**Εργασία 3**

**Ονοματεπώνυμο: Νικόλαος Θεοδώρου**

**Αριθμός Ταυτότητας: 1030496**

**Ερώτημα:**

**0) Προεργασια και Τροποποιημένη main:**

int myMain(int argc, char \*argv[], char \*exec\_type)

{

int i;

char \*log\_file = calloc(1, strlen("logfile\_") + strlen(exec\_type) + 1);

strcpy(log\_file, "logfile\_");

strcat(log\_file, exec\_type);

char \*data\_file = calloc(1, strlen("data\_") + strlen(exec\_type) + 1);

strcpy(data\_file, "data\_");

strcat(data\_file, exec\_type);

FILE \*lfp = fopen(log\_file, "w");

FILE \*dfp = fopen(data\_file, "w");

if (lfp == NULL || dfp == NULL)

{

printf("Please create the ./outputs directory\n");

return -1;

}

if (argc == 3)

{

timeSteps = atoi(argv[1]);

bodies = atoi(argv[2]);

thread\_count = 1;

printf("%%\*\*\* RUNNING WITH VALUES %d timesteps %d bodies and %d thread(s) \*\*\*\n", timeSteps, bodies, thread\_count);

}

else if (argc == 4)

{

timeSteps = atoi(argv[1]);

bodies = atoi(argv[2]);

thread\_count = atoi(argv[3]);

printf("%%\*\*\* RUNNING WITH VALUES %d timesteps %d bodies and %d thread(s) \*\*\*\n", timeSteps, bodies, thread\_count);

}

else if (argc == 2)

{

timeSteps = 10000;

bodies = 200;

thread\_count = atoi(argv[1]);

printf("%%\*\*\* RUNNING WITH DEFAULT VALUES %d timesteps %d bodies and %d thread(s) \*\*\*\n", timeSteps, bodies, thread\_count);

}

else

{

timeSteps = 10000;

bodies = 200;

thread\_count = 1;

printf("%%\*\*\* RUNNING WITH DEFAULT VALUES %d timesteps %d bodies and %d thread(s) \*\*\*\n", timeSteps, bodies, thread\_count);

}

initiateSystemRND(bodies);

// initiateSystem("input.txt");

fprintf(stdout, "Running With %d Bodies for %d timeSteps. Initial state:\n", bodies, timeSteps);

fprintf(stderr, "Running With %d Bodies for %d timeSteps. Initial state:\n", bodies, timeSteps);

fprintf(lfp, "Running With %d Bodies for %d timeSteps. Initial state:\n", bodies, timeSteps);

fprintf(lfp, "Body \t\t\t:\t\tx\t\ty\t\t\tz\t\t|\t\tvx\t\t\tvy\t\t\tvz\t\t\n");

printBodiesInfo(lfp, dfp);

startTime(0);

pthread\_t threads[thread\_count];

int ids[thread\_count];

if (strcmp(exec\_type, "static") == 0 || strcmp(exec\_type, "dynamic") == 0)

{

assert(pthread\_barrier\_init(&barrier, NULL, thread\_count) == 0);

}

if (strcmp(exec\_type, "static") == 0)

{

for (i = 0; i < thread\_count; i++)

{

ids[i] = i;

assert(pthread\_create(&threads[i], NULL, (void \*)n\_body\_pThreads\_static, (void \*)&ids[i]) == 0);

}}

else if (strcmp(exec\_type, "dynamic") == 0)

{

for (i = 0; i < thread\_count; i++)

{

ids[i] = i;

assert(pthread\_create(&threads[i], NULL, (void \*)n\_body\_pThreads\_dynamic, (void \*)&ids[i]) == 0);

}

}

else

{

for (i = 0; i < timeSteps; i++)

{

simulate();

#ifdef DEBUG

int j;

// printf("\nCycle %d\n",i+1);

for (j = 0; j < bodies; j++)

fprintf(dfp, "%d\t%d\t%lf\t%lf\t%lf\n", i, j, positions[j].x, positions[j].y, positions[j].z);

