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.

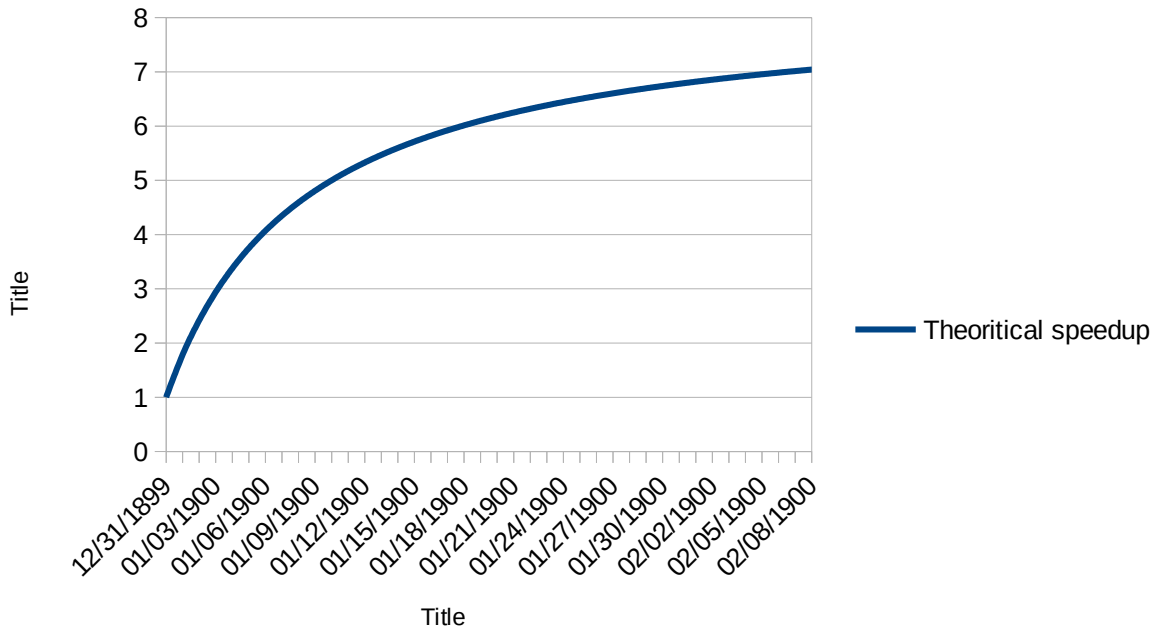
Εάν το αναγνωριστικό νήματος είναι 0, τότε ο συνολικός χρόνος προσομοίωσης αυξάνεται και οι θέσεις, οι ταχύτητες και οι συγκρούσεις ενημερώνονται. Ένα άλλο barrier χρησιμοποιείται για τον συγχρονισμό όλων των νημάτων πριν ξεκινήσει το επόμενο χρονικό βήμα.

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%)
Σύμφωνα με την προηγούμενη εργασία η συνάρτηση Compute Accelerations απασχολούσε το πρόγραμμα για 88% του χρόνου. Επίσης, σύμφωνα με το νόμο του Amdahl η θεωρητική μέγιστη επιτάχυνση που μπορεί να επιτευχθεί με την παραλληλοποίηση της συνάρτησης εξαρτάται από το ποσοστό της συνάρτησης που μπορεί να παραλληλοποιηθεί και τον αριθμό των επεξεργαστών που χρησιμοποιούνται. Για το συγκεκριμένο ποσοστό παραλληλοποίησης, αν χρησιμοποιηθούν 40 επεξεργαστές, η μέγιστη θεωρητική επιτάχυνση που μπορεί να επιτευχθεί βάσει του νόμου του Amdahl είναι 1 / ((1 - 0.88) + (0.88 / 40)) = 7.04.



Μπορούμε να πούμε ότι θεωρητικά μετά τα 14-15 threads η γραμμή του γραφήματος να αρχίζει να ευθυγραμμίζεται.

Number of Thread	PTHREADS - Static -O0 S	PTHREADS-Static -O3 S	PTHREADS - Dynamic -O0 S	PTHREADS-Dynamic -O3 Speedu
1	0.984382529114324	0.969512387104602	0.996118946400091	0.973254016021883
2	1.65686460736202	1.66146020967299	1.36903517732148	1.362575737625655
3	2.1863720918563	2.28914794287193	2.1250120185106	2.234531071486413
4	2.57370572839388	2.76090255818124	2.15085054973624	2.288265232143337
5	2.90368707963973	3.1611319731911	2.25859896761046	2.390238361195785
6	3.16991689967577	3.48896898548382	2.37865109945673	2.525817101832464
7	3.39385774802887	3.78743845412512	2.52013283165705	2.710734077523715
8	3.56586265232371	4.02900886381955	2.60811863148737	2.822162413490944
9	3.69010386734984	4.17519964481866	2.69672935358955	2.933280360032515
10	3.83398308522085	4.33900984654512	2.7589201315236	3.002536995120417
11	3.91083635051864	4.40802417517575	2.85338823480984	3.123237380884006
12	4.06123003339529	4.52907080507148	3.00580465226289	3.223755438890306
13	4.06950627771378	4.42052981581709	3.03765597865474	3.197011901747284
14	4.10770722597328	4.34099784497394	3.11557984005328	3.211195563081559
15	4.16332804158414	4.28178739611366	3.14845751388848	3.188409091196085
16	4.23205147654657	4.19262148432084	3.21093078040357	3.196904636651213
17	4.30002016075026	4.09141434927878	3.31104858918572	3.198244955164412
18	4.23489553924337	3.95777330188014	3.29519075012947	3.127527568691035
19	4.30446643668742	3.88654969049981	3.39205480670848	3.150136765838795
20	4.17365887568472	3.7979319696376	3.37901216148945	3.057015456996577
21	3.07937725905135	2.48350192459105	3.24295357579048	2.997472032992317
22	3.14712337445918	2.14262590897192	2.49490725352702	2.091422929615544
23	3.28670994587166	2.24096646449155	2.90227516169479	2.181659201504055
24	3.26045358432531	2.30730148983143	3.18513131965734	2.200201808609406
25	3.36685593425837	2.00251007698759	3.03990750707651	2.217210205138044
26	3.06704972944599	1.99861165748625	3.38578399956188	2.112965406781412
27	3.12678978814376	1.91383643103235	3.27974152052401	2.102376142984859
28	2.94421248268867	1.9625209027907	2.93279946767286	2.000805076418877
29	2.8940529899521	1.92484857540242	3.1632200819879	1.957488011839515
30	3.00875399440865	1.945353667222279	3.02830522139393	1.922306097364607
31	2.76245846122912	1.8605837001078	3.01573124066854	1.842463194517417
32	2.78172657968628	1.92724920477118	2.88096694696533	1.748905029608447
33	2.75099276263412	1.88534560064871	2.78488879771633	1.699633384030655
34	2.80440171537437	1.871018990287	2.79995899513603	1.702737732533285
35	2.88735618836165	1.78235818340642	2.70848207784089	1.651831259385559
36	2.77338616204577	1.82510572060414	2.70689076763872	1.647980310388117
37	2.84240230523487	1.75184264359646	2.58805979521234	1.541473104957232
38	3.01124626432687	1.73478950304636	2.58025364316305	1.497462764328127
39	2.79257401160429	1.69557367972728	2.54525830894707	1.477810312365
40	2.80999288749341	1.6334953890145	2.4543043401757	1.432276303967044

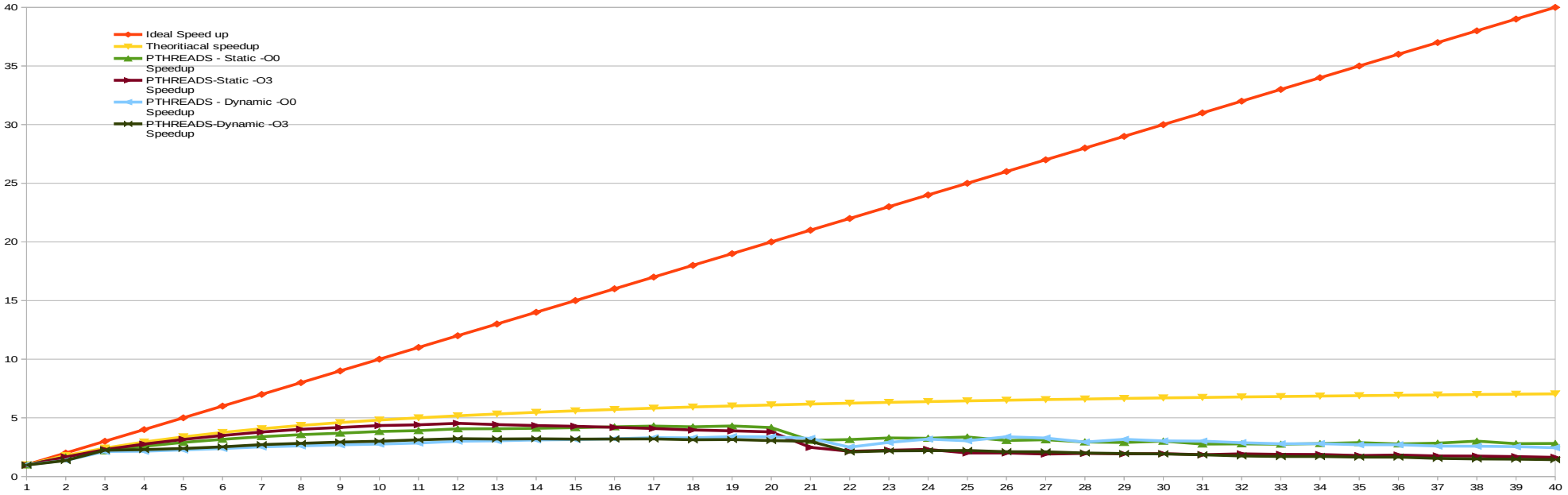
Μετά από πιάσιμο μετρήσεων ο ιδανικός αριθμός threads που επέλεξα για μέγιστο speedup και λιγότερους πόρους για κάθε περίπτωση έχει ως εξής.

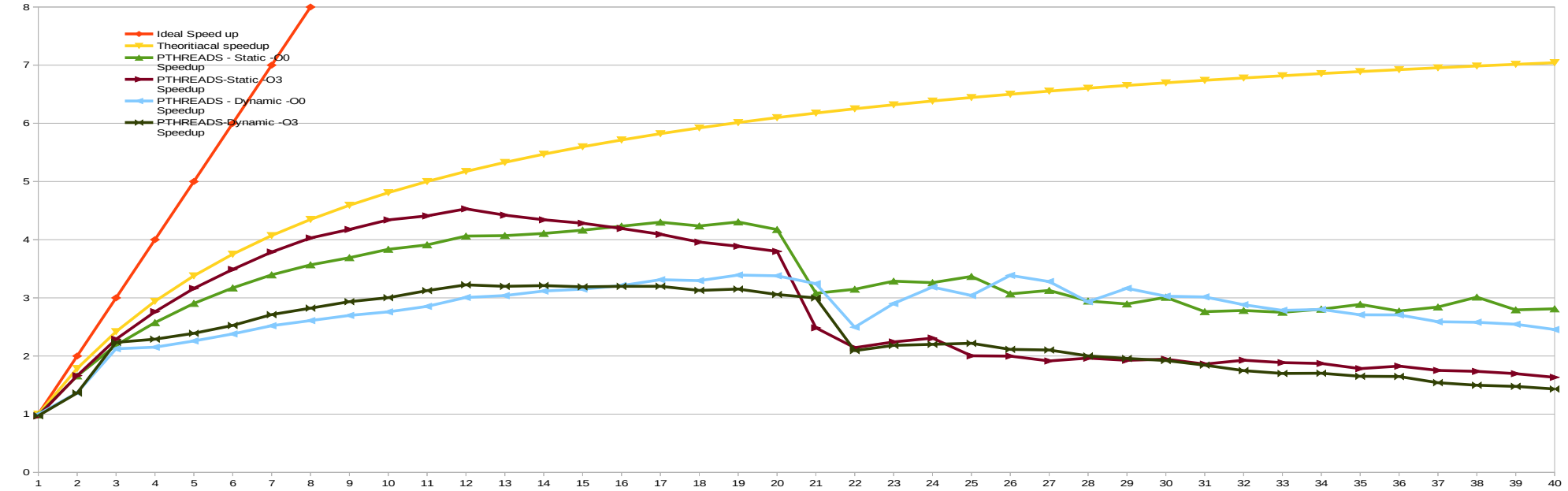
Schedule	Threads	Speedup
Static O0	17	4.30
Static O3	12	4.53
Dynamic O0	17	3.31
Dynamic O3	12	3.22

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

Number of Threads	Ideal Speed up	PTHREADS - Static -O0 Speedup	PTHREADS - Static -O0 Efficiency	PTHREADS-Static -O3 Speedup	PTHREADS - Static -O3 Efficiency	PTHREADS - Dynamic -O0 Speedup	PTHREADS - Dynamic-O0 Efficiency	PTHREADS-Dynamic -O3 Speedup	PTHREADS - Dynamic-O3 Effi
1	1	0.984	0.984	0.970	0.970	0.996	0.996	0.973	0.973
2	2	1.657	0.828	1.661	0.831	1.369	0.685	1.363	0.681
3	3	2.186	0.729	2.289	0.763	2.125	0.708	2.235	0.745
4	4	2.574	0.643	2.761	0.690	2.151	0.538	2.288	0.572
5	5	2.904	0.581	3.161	0.632	2.259	0.452	2.390	0.478
6	6	3.170	0.528	3.489	0.581	2.379	0.396	2.526	0.421
7	7	3.394	0.485	3.787	0.541	2.520	0.360	2.711	0.387
8	8	3.566	0.446	4.029	0.504	2.608	0.326	2.822	0.353
9	9	3.690	0.410	4.175	0.464	2.697	0.300	2.933	0.326
10	10	3.834	0.383	4.339	0.434	2.759	0.276	3.003	0.300
11	11	3.911	0.356	4.408	0.401	2.853	0.259	3.123	0.284
12	12	4.061	0.338	4.529	0.377	3.006	0.250	3.224	0.269
13	13	4.070	0.313	4.421	0.340	3.038	0.234	3.197	0.246
14	14	4.108	0.293	4.341	0.310	3.116	0.223	3.211	0.229
15	15	4.163	0.278	4.282	0.285	3.148	0.210	3.188	0.213
16	16	4.232	0.265	4.193	0.262	3.211	0.201	3.197	0.200
17	17	4.300	0.253	4.091	0.241	3.311	0.195	3.198	0.188
18	18	4.235	0.235	3.958	0.220	3.295	0.183	3.128	0.174
19	19	4.304	0.227	3.887	0.205	3.392	0.179	3.150	0.166
20	20	4.174	0.209	3.798	0.190	3.379	0.169	3.057	0.153
21	21	3.079	0.147	2.484	0.118	3.243	0.154	2.997	0.143
22	22	3.147	0.143	2.143	0.097	2.495	0.113	2.091	0.095
23	23	3.287	0.143	2.241	0.097	2.902	0.126	2.182	0.095
24	24	3.260	0.136	2.307	0.096	3.185	0.133	2.200	0.092
25	25	3.367	0.135	2.003	0.080	3.040	0.122	2.217	0.089
26	26	3.067	0.118	1.999	0.077	3.386	0.130	2.113	0.081
27	27	3.127	0.116	1.914	0.071	3.280	0.121	2.102	0.078
28	28	2.944	0.105	1.963	0.070	2.933	0.105	2.001	0.071
29	29	2.894	0.100	1.925	0.066	3.163	0.109	1.957	0.067
30	30	3.009	0.100	1.945	0.065	3.028	0.101	1.922	0.064
31	31	2.762	0.089	1.861	0.060	3.016	0.097	1.842	0.059
32	32	2.782	0.087	1.927	0.060	2.881	0.090	1.749	0.055
33	33	2.751	0.083	1.885	0.057	2.785	0.084	1.700	0.052
34	34	2.804	0.082	1.871	0.055	2.800	0.082	1.703	0.050
35	35	2.887	0.082	1.782	0.051	2.708	0.077	1.652	0.047
36	36	2.773	0.077	1.825	0.051	2.707	0.075	1.648	0.046
37	37	2.842	0.077	1.752	0.047	2.588	0.070	1.541	0.042
38	38	3.011	0.079	1.735	0.046	2.580	0.068	1.497	0.039
39	39	2.793	0.072	1.696	0.043	2.545	0.065	1.478	0.038
40	40	2.810	0.070	1.633	0.041	2.454	0.061	1.432	0.036

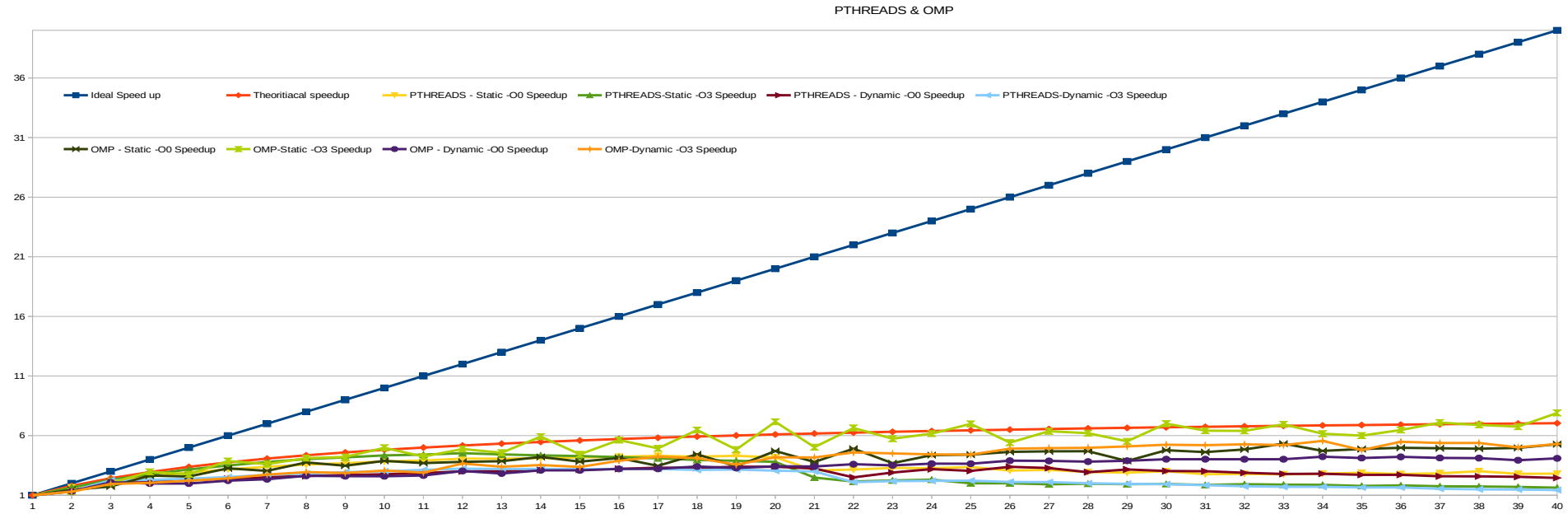
5) Γραφική Παράσταση (10%) και Σχολιασμός (~200 λέξεις) για τις λύσεις πιο πάνω. (20%)

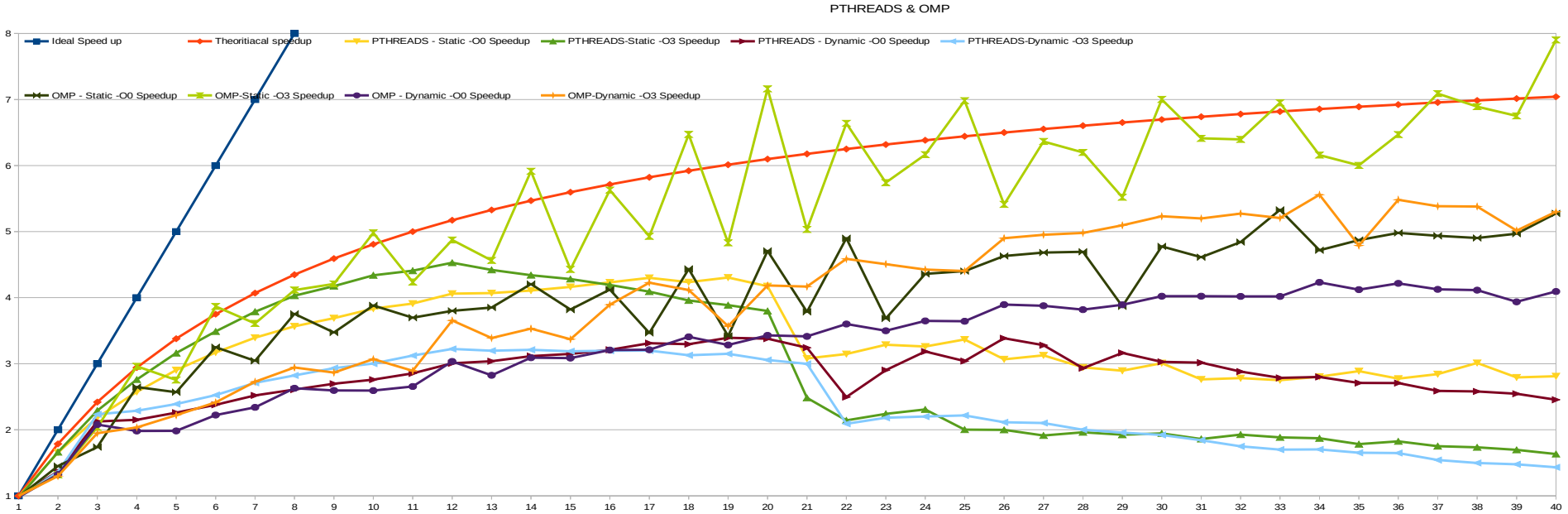




Από τις πιο πάνω μετρησεις μπορούμε να συμπεράνουμε ότι οι εκτελέσεις μας δεν είναι καθόλου κοντά στο ιδεατό speedup. Αυτό γιατί δεν είναι 100% του προγράμματος μας παραλληλισμένο. Από την άλλη παρατηρούμε πως εφαρμόζεται με σχετική ακρίβια ο νόμος του Αμνταλ, αφού τα πραγματικά μας speedup είναι κοντά στο θεωριτικό speedup που προτείνει ο νόμος. Στο συγκεκριμένο πείραμα παρατηρούμε πως το dynamic scheduling μας δίνει τα χειρότερα speedups σε σύγκριση με το static scheduling. Το στατικό scheduling φαίνεται να μας δίνει πιο μεγάλο speedup. Αυτό γίνεται επειδή στο συγκεκριμένο πείραμα εκμεταλευόμαστε την χωρική τοπικότητα του πινάκων positions και accelerations, σε αντίθεση με το dynamic schedule που φαίνεται να χάνει αυτό το πλεονέκτημα εκμετάλλευσης της τοπικής χωρικότητας. Όσο αφορά τις βελτιστοποιήσης μεταξύ O0 και O3, μπορούμε να πούμε πως οι εκτελέσεις με O3 μας δίνουν παρόμοιο speedup, όπου για λίγα threads το O3 να είναι δίνει καλύτερο speedup, ενώ για πολλά threads μας δίνει χειρότερο. Επίσης βλέπουμε πως πάνω απο 21 threads το speedup καταρρέει. Αυτό επειδή έχουμε μεγάλο overhead κατα την διαχείρηση και συγχρονισμό των νημάτων.

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





Ο νόμος του Amdahl εφαρμόστηκε για τον υπολογισμό της θεωρητικής επιτάχυνσης, η οποία προϋποθέτει οτι μόνο το 88% του προγράμματος μπορεί να παραλληλιστεί. Η υλοποίηση OpenMP ξεπερνά τα Pthread σε όλα τα σενάρια, με τον στατικό αλγόριθμο προγραμματισμού να παράγει καλύτερα αποτελέσματα από τον δυναμικό προγραμματισμό. Το OpenMP static O3 παρουσιάζει την καλύτερη επιτάχυνση, ξεπερνώντας κάποιες φορές τη θεωρητική επιτάχυνση. Ο στατικός αλγόριθμος προγραμματισμού αναθέτει ίση εργασία σε κάθε νήμα κατά το χρόνο μεταγλώττισης, μειώνοντας την επιβάρυνση του συγχρονισμού νημάτων και την εξισορρόπηση φορτίου, καθώς και εκμεταλεύεται την χωρική τοπικότητα των πινάκων positions και accelerations. Ο δυναμικός προγραμματισμός είναι αναποτελεσματικός για αυτό το πρόβλημα, με ομοιόμορφο και προβλέψιμο φόρτο εργασίας μεταξύ των νημάτων. Μετά από 21 νήματα, η επιτάχυνση του pthread καταρρέει, ενώ το OpenMP προσαρμόζεται δυναμικά στις δυνατότητες του συστήματος και στις μεταβλητές περιβάλλοντος. Το OpenMP είναι ένα μοντέλο προγραμματισμού που φαίνεται να μπορεί να διαχειριστεί καλύτερα τα πολλαπλά νήματα πέρα των 21. Το γράφημα παρέχει πληροφορίες για την απόδοση παράλληλης υλοποίησης, υπογραμμίζοντας τη σημασία του σχεδιασμού και της βελτιστοποίησης. Μια σημαντική διαφορά μεταξύ του OpenMP και των pthreads είναι το επίπεδο αφαίρεσης που παρέχουν. Συνεπώς το OpenMP είναι πιο ισχυρό εργαλείο από τα Pthreads για αυτό το σενάριο.