ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ Η/Υ
Αρχιτεκτονική Παράλληλων και Κατανεμημένων Υπολογιστών (ΗΡΥ418)
Εαρινό Εξάμηνο 2018

Άσκηση 2: Παραλληλισμός με χρήση SIMD και MPI

Ομάδα Εργασίας LAB41835594:
Νικόλαος Κυπαρισσάς, 2012030112
Σοφία Μαραγκού, 20120300078

1. Εισαγωγή

Ο σκοπός της 2ης άσκησης ήταν να επιταχύνουμε τον υπολογισμό μιας απλοποιημένης μορφής του *ω statistic* σε ακολουθίες DNA με την χρήση των Streaming SIMD Extensions (SSE) και του Message Passing Interface (MPI).

2. Ο σειριακός κώδικας και οι αλλαγές που εφαρμόστηκαν

Ο σειριακός κώδικας που μας δόθηκε υπολογίζει τα μεγέθη που απαιτούνται για την εξαγωγή του ω για κάθε μία από τις N θέσεις μιας ακολουθίας DNA με σκοπό την εύρεση της μέγιστης τιμής του ω. Το αντίστοιχο μέρος του κώδικα που μας ενδιαφέρει είναι το εξής:

```
for(int i=0; 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;
}
```

2.1 Streaming SIMD Extensions 4.2 (SSE 4.2)

Ο παραπάνω κώδικας αντικαταστάθηκε με το κομμάτι που ακολουθεί, το οποίο αποτελείται από τους ίδιους υπολογισμούς υλοποιημένους με εντολές SSE (128-bit).

```
//N/4: SSE processes 4 float numbers simultaneously
for(int i=0; i<N/4; i++){
	//float num_0 = LVec[i]+RVec[i];
	mLVec = _mm_load_ps(LVec+(i*4));
	mRVec = _mm_load_ps(RVec(i*4));
	num_0 = _mm_add_ps(mLVec, mRVec);
	//float num_1 = mVec[i]*(mVec[i]-1.0)/2.0;
	mmVec = _mm_load_ps(mVec+(i*4));
	num_1 = _mm_mul_ps(_mm_div_ps(_mm_sub_ps(mmVec,
		_mm_set_ps(1, 1, 1, 1)), _mm_set_ps(2, 2, 2, 2)), mmVec);
	//float num_2 = nVec[i]*(nVec[i]-1.0)/2.0;
	mnVec = _mm_load_ps(nVec+(i*4));
	num_2 = _mm_mul_ps(_mm_div_ps(_mm_sub_ps(mnVec,
		_mm_set_ps(1, 1, 1, 1)), _mm_set_ps(2, 2, 2, 2)), mnVec);
	//float num = num_0/(num_1+num_2);
	num = _mm_div_ps(num_0, _mm_add_ps(num_1, num_2));
	//float den_0 = CVec[i]-LVec[i]-RVec[i];
	mCVec = _mm_load_ps(CVec+(i*4));
	den_0 = _mm_sub_ps(_mm_sub_ps(mCVec, mLVec), mRVec);
	//float den_1 = mVec[i]*nVec[i];
	den_1 = _mm_mul_ps(_mm_load_ps(mVec+(i*4)), _mm_load_ps(nVec+(i*4)));
	//float den = den_0/den_1;
	den = _mm_div_ps(den_0, den_1);
	//FVec[i] = num/(den+0.01);
	mFVec = _mm_div_ps(num, _mm_add_ps(den,
		_mm_set_ps(0.01, 0.01, 0.01, 0.01)));
	//maxF = FVec[i]>maxF?FVec[i]:maxF;
	maxF4 = _mm_max_ps(maxF4, mFVec);
}

maxF = horizontal_max_Vec4(maxF4); //returns max float from m128

// Process the residue data in case N/4 leaves a remainder
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;
}
```

Ένας αριθμός κινητής υποδιαστολής απλής ακρίβειας (float) αναπαρίσταται από 32 bits, που σημαίνει ότι στα 128 bits μιας SSE μεταβλητής εμπεριέχονται μέχρι και 4 αριθμοί floating point, για αυτό και το for-loop πλέον κάνει N/4 επαναλήψεις.

Σε περίπτωση που το N δεν διαιρείται ακριβώς με το 4, το ω των θέσεων DNA που έμειναν έξω από το SSE κομμάτι του κώδικα υπολογίζεται σειριακά.

Τέλος, οι μεταβλητές που χρησιμοποιούνται στις SSE εντολές είναι δηλωμένες ως:

```
__attribute__((aligned(16))) __m128
```

και ο χώρος για κάθε πίνακα δεσμευμένος πλέον ως εξής:

```
(float*)_mm_malloc(sizeof(float)*N, 16);
```

αντί για την αρχική δέσμευση με *malloc()*, έτσι ώστε το λειτουργικό να δεσμεύσει ευθυγραμμισμένα κομμάτια μνήμης, κάτι που είναι απαραίτητο για την αποδοτική χρήση των εντολών SSE.

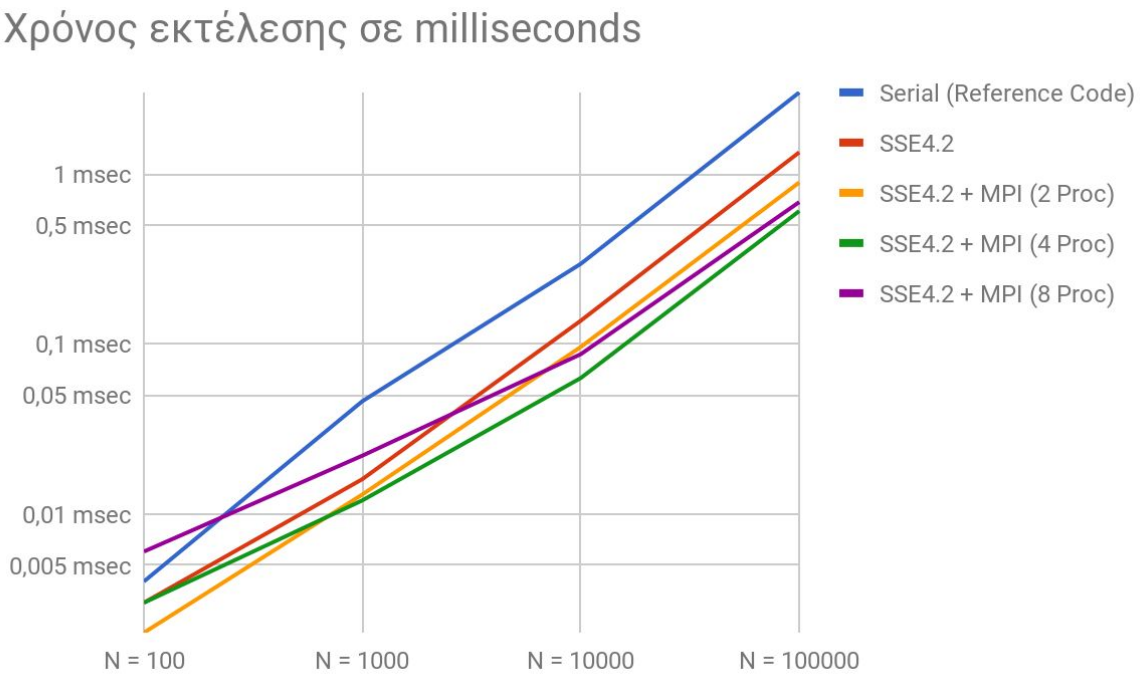
2.2 Message Passing Interface (MPI)

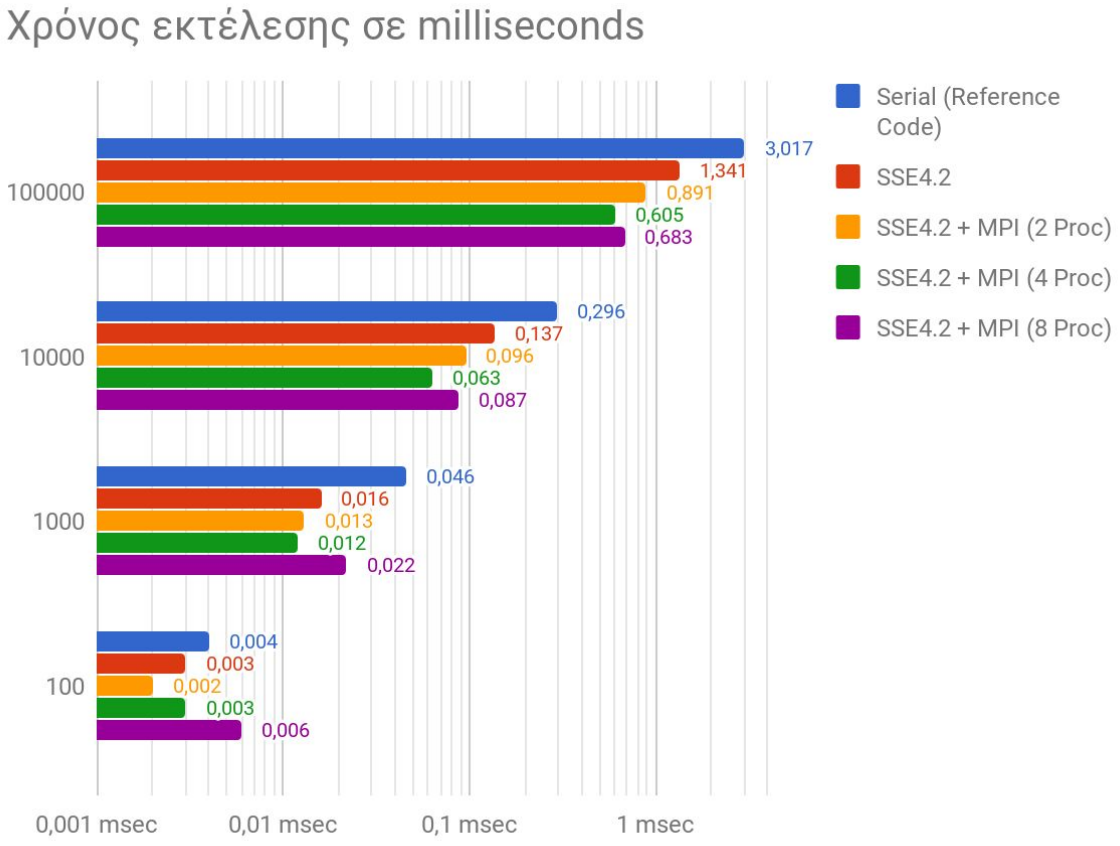
Η υλοποίηση με SSE μας δίνει σημαντικό speedup της τάξης του 2.3x. Με τη χρήση του MPI μπορούμε να βελτιώσουμε ακόμη περισσότερο την απόδοση του προγράμματος, διαμοιράζοντας τον φόρτο εργασίας σε διαφορετικές διεργασίες (processes) που τρέχουν παράλληλα.

Ο κώδικας όπως παρουσιάστηκε παραπάνω δεν αλλάζει σημαντικά. Οι μόνες αλλαγές που συντελούνται είναι η εξής:

1. Όλες οι διεργασίες δημιουργούν και έχουν στη μνήμη τους τους ίδιους πίνακες που απαιτούνται για την επεξεργασία.
2. Όταν έρθει η στιγμή να ξεκινήσει η επεξεργασία, η κάθε διεργασία προσπελαύνει το κομμάτι του DNA που αντιστοιχεί στον αριθμό της. Χρησιμοποιείται δηλαδή το rank/id της κάθε διεργασίας για τον ισομοιρασμό του φόρτου επεξεργασίας μεταξύ των διεργασιών.
3. Στο τέλος της επεξεργασίας, η κάθε διεργασία στέλνει το μέγιστο ω της στη διεργασία με τον αριθμό μηδέν. Η διεργασία με τον αριθμό μηδέν, αφού έχει επεξεργαστεί και αυτή το δικό της κομμάτι του DNA, λαμβάνει τα μέγιστα ω που υπολόγισαν οι υπόλοιπες διεργασίες. Έτσι είναι σε θέση να κρατήσει και να τυπώσει το μέγιστο ω για όλο το DNA.

3. Αποτελέσματα και Συμπεράσματα





Παρατηρούμε ότι η υλοποίηση με SSE μας δίνει σημαντικό speedup της τάξης του 2.3x σε σχέση με τη σειριακή υλοποίηση.

Οι υλοποιήσεις με MPI δίνουν ακόμη μεγαλύτερο speedup όσο τα δεδομένα πληθαίνουν. Αντιθέτως, για μικρό όγκο δεδομένων ο φόρτος της επικοινωνίας μεταξύ των διεργασιών παραμένει κοστοβόρος και ρίχνει την απόδοση.

Τέλος, παρατηρούμε ότι οι 8 διεργασίες αποδίδουν χειρότερα από ότι οι 4 ακόμη και για μεγάλο όγκο δεδομένων. Αυτό συμβαίνει γιατί στην περίπτωσή μας τα προγράμματα έτρεξαν σε επεξεργαστή με 4 πυρήνες, οπότε οι 8 διεργασίες δεν έτρεχαν όλες παράλληλα.

4. Παράρτημα

Πληροφορίες συστήματος μετρήσεων:

1. CPU: Intel Core i5 2410M (2.3 GHz, 2 πυρήνες ~ 4 threads, 3MB L3 Cache)
2. RAM: 8 GB DDR3, 1333 MHz
3. OS: Bash on Ubuntu on Windows 10 (16.04 xenial)

Για να εκτελεστεί το πρόγραμμα, εισάγετε την παρακάτω εντολή:
./script.sh
Το παραπάνω script κάνει compile και εκτελεί διαδοχικά τα *serial.c*, *simd.c* και *simd_mpi.c* με παραμέτρους N = 100, 1.000, 10.000, 100.000 και P = 2, 4, 8.