#endif

}

}

if (strcmp(exec\_type, "static") == 0 || strcmp(exec\_type, "dynamic") == 0)

{

for (i = 0; i < thread\_count; i++)

{

assert(pthread\_join(threads[i], NULL) == 0);

}}

stopTime(0);

fprintf(lfp, "\nLast Step = %d\n", i);

printBodiesInfo(lfp, dfp);

printf("\nSimulation Time:");

elapsedTime(0);

fclose(lfp);

fclose(dfp);

return 0;

}

int main(int argc, char \*argv[])

{

pthread\_mutex\_init(&mutex, NULL);

printf("Static\n");

fflush(stdout);

// thread\_count = 4;

myMain(argc, argv, "static");

printf("Dynamic\n");

fflush(stdout);

myMain(argc, argv, "dynamic");

// printf("Serial\n");

// fflush(stdout);

// myMain(argc, argv, "serial");

return 0;

}

**Τροποποίησα τη main για να δημιουργώ τα threads από πριν την εκτέλεση του προγράμματος και έξω από το βρόγχο των timesteps, διότι είχα τεράστιο overhead κατα τη δημιουργία threads και κατα τη συνάρτηση join των pthreads. Επίσης όπως θα δείτε πιο κάτω παραλληλοποίησα μόνο την compute accelerations, αφού είναι η πιο χρονοβόρα συνάρτηση και εκτελείτε για το 88% του χρόνου εκτέλεσης.**

**1) Συνάρτηση** **n\_body\_pThreads\_static (20% Κώδικας + 5% Σχόλια) (**Courier New 10**):**

**void n\_body\_pThreads\_static(void \*args)**

{

int i;

char \*execution\_type = "static";

int tid = \*(int \*)args;

for (i = 0; i < timeSteps; i++)

{

computeAccelerations(execution\_type, tid);

pthread\_barrier\_wait(&barrier);

if (tid == 0)

{

SimulationTime++;

computePositions();

computeVelocities();

resolveCollisions(execution\_type);

}

pthread\_barrier\_wait(&barrier);

}}

**void computeAccelerations(char \*exec\_type, int tid)**

{

int i, j;

if (strcmp(exec\_type, "static") == 0)

{

computeAccelerations\_staticWorker(tid);

}

**void computeAccelerations\_staticWorker(int tid)**

{

int i, j, start, end;

// Divide the work among the threads

start = tid \* bodies / thread\_count;

end = (tid == thread\_count - 1) ? bodies : (tid + 1) \* bodies / thread\_count;

for (i = start; i < end; i++)

{

accelerations[i].x = 0;

accelerations[i].y = 0;

accelerations[i].z = 0;

for (j = 0; j < bodies; j++)

{

if (i != j)

{

vector sij = {positions[i].x - positions[j].x, positions[i].y - positions[j].y, positions[i].z - positions[j].z};

vector sji = {positions[j].x - positions[i].x, positions[j].y - positions[i].y, positions[j].z - positions[i].z};

double mod = sqrt(sij.x \* sij.x + sij.y \* sij.y + sij.z \* sij.z);

double mod3 = mod \* mod \* mod;

double s = GravConstant \* masses[j] / mod3;

vector S = {s \* sji.x, s \* sji.y, s \* sji.z};

accelerations[i].x += S.x;

accelerations[i].y += S.y;

accelerations[i].z += S.z;

}}}}

**Η συνάρτηση n\_body\_pThreads\_static είναι το σημείο εισόδου για κάθε νήμα, το οποίο παίρνει ένα αναγνωριστικό νήματος. Μέσα στον βρόχο με τα χρονικά βήματα, η συνάρτηση computeAccelerations καλείται με έναν τύπο εκτέλεσης "static" και το αναγνωριστικό νήματος ως ορίσμα. Στη συνέχεια, η συνάρτηση pthread\_barrier\_wait χρησιμοποιείται για τον συγχρονισμό όλων των νημάτων πριν προχωρήσει στο επόμενο βήμα της προσομοίωσης.**

**Εάν το αναγνωριστικό νήματος είναι 0, τότε ο συνολικός χρόνος προσομοίωσης αυξάνεται και οι θέσεις, οι ταχύτητες και οι συγκρούσεις ενημερώνονται. Ένα άλλο εμπόδιο χρησιμοποιείται για τον συγχρονισμό όλων των νημάτων πριν ξεκινήσει το επόμενο χρονικό βήμα.**

**Η συνάρτηση computeAccelerations καθορίζει τον τύπο εκτέλεσης και καλεί την αντίστοιχη συνάρτηση υπολογισμού επιτάχυνσης. Σε αυτήν την περίπτωση, εάν ο τύπος εκτέλεσης είναι "static", καλείται η συνάρτηση computeAccelerations\_staticWorker.**

**Η συνάρτηση computeAccelerations\_staticWorker υπολογίζει τις επιταχύνσεις ενός υποσυνόλου σωμάτων που έχουν εκχωρηθεί στο νήμα κλήσης. Η εργασία κατανέμεται εξίσου μεταξύ των νημάτων χρησιμοποιώντας στατικό προγραμματισμό, όπου σε κάθε νήμα εκχωρείται ένα συνεχόμενο κομμάτι σωμάτων για υπολογισμό. Οι δείκτες έναρξης και λήξης του κομματιού υπολογίζονται με βάση το αναγνωριστικό νήματος και τον συνολικό αριθμό των νημάτων. Έτσι εκμεταλευόμαστε τη χωρικη τοπικότητα.**

**2) Συνάρτηση n\_body\_pThreads\_dynamic** **(20% Κώδικας + 5% Σχόλια) (**Courier New 10**):**

**void n\_body\_pThreads\_dynamic(void \*args)**

{

int i;

char \*execution\_type = "dynamic";

int tid = \*(int \*)args;

for (i = 0; i < timeSteps; i++)

{

computeAccelerations(execution\_type, tid);

pthread\_barrier\_wait(&barrier);

if (tid == 0)

{

Gstart = 0;

SimulationTime++;

computePositions();

computeVelocities();

resolveCollisions(execution\_type);

}

pthread\_barrier\_wait(&barrier);

}

}

**void computeAccelerations(char \*exec\_type, int tid)**

{

int i, j;

if (strcmp(exec\_type, "static") == 0)

{

computeAccelerations\_staticWorker(tid);

}

else if (strcmp(exec\_type, "dynamic") == 0)

{

computeAccelerations\_dynamicWorker();

}

**void computeAccelerations\_dynamicWorker()**

{

int start = -1, end = -1;

int i, j;

while (Gstart < bodies)

{

pthread\_mutex\_lock(&mutex);

start = Gstart;

if (start >= bodies)

{

pthread\_mutex\_unlock(&mutex);

return;

}

end = start + CHUNK\_SIZE;

if (end > bodies)

{

end = bodies;

}

Gstart = end;

pthread\_mutex\_unlock(&mutex);

for (i = start; i < end; i++)

{

accelerations[i].x = 0;

accelerations[i].y = 0;

accelerations[i].z = 0;

for (j = 0; j < bodies; j++)

{

if (i != j)

{

vector sij = {positions[i].x - positions[j].x, positions[i].y - positions[j].y, positions[i].z - positions[j].z};

vector sji = {positions[j].x - positions[i].x, positions[j].y - positions[i].y, positions[j].z - positions[i].z};

double mod = sqrt(sij.x \* sij.x + sij.y \* sij.y + sij.z \* sij.z);

double mod3 = mod \* mod \* mod;

double s = GravConstant \* masses[j] / mod3;

vector S = {s \* sji.x, s \* sji.y, s \* sji.z};

accelerations[i].x += S.x;

accelerations[i].y += S.y;

accelerations[i].z += S.z;

}}}}}}

**Η συνάρτηση n\_body\_pThreads\_dynamic είναι παρόμοια με την n\_body\_pThreads\_static, με μερικές βασικές διαφορές. Όπως και πριν, η συνάρτηση παίρνει ένα όρισμα κενού δείκτη που μεταδίδεται σε έναν ακέραιο που αντιπροσωπεύει το αναγνωριστικό νήματος. Μέσα στο βρόχο με τα βήματα του χρόνου, η συνάρτηση computeAccelerations καλείται με έναν τύπο εκτέλεσης "dynamic" και χωρίς όρισμα νήμα ID. Στη συνέχεια, χρησιμοποιείται ένα φράγμα για τον συγχρονισμό όλων των νημάτων πριν προχωρήσει στο επόμενο βήμα της προσομοίωσης.**

**Εάν το αναγνωριστικό νήματος είναι 0, τότε ο συνολικός χρόνος προσομοίωσης αυξάνεται και οι θέσεις, οι ταχύτητες και οι συγκρούσεις ενημερώνονται. Ένα άλλο barrier χρησιμοποιείται για τον συγχρονισμό όλων των νημάτων πριν ξεκινήσει το επόμενο χρονικό βήμα.**

**Η συνάρτηση computeAccelerations καθορίζει τον τύπο εκτέλεσης και καλεί την αντίστοιχη συνάρτηση υπολογισμού επιτάχυνσης. Σε αυτήν την περίπτωση, εάν ο τύπος εκτέλεσης είναι "dynamic", καλείται η συνάρτηση computeAccelerations\_dynamicWorker.**

**Η συνάρτηση computeAccelerations\_dynamicWorker υπολογίζει τις επιταχύνσεις ενός υποσυνόλου σωμάτων που έχουν εκχωρηθεί στο νήμα κλήσης. Η εργασία χωρίζεται χρησιμοποιώντας δυναμικό προγραμματισμό, όπου κάθε νήμα αποκτά επανειλημμένα ένα κλείδωμα σε ένα mutex και ανακτά ένα κομμάτι σωμάτων για να υπολογίσει μέχρι να υποστούν επεξεργασία όλα τα σώματα. Το μέγεθος του κομματιού καθορίζεται από τη σταθερά CHUNK\_SIZE και η καθολική μεταβλητή Gstart χρησιμοποιείται για την παρακολούθηση του επόμενου σώματος προς υπολογισμό. Μόλις ένα νήμα ολοκληρώσει τον υπολογισμό του εκχωρημένου κομματιού του, απελευθερώνει το κλείδωμα mutex και ανακτά ένα νέο κομμάτι μέχρι να υποβληθούν σε επεξεργασία όλα τα σώματα.**

**Συνοπτικά, η κύρια διαφορά μεταξύ των προσεγγίσεων στατικού και δυναμικού προγραμματισμού είναι ο τρόπος με τον οποίο η εργασία κατανέμεται μεταξύ των νημάτων. Με τον στατικό προγραμματισμό, σε κάθε νήμα εκχωρείται ένα σταθερό κομμάτι εργασίας, ενώ με τον δυναμικό προγραμματισμό, κάθε νήμα ανακτά ένα νέο κομμάτι εργασίας αφού ολοκληρώσει το προηγούμενο κομμάτι του. Αυτό μπορεί να οδηγήσει σε καλύτερη εξισορρόπηση του φορτίου σε περιπτώσεις όπου η κατανομή της εργασίας δεν είναι ομοιόμορφη, αλλά επιφέρει επίσης κάποια επιβάρυνση λόγω της ανάγκης για συγχρονισμό με χρήση κλειδαριάς mutex.**

**3) Ο ιδανικός αριθμός threads. (5%)**

**4) Αναμενόμενο και καταμετρημένο (Speedup). Efficiency. (5%)**

**5) Γραφική Παράσταση (10%) και Σχολιασμός (~200 λέξεις) για τις λύσεις πιο πάνω. (20%)**

**6) Γραφική Παράσταση Μαζί με OpenMP και Σχολιασμός (~200 λέξεις) για τις λύσεις πιο πάνω. (10%)**