βρει το max του, το μόνο που χρειάζεται είναι να συγκριθεί αυτή η τιμή με τις υπόλοιπες. Μεταφέρουμε λοιπόν τα max του κάθε Process σε μία θέση του πίνακα Pmaxs στο Process 0 με την χρήση του MPI_Gather ώστε το Process 0 να συγκρίνει τις τιμές και να βρει το ολικο max. Για να μετρηθεί ο χρόνος που χρειάστηκε χρησιμοποιείται το MPI_Barrier ώστε να περιμένει το Process 0 μέχρι να τελειώσουν όλοι οι υπολογισμοί και μετά να υπολογίσει τον χρόνο (έτσι ο ολικός χρόνος να είναι ο χρόνος του Process που καθυστέρησε περισσότερο).

Εκτέλεση

Για να εκτελέσουμε τον κώδικα το μόνο που χρειάζεται είναι να είμαστε σε ένα Linux-Based σύστημα με εγκατεστημένα ήδη το gcc, make, mpich ώστε να μπορεί να γίνει compile και να τρέξει ο κώδικας.
Προκειμένου να εκτελέσουμε αυτόματα όλες τις περιπτώσεις οι οποίες ζητούνται στην εκφώνηση μεταβαίνουμε με ένα terminal στο directory του project και εκτελούμε την παρακάτω εντολή:

- $ sh run.sh

Αν χρειάζεται να εκτελέσουμε μόνο για ένα συγκεκριμένο ζευγάρι N και P τον κώδικα, αρκεί να μεταβούμε με ένα terminal στο directory του project και εκτελούμε την παρακάτω εντολή:

- $ sh run.sh {N} {P}

Σημείωση: Αν έχουμε πρόβλημα με τα δικαιώματα αρκεί πριν τρέξουμε το πρόγραμμα να εκτελέσουμε την εντολή

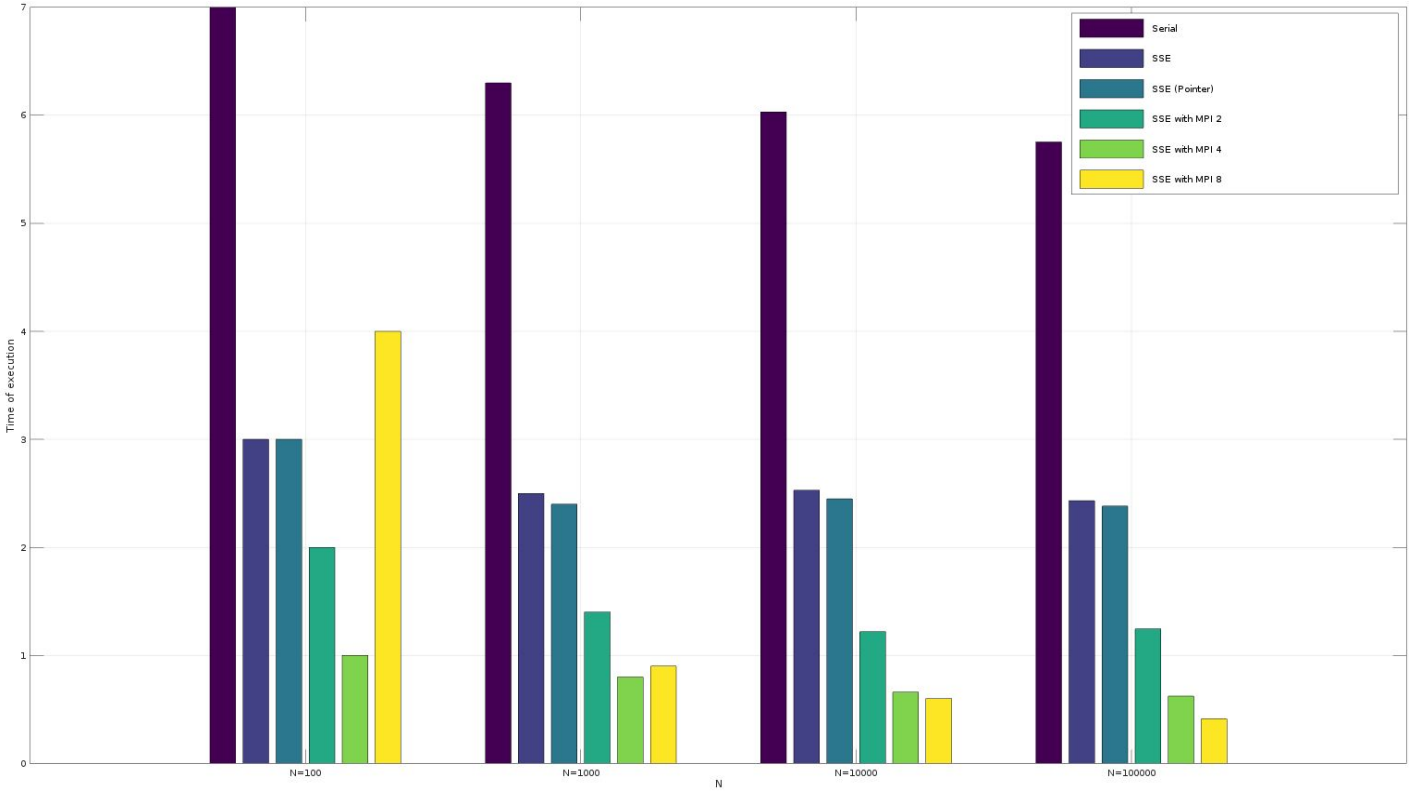
- $ chmod +x run.sh

Σε περίπτωση επιτυχίας η έξοδος που θα εμφανιστεί στην οθόνη μας είναι της μορφής:

```
Building project...
gcc -Wall -std=c99 -o serialR serial_DNA.c
gcc -Wall -std=c99 -msse4.2 -o sseR sse_DNA.c
gcc -Wall -std=c99 -msse4.2 -o sseRP sse_DNA_PTR.c
mpicc -Wall -std=c99 -msse4.2 -o mpiR sse-mpi_DNA.c

********** N=100 **********
Serial
Time 0.000006 Max 15.967351

SSE
Time 0.000003 Max 15.967351

SSE (Pointer Implementation)
Time 0.000003 Max 15.967351

MPI
Number of tasks 2
Time 0.000002 Max 15.967351

MPI
Number of tasks 4
Time 0.000001 Max 15.967351

MPI
Number of tasks 8
Time 0.000001 Max 15.967351

...
```

Συμπεράσματα

Το παρακάτω Bargraph απεικονίζει τους χρόνους για κάθε υλοποίηση ζευγαριού N και P (ο Y άξονας **ΔΕΝ** είναι ο ακριβής χρόνος εκτέλεσης. Για κάθε N, **οι χρόνοι έχουν πολλαπλασιαστεί με μία κοινή σταθερά** ώστε να μπορούν να εμφανιστούν στο ίδιο διάγραμμα όλες οι περιπτώσεις, για να μπορέσουμε να συγκρίνουμε τους χρόνους της κάθε υλοποίησης για κάθε περίπτωση)



Εικόνα 1.1: Bargraph για χρόνους εκτέλεσης κάθε N και P
Τα παραπάνω αποτελέσματα προέκυψαν από επεξεργαστή i7 6700HQ

Αν παρατηρήσουμε το παραπάνω διάγραμμα βγάζουμε τα εξής συμπεράσματα:

1) Με την χρήση του SSE μπορεί να υπάρξει περίπου από 2.3 μέχρι 2.6 speedup ανάλογα με τον τρόπο που θα υλοποιηθεί (Pointer/Load). Το οποίο είναι μικρότερο από το θεωρητικό 4 λόγω του ότι με μία πράξη σε SSE γίνονται τέσσερις του Serial.
2) Η χρήση του SSE με pointer είναι αιτία για ~0.16 μεγαλύτερο speedup από αυτήν της υλοποίησης με load (οι λόγοι είναι ότι στην υλοποίηση με pointer δεν υπάρχει ο έξτρα χρόνος που χρειάζεται για το load).
3) Με την δημιουργία των processes και την χρήση του MPI ο υπολογισμός μπορεί να γίνει από 4 μέχρι και 15 φορές πιο γρήγορα (ανάλογα με τον αριθμό των processes που θα επιλεχθούν). Το οποίο είναι επίσης μικρότερο από τις θεωρητικές τιμές στις οποίες θα έπρεπε το speedup να είναι από 8 μέχρι 32 για τον λόγο του ότι αν είχαμε π.χ. SSE και MPI 8 Processes για “1 πράξη” του θα έκανε την ίδια δουλειά που θα χρειαζόντουσαν “32 πράξεις” στο Serial.

4) Για μικρού μήκους (αριθμός N) arrays (π.χ N=100) το ότι αυξάνουμε τον αριθμό των processes ΔΕΝ σημαίνει ότι υπάρχει υποχρεωτικά μείωση του χρόνου εκτέλεσης.

Επίσης στον υπολογισμό του max, υπάρχουν μικρές διαφορές της τιμής του όταν το υπολογίζουμε σειριακά με όταν το υπολογίζουμε χρησιμοποιώντας SSE, ο λόγος είναι ότι οι πράξεις σε αυτήν την περίπτωση γίνονται με λίγο διαφορετική σειρά.