Πολυτεχνική Σχολή
Τμήμα Μηχανικών Η/Υ & Πληροφορικής

Δ.Π.Μ.Σ. «ΥΠΟΛΟΓΙΣΤΙΚΗ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΑΠΟΦΑΣΕΩΝ – Υ.Δ.Α.»

Υπολογιστική Υψηλών Επιδόσεων Επιστήμης Δεδομένων

ΑΝΑΦΟΡΑ ΕΡΓΑΣΤΗΡΙΑΚΗΣ ΑΣΚΗΣΗΣ

Παράλληλη Υλοποίηση του αλγορίθμου KNN

Σκόνδρας Γεώργιος
Α.Μ. 1020408
Πάτρα, 2023

Περιεχόμενα

Μηχάνημα Υλοποίησης

Για την παρούσα άσκηση χρησιμοποιήθηκε ένας φορητός υπολογιστής Huawei Matebook d14 με 8GB DDR4 μνήμης RAM και επεξεργαστή Ryzen 5 3500U με 4 cores και 8 threads.

Σταθμισμένος μέσος όρος αντίστροφης απόστασης

Η δοθείσα υλοποίηση βρίσκει τους πλησιέστερους *k* γείτονες και στη συνέχεια προσεγγίζει την τιμή της συνάρτησης λαμβάνοντας τις μέσες τιμές της συνάρτησης των γειτονικών σημείων. Παρακάτω βλέπουμε πως γίνεται ο υπολογισμός της μέσης τιμής στη συνάρτηση predict_value() στο αρχείο func.c

```
double predict_value(int dim, int knn, double *xdata, double *ydata, double *point, double *dist)
{
   int i;
   double sum_v = 0.0;
   double sum_d = 0.0;
   // TODO: implement inverse distance weight
   for (i = 0; i < knn; i++) {
       sum_v += ydata[i];
   }

   return sum_v/knn;
}
```

Η εκτέλεση του κώδικα αρχικού κώδικα έχει τα παρακάτω αποτελέσματα:

```
gskondras@Peregrine-Falcon:~/Desktop/HPC-course/C_Project$ ./myknn x.txt q.txt
Results for 1024 query points
APE = 4.86 %
MSE = 1.803033
R2 = 1 - (MSE/Var) = 0.920909
Total time = 10536.234140 ms
Time for 1st query = 10.815859 ms
Time for 2..N queries = 10525.418282 ms
Average time/query = 10.288776 ms
```

Για να εφαρμόσουμε την τεχνική inverse distance weighted average στον knn, κατά το prediction υπολογίζουμε το σταθμισμένο μέσο όρο με βάση την αντίστροφη απόσταση από την τιμή του αντίστοιχου query point. Ουσιαστικά πολλαπλασιάζουμε κάθε τιμή με την αντίστροφη απόστασή της από το αντίστοιχο σημείο και στη συνέχεια στο συνολικό άθροισμα διαιρούμε με το άθροισμα των αντίστροφων αποστάσεων, όπως βλέπουμε παρακάτω.

```
double predict_value(int dim, int knn, double *xdata, double *ydata, double *point, double
*dist)
{
   int i;
   double sum_v = 0.0;
   double sum_d = 0.0;
   for (i = 0; i < knn; i++) {
       sum_v += ydata[i]/dist[i];
       sum_d += 1/dist[i];
   }

   return sum_v/sum_d;
}
```

Η εκτέλεση του κώδικα με τη χρήση του inverse distance weighted average μας παρέχει ανάλογες προβλέψεις σε σχέση με τον αρχικό κώδικα, όπως φαίνεται παρακάτω.cl

```
gskondras@Peregrine-Falcon:~/Desktop/HPC-course/C_Project$ ./myknn x.txt q.txt
Results for 1024 query points
APE = 4.82 %
MSE = 1.782709
R2 = 1 - (MSE/Var) = 0.921801
Total time = 10670.474291 ms
Time for 1st query = 11.015177 ms
Time for 2..N queries = 10659.459114 ms
Average time/query = 10.419804 ms
```

Αναδιοργάνωση κώδικα

Στην δοθείσα υλοποίηση στη συνάρτηση main() το κάθε query element διαβάζεται από το αρχείο, γίνεται το prediction, διαβάζεται το επόμενο query element κ.ο.κ και έτσι οι χρονομετρήσεις εμπεριέχουν και I/O overhead. Θα αναδιοργανωσω τον κώδικα ώστε το σύνολο δεδομένων να φορτωθεί εκ των προτέρων ώστε οι χρονομετρήσεις να μην περιλαμβάνουν το I/O overhead. Δημιουργώ το αρχείο my_knn_parallel.c που έχει τον αναδιοργανομένο κώδικα. Έτσι αντί να διαβάζω κάθε φορά τις τιμές του στοιχείου από το διάνυσμα

```
   double x[PROBDIM];
```

