Παράλληλη Επεξεργασία

Εαρινό Εξάμηνο 2020-2021

Εργασία

"Παράλληλη Ολική Βελτιστοποίηση"

Τσαμπάς Στυλιανός 1039884
Μπούζας Γεώργιος 1051850
Παλουμπής Γεώργιος 1069366
Γιάνναρος Αθανάσιος 1067487

α. Εισαγωγή

Σκοπός της εργασίας αυτής, είναι η παραλληλοποίηση της υλοποίησης του κώδικα της Ολικής Βελτιστοποίησης όπως αυτός υλοποιείται από την NetLib. Ως αρχική υλοποίηση χρησιμοποιήθηκε το παρεχόμενο αρχείο `multistart_hooke_seq.c`. Οι ζητούμενες υλοποιήσεις υπάρχουν στα αρχεία `multistart_hooke_omp.c`, `multistart_hooke_omp_tasks.c`, `multistart_hooke_mpi.c` και `multistart_hooke_mpi_omp.c`.

Ο κώδικας και οι μετρήσεις υλοποιήθηκαν σε σύστημα με επεξεργαστή Intel(R) Core(TM) i5-4690K CPU @ 3.50GHz αρχιτεκτονικής Haswell με τέσσερεις επεξεργαστικούς πυρήνες, ενός νήματος εκτέλεσης έκαστος, και 6Mb cache. Το σύστημα αυτό χρησιμοποιεί 16GB DDR3 (4x4GB) μνήμης στα 1600 MT/s. Το λειτουργικό σύστημα είναι bare-metal Linux με πυρήνα στην έκδοση 5.12.9. Ο πυρήνας χρησιμοποιεί τον Multiple Queue Skiplist Scheduler (MuQSS) και είναι ρυθμισμένος για low latency, ως εκ τούτου δεν έχει το μέγιστο δυνατό throughput. Ο πυρήνας του Linux του συστήματος καθώς και τα προγράμματα της εργασίας είναι χτισμένα με τα `-march=native` και `-O3` switches του GCC.

Στην σειριακή υλοποίηση αντικαταστάθηκε η `srand(time(0))` με την `srand(1)` όπως προτεινόταν από την εκφώνηση για μπορούν να αναπαραχθούν τα αποτελέσματα μεταξύ των εκτελέσεων. Από την εκτέλεση της σειριακής υλοποίησης πάρθηκαν οι παρακάτω μετρήσεις για αυτό το σύστημα για τρεις διαδοχικές εκτελέσεις.

	Elapsed Time (seconds)	Function evaluations
	84.445	3123609265
	84.903	3123609265
	85.127	3123609265
MEAN:	84.83	

β. Τεκμηρίωση παράλληλων υλοποιήσεων

<u>Κοινά χαρακτηριστικά μεταξύ των υλοποιήσεων</u>

Σε όλες τις υλοποιήσεις αποφασίσαμε να παραλληλοποιήσουμε τον κύριο βρόγχο εύρεσης ελαχίστων, δηλαδή τον υπολογισμό των `hooke()` και `f()` για κάθε trial. Πιστεύουμε ότι δεδομένου του μικρού αριθμού στοιχείων (16) για κάθε trial, η παραλληλοποίηση εντός αυτών δεν θα ήταν η καλύτερη στρατηγική.

Η μεταβλητή `funevals` έγινε τοπική μεταβλητή της `main()` και χρησιμοποιήθηκε σαν indirect return στην κάθε συνάρτηση που καλεί την συνάρτηση Rosenbrock.

Για λόγους αναπαραγωγής των αποτελεσμάτων, λόγω της χρήσης της `drand()`, η αρχικοποίηση του πίνακα `startpt[]` γίνεται εκτός του παράλληλου τμήματος στην περίπτωση του OpenMP και μόνο στον Master στην περίπτωση του MPI. Για λόγους ισότητας, το allocation της απαιτούμενης μνήμης γίνεται εκτός της χρονομετρούμενης περιοχής του κώδικα, ενώ η αρχικοποίηση του γίνεται εντός. Στην περίπτωση του MPI και η αποστολή των χρησιμοποιούμενων τμημάτων του πίνακα γίνεται και αυτή εντός του χρονομετρούμενης περιοχής. Αυτό αυξάνει τις απαιτήσεις μνήμης των υλοποιήσεων κατά περίπου 250Mb ανά διεργασία η ποία στα πλαίσια των απαιτήσεων της εργασίας δεν ήταν αποτρεπτική. Ο χρόνος αρχικοποίησης του πίνακα κατά τις δοκιμές μας δεν ήταν μετρήσιμα σημαντικός.

Η παραπάνω αλλαγή έγινε στο OpenMP με τον παρακάτω κώδικα

```
srand48(1);
// allocate guess array for every trial
double **startpt = (double **)malloc(ntrials * sizeof(double *));
for (int i = 0; i < ntrials; i++)
  startpt[i] = (double *)malloc(MAXVARS * sizeof(double));
t0 = get_wtime();
/* initializing guess for rosenbrock test function, search space in [-4, 4) */
for (int j = 0; j < ntrials; j++)
  for (int i = 0; i < nvars; i++)
    startpt[j][i] = 4.0 * drand48() - 4.0;
```

Ενώ στην περίπτωση του MPI γίνεται μόνο στον Master (`rank = 0`)

```
srand48(1);
// allocate guess array for every trial
double **startpt = (double **)malloc(ntrials * sizeof(double *));
for (int i = 0; i < ntrials; i++)
  startpt[i] = (double *)malloc(MAXVARS * sizeof(double));
t0 = get_wtime();
/* initializing guess for rosenbrock test function, search space in [-4, 4) */
if (!rank)
  for (int j = 0; j < ntrials; j++)
    for (int i = 0; i < nvars; i++)
      startpt[j][i] = 4.0 * drand48() - 4.0;
```

Υλοποίηση με OpenMP

Στην περίπτωση του OpenMP έγινε παραλληλοποίηση των trials μέσω ενός omp parallel for όπως φαίνεται παρακάτω.

```
#pragma omp parallel for default(shared) private(endpt, jj, fx) reduction(+:funevals)
for (trial = 0; trial < ntrials; trial++) {
```

Οι μεταβλητές δηλώθηκαν by default ως shared, ενώ συγκεκριμένες μεταβλητές οι οποίες είναι έπρεπε να προστατευτούν στην κάθε επανάληψη δηλώθηκαν ως `private`. Μέσω της δήλωση της `reduction` γίνεται αυτόματη πρόσθεση της τοπικής μεταβλητής `funevals` στο τέλος κάθε thread.

Στην κάθε επανάληψη οι μεταβλητές `best_fx`, `best_trial`, `best_jj`, `best_pt` είναι `shared`, και έτσι πρέπει να προστατευτούν ανάμεσα στα διαφορετικά threads ώστε μόνο ένα να μπορεί να τις ανανεώσει κάθε φορά μόνο ένα. Αυτό γίνεται με την δήλωση του block ως `critical`. Ως micro-optimization το block δηλώθηκε μετά την σύγκριση για να μην εισάγεται συγχρονισμός αν αυτή αποτύχει.

```
  if (fx < best_fx) {
    #pragma omp critical
    {
    best_fx = fx;
    best_trial = trial;
    best_jj = jj;
    for (int i = 0; i < nvars; i++)
      best_pt[i] = endpt[i];
    }
  }
```

Υλοποίηση με OpenMP tasks

Όπως και παραπάνω η παραλληλοποίηση έγινε γύρω από τα trials με δήλωση των παρακάτω directives.

```
#pragma omp parallel
#pragma omp single nowait
for (trial = 0; trial < ntrials; trial++) {
```

Δηλώθηκε η περιοχή της `for` ως παραλλήλη καθώς και ότι μόνο ένα thread μπορεί κάθε φορά να εισέλθει και απενεργοποιήθηκε το implicit barrier.

Το σώμα της `for` δηλώθηκε ως task

```
  #pragma omp task shared(best_fx, best_trial, best_jj, best_pt, funevals)
  {
    unsigned long funnyvals = 0;
    jj = hooke(nvars, startpt[trial], endpt, rho, epsilon, itermax,
               &funnyvals);
    fx = f(endpt, nvars, &funnyvals);
    #pragma omp critical
    {
      if (fx < best_fx) {
        best_trial = trial;
        best_jj = jj;
        best_fx = fx;
        for (int i = 0; i < nvars; i++)
          best_pt[i] = endpt[i];
      }
      funevals += funnyvals;
    }
  }
```

Οι μεταβλητές `best_fx`, `best_trial`, `best_jj`, `best_pt` και `funevals` δηλώθηκαν ως `shared`, και η ανανέωση τους προστατεύτηκε μέσω του critial construct. Για την καταμέτρηση των επαναλήψεων δηλώθηκε τοπική μεταβλητή `funnyvals`.

Υλοποίηση με MPI

Όπως αναφέρθηκε παραπάνω, η αρχικοποίηση του πίνακα `startpt` γίνεται στο master, ο οποίος υλοποιεί και μέρος των υπολογισμών. Ως εκ τούτου, το master πρέπει να αποστείλει τα αντίστοιχα τμήματα του `startpt[]` στα slave. Αυτό γίνεται με την παρακάτω διαδικασία.

```
if (!rank) {
  for (int c = 1; c < procs; c++) {
    unsigned long c_start = c * step;
    unsigned long c_end = (c + 1) * step;
    if (c == procs - 1)
      c_end = ntrials;
    MPI_Send(startpt[c_start], (c_end - c_start) * MAXVARS, MPI_DOUBLE, c, 50,
             MPI_COMM_WORLD);
  }
} else {
  MPI_Status status;
  MPI_Recv(startpt[start], (end - start) * MAXVARS, MPI_DOUBLE, 0, 50,
           MPI_COMM_WORLD, &status);
}
```

Αρχικά στο master υπολογίζονται τα offsets του πίνακα για κάθε slave και αποστέλλονται, ενώ τα slave περιμένουν την παραλαβή των μηνυμάτων.

Μετά το τέλος των επαναλήψεων που αναλογούν σε κάθε process, τα slave αποστέλλουν στο master τα αποτελέσματα τους όπως φαίνεται παρακάτω

```
if (rank) {
  MPI_Send(&best_fx, 1, MPI_DOUBLE, 0, 60, MPI_COMM_WORLD);
  MPI_Send(&best_trial, 1, MPI_INT, 0, 61, MPI_COMM_WORLD);
  MPI_Send(&best_jj, 1, MPI_INT, 0, 62, MPI_COMM_WORLD);
  MPI_Send(&best_pt, MAXVARS, MPI_DOUBLE, 0, 63, MPI_COMM_WORLD);
  MPI_Send(&funevals, 1, MPI_UNSIGNED_LONG, 0, 64, MPI_COMM_WORLD);
```

Το master process αντίστοιχα περιμένει τη παραλαβή των αποτελεσμάτων από κάθε slave

```
} else {
  double best_fx_buf[procs-1];
  int best_trial_buf[procs-1];
  int best_jj_buf[procs-1];
  double best_pt_buf[procs-1][MAXVARS];
  unsigned long funevals_buf[procs-1];
  MPI_Status status;
  for (int c = 0; c < procs-1; c++) {
    MPI_Recv(&best_fx_buf[c], 1, MPI_DOUBLE, c+1, 60, MPI_COMM_WORLD, &status);
    MPI_Recv(&best_trial_buf[c], 1, MPI_INT, c+1, 61, MPI_COMM_WORLD, &status);
    MPI_Recv(&best_jj_buf[c], 1, MPI_INT, c+1, 62, MPI_COMM_WORLD, &status);
    MPI_Recv(&best_pt_buf[c], MAXVARS, MPI_DOUBLE, c+1, 63, MPI_COMM_WORLD,
             &status);
    MPI_Recv(&funevals_buf[c], 1, MPI_UNSIGNED_LONG, c+1, 64, MPI_COMM_WORLD,
             &status);
  }
  for (int c = 0; c < procs-1; c++) {
    if (best_fx_buf[c] < best_fx) {
      best_trial = best_trial_buf[c];
      best_jj = best_jj_buf[c];
      best_fx = best_fx_buf[c];
      for (int v = 0; v < MAXVARS; v++)
        best_pt[v] = best_pt_buf[c][v];
    }
    funevals += funevals_buf[c];
  }
}
```

Στο τέλος συγκρίνει τα αποτελέσματα από τα slave με τα δικά του για να αποφασίσει το καλύτερο.

Υλοποίηση με MPI και OpenMP

Η υλοποίηση του υβριδικού μοντέλου MPI και OpenMP είναι βασισμένη στα παραπάνω, οι διαδικασίες της είναι ίδιες με του MPI με την διαφορά της χρήσης της

```
MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
```

στην οποία χρησιμοποιήθηκε η οδηγία `MPI_THREAD_FUNNELED` γιατί θέλαμε επικοινωνία μόνο από το κύριο thread του κάθε process.

Για την παραλληλοποίηση του κυρίως βρόγχου `for` χρησιμοποιήσαμε την υλοποίηση όπως αυτή έγινε στο απλό OpenMP.

γ. Αποτελέσματα

Υλοποίηση με OpenMP

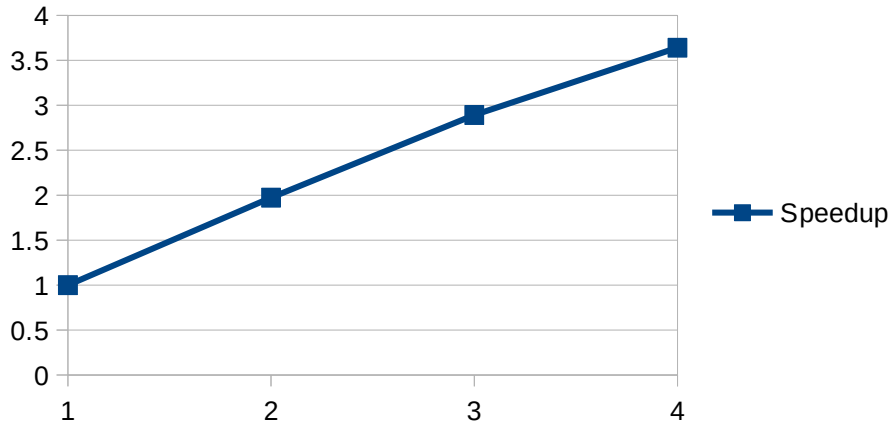
Στον παρακάτω πίνακα φαίνονται οι χρόνοι εκτέλεσης της συγκεκριμένης υλοποίησης για 1 έως 4 threads.

OMP_NUM_THREADS	Elapsed Time (seconds)	Function evaluations
1	84.929	3123609265
1	85.762	3123609265
1	85.585	3123609265
MEAN:	85.43	
OMP_NUM_THREADS	Elapsed Time (seconds)	Function evaluations
2	42.751	3123609265
2	43.015	3123609265
2	43.249	3123609265
MEAN:	43.005	
OMP_NUM_THREADS	Elapsed Time (seconds)	Function evaluations
3	29.372	3123609265
3	29.255	3123609265
3	29.376	3123609265
MEAN:	29.33	
OMP_NUM_THREADS	Elapsed Time (seconds)	Function evaluations
4	23.135	3123609265
4	23.500	3123609265
4	23.249	3123609265
MEAN:	23.294	

Και εδώ φαίνεται το speedup του μέσου χρόνου για κάθε διαφορετικό αριθμό threads.

Number of threads	Speedup
1	1
2	1.972
3	2.892
4	3.641

Από τους παραπάνω πίνακες φαίνεται για δύο και τρία threads έχουμε σχεδόν γραμμική βελτίωση σε σχέση με τον σειριακό κώδικα καθώς είναι κοντά στις βέλτιστες περιπτώσεις για αυτό τον αριθμό threads. Στην περίπτωση των τεσσάρων threads, η βελτίωση είναι αρκετά μικρότερη της βέλτιστης και πιστεύουμε ότι είναι λόγω του ότι εξυπηρετούνταν και άλλες διεργασίες ταυτόχρονα στο σύστημα.



Υλοποίηση με OpenMP tasks

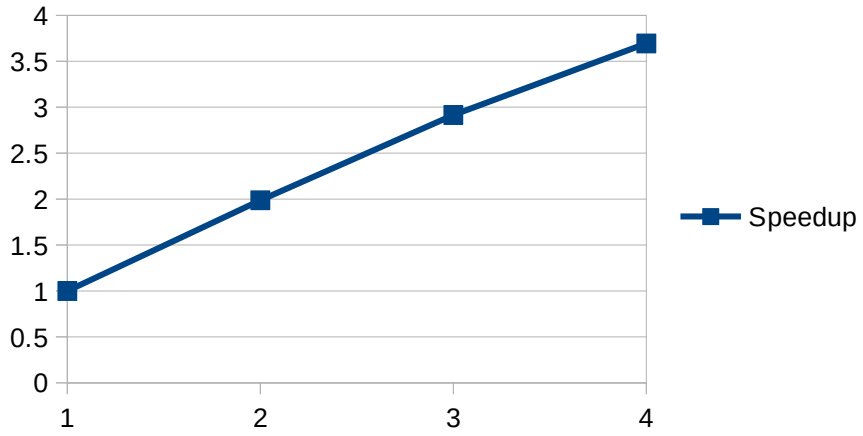
Στον παρακάτω πίνακα φαίνονται οι χρόνοι εκτέλεσης της συγκεκριμένης υλοποίησης για 1 έως 4 threads.

OMP_NUM_THREADS	Elapsed Time (seconds)	Function evaluations
1	84.966	3123609265
1	84.931	3123609265
1	84.925	3123609265
MEAN:	84.94	
OMP_NUM_THREADS	Elapsed Time (seconds)	Function evaluations
2	42.681	3123609265
2	42.720	3123609265
2	42.701	3123609265
MEAN:	42.700	
OMP_NUM_THREADS	Elapsed Time (seconds)	Function evaluations
3	29.111	3123609265
3	29.099	3123609265
3	29.096	3123609265
MEAN:	29.1	
OMP_NUM_THREADS	Elapsed Time (seconds)	Function evaluations
4	23.014	3123609265
4	22.941	3123609265
4	22.942	3123609265
MEAN:	22.97	

Και εδώ φαίνεται το speedup του μέσου χρόνου για κάθε διαφορετικό αριθμό threads.

Number of threads	Speedup
1	1
2	1.987
3	2.915
4	3.694

Από τους παραπάνω πίνακες φαίνεται για δύο και τρία threads έχουμε σχεδόν γραμμική βελτίωση σε σχέση με τον σειριακό κώδικα καθώς είναι κοντά στις βέλτιστες περιπτώσεις για αυτό τον αριθμό threads. Στην περίπτωση των τεσσάρων threads, η βελτίωση είναι αρκετά μικρότερη της βέλτιστης και πιστεύουμε ότι είναι λόγω του ότι εξυπηρετούνταν και άλλες διεργασίες ταυτόχρονα στο σύστημα.



Υλοποίηση με MPI

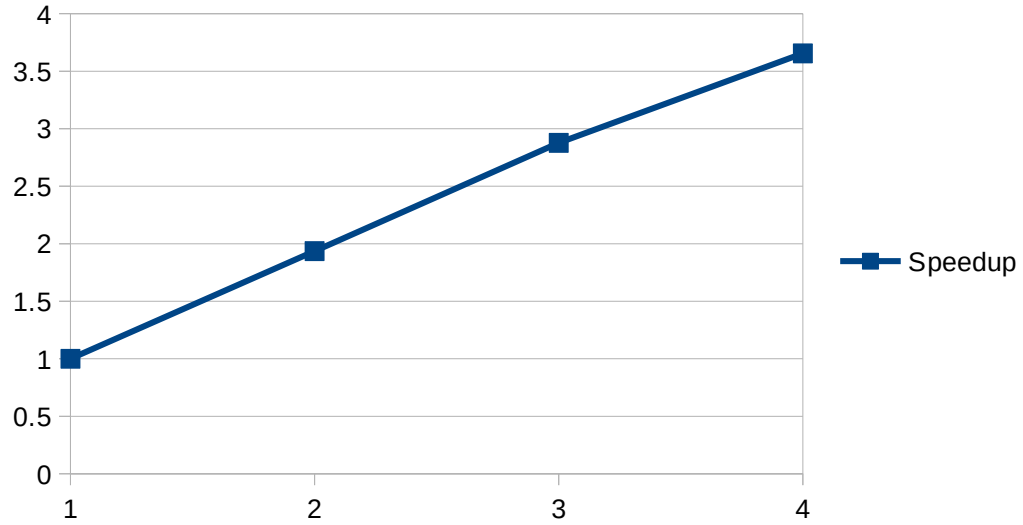
Στον παρακάτω πίνακα φαίνονται οι χρόνοι εκτέλεσης της συγκεκριμένης υλοποίησης για 1 έως 4 processes.

-n	Elapsed Time (seconds)	Function evaluations
1	85.563	3123609265
1	86.684	3123609265
1	86.474	3123609265
MEAN:	86.24	
-n	Elapsed Time (seconds)	Function evaluations
2	43.570	3118306570
2	44.113	3118306570
2	43.825	3118306570
MEAN:	43.836	
-n	Elapsed Time (seconds)	Function evaluations
3	29.479	3116481084
3	29.463	3116481084
3	29.521	3116481084
MEAN:	29.49	
-n	Elapsed Time (seconds)	Function evaluations
4	23.138	3117065950
4	23.310	3117065950
4	23.186	3117065950
MEAN:	23.21	

Και εδώ φαίνεται το speedup του μέσου χρόνου για κάθε διαφορετικό αριθμό processes.

Number of processes	Speedup
1	1
2	1.935
3	2.877
4	3.655

Από τους παραπάνω πίνακες φαίνεται ότι έχουμε πτωτική βελτίωση όσο ανεβαίνει ο αριθμός των processes με ραγδαία πτώση όταν έχουμε τόσα processes όσα και cores.



Υλοποίηση με MPI και OpenMP

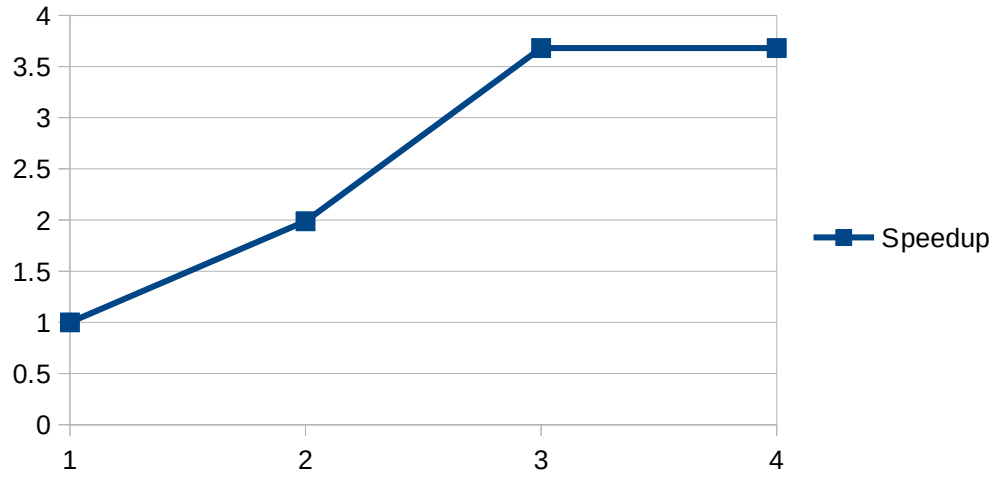
Στον παρακάτω πίνακα φαίνονται οι χρόνοι εκτέλεσης της συγκεκριμένης υλοποίησης για 1 έως 4 processes.

-n	Elapsed Time (seconds)	Function evaluations
1	85.982	3123609265
2	85.923	3123609265
3	85.053	3123609265
MEAN:	85.65	
-n	Elapsed Time (seconds)	Function evaluations
1	42.687	3118306570
2	42.683	3118306570
3	42.667	3118306570
MEAN:	42.68	
-n	Elapsed Time (seconds)	Function evaluations
1	22.983	3116481084
2	23.084	3116481084
3	23.088	3116481084
MEAN:	23.05	
-n	Elapsed Time (seconds)	Function evaluations
1	23.101	3117065950
2	23.014	3117065950
3	23.039	3117065950
MEAN:	23.05	

Και εδώ φαίνεται το speedup του μέσου χρόνου για κάθε διαφορετικό αριθμό processes.

Number of processes	Speedup
1	1
2	1.988
3	3.680
4	3.680

Από τους παραπάνω πίνακες φαίνεται ότι έχουμε σχεδόν γραμμική βελτίωση από ένα σε δύο processes και ραγδαία βελτίωση από 2 σε 3 ή τέσσερα. Η βελτίωση αυτή είναι πλασματική και συμβαίνει γιατί παρατηρήσαμε το παρακάτω φαινόμενο για το οποίο δεν μπορέσαμε να το εξηγήσουμε εμπεριστατωμένα ή να βρούμε scalable λύση. Για 1 ή 2 processes, δεν δημιουργούνταν threads από το OpenMP, για 3 και παραπάνω processes, υπήρχαν "-n x" processes και x*(num_cores-1) threads. Δηλαδή για 3 processes είχαμε 9 OpenMP threads και για 4 processes 12 OpenMP threads. Η βελτίωση σε αυτή την περίπτωση είναι οριακά μικρότερη από την υλοποίηση με OpenMP tasks.



γ. Συμπεράσματα

Το πρόβλημα σαν θέμα της εργασίας είναι ένα ευνοϊκό πρόβλημα για παραλληλοποίηση καθώς κάθε επανάληψη είναι ανεξάρτητη από την προηγούμενη ως διαδικασία. Η μόνη εξάρτηση είναι η εύρεση του ολικού ελαχίστου που απαιτεί σύγκριση των αποτελεσμάτων. Δεδομένου αρκετού αποθηκευτικού χώρου, η σύγκριση θα μπορούσε να αποφευχθεί τελείως και να γίνει μέσω user-defined reduction για το OpenMP μετά το πέρας των υπολογισμών. Αυτό θα αφαιρούσε το critical κομμάτι και θα μπορούσε να βελτιώσει παραπάνω την απόδοση. Στην περίπτωση του MPI, καθώς δεν χρειάζεται τέτοια προστασία, η αύξηση των processes θα συνέχιζε να οδηγεί σε βελτίωση.

Στις υλοποιήσεις μας, τα OpenMP tasks φαίνονται οριακά καλύτερα από τις άλλες μεθόδους.