Φτιάχνω έναν πίνακα `xmem_test` με συνεχόμενες θέσεις μνήμης και φορτώνω όλα τα δεδομένα εκεί, όπως φαίνεται παρακάτω:

```
double *xmem_test = (double *)malloc(QUERYELEMS*PROBDIM*sizeof(double));
   xtest = (double **)malloc(QUERYELEMS*sizeof(double *));

   for (int i = 0; i < QUERYELEMS; i++) xtest[i] = xmem_test + i*PROBDIM; //&mem[i*PROBDIM];

   for (int i=0;i<QUERYELEMS;i++) {
           for (int k = 0; k < PROBDIM; k++)
               xtest[i][k] = read_nextnum(fpin);
   #if defined(SURROGATES)
               ytest[i] = read_nextnum(fpin);
   #else
               ytest[i] = 0;
   #endif
   }
   fclose(fpin);
```

Profiling

Για να δούμε ποιο τμήμα του κώδικα είναι αρκετά απαιτητικό υπολογιστικά κάνουμε χρήση του gprof κάνοντας αρχικά compile τον σειριακό κώδικα χωρίς optimizations. Παρακάτω βλέπουμε ότι ότι στον περισσότερο χρόνο το πρόγραμμα εκτελεί την συνάρτηση compute_dist().

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 68.17     22.74    22.74                             compute_dist
 19.51     29.25     6.51                             compute_knn_brute_force
 11.51     33.09     3.84                             _init
  0.39     33.22     0.13                             main
  0.27     33.31     0.09                             read_nextnum
  0.12     33.35     0.04                             compute_max_pos
  0.03     33.36     0.01                             find_knn_value
```

Ωστόσο το for loop που βρίσκεται στην compute_list() έχει έχει μόνο PROBDIM επαναλήψεις, στην προκειμένη μόνο 16, οπότε η παραλληλοποίηση του δε θα βοηθήσει. Επίσης η παραλληλοποίηση του for loop στην compute_knn_brute_force() που καλεί την compute_dist() όπως φαίνεται παρακάτω πάλι δεν πέτυχε σημαντική επιτάχυνση.

```
#pragma omp paraller for private(i,new_d) firstprivate(max_d)
   for (i = 0; i < npat; i++) {
       new_d = compute_dist(q, xdata[i], lpat);    // euclidean
       if (new_d < max_d) {    // add point to the  list of knns, replace element max_i
           nn_x[max_i] = i;
           nn_d[max_i] = new_d;
           max_d = compute_max_pos(nn_d, knn, &max_i);
       }
   }
```

Συνεπώς θα παραλληλοποιήσουμε το βασικό for loop της main στο οποίο διαβάζονται τα στοιχεία του query dataset και γίνονται τα prediction.

Παραλληλοποίηση με OpenMP

Για την παραλληλοποίηση του loop που αναφέραμε, στο αρχείο myknn_parallel.c, εισάγουμε οδηγίες OpenMP, όπως φαίνεται παρακάτω:

```
t0 = omp_get_wtime();
   #pragma omp parallel for private(t0,t1,err) reduction(+:t_sum,sse,err_sum)
   for (int i=0;i<QUERYELEMS;i++) {
       double yp = find_knn_value(xtest[i], PROBDIM, NNBS);
       sse += (ytest[i]-yp)*(ytest[i]-yp);
       err = 100.0*fabs((yp-ytest[i])/ytest[i]);
       err_sum += err;

   }
   t1 = omp_get_wtime();
   t_sum = (t1-t0);
```

Με τη χρήση του #pragma omp parallel for ανοίγουμε μια παράλληλη περιοχή και οι επαναλήψεις του for μοιράζονται στα threads με την default μεθοδο schedule(static). Με το clause private(t0,t1,err) διασφαλίζουμε ότι θε θα έχουμε race conditions και με τη χρήση του reduction(+:t_sum,sse,err_sum) υπολογίζουμε αποδοτικά το συνολικό χρόνο, το τετραγωνικό και σχετικό σφάλμα. Όσον αφορά το yp εφόσον ορίζεται μέσα στο for δε χρειάζεται να μπει στο private.

Παρακάτω βλέπουμε την εκτέλεση του παράλληλου κώδικα με 2 threads.

```
gskondras@Peregrine-Falcon:~/Desktop/HPC-course/C_Project$ export OMP_NUM_THREADS=2
gskondras@Peregrine-Falcon:~/Desktop/HPC-course/C_Project$ ./myknn_parallel x.txt q.txt
Results for 1024 query points
APE = 4.82 %
MSE = 1.782709
R2 = 1 - (MSE/Var) = 0.921801
Total time = 6154.527601 ms
Average time/query = 6.010281 ms
```

Παραλληλοποίηση με OpenMP(tasks)

Για την παραλληλοποίηση με χρήση tasks δημιουργώ το αρχείο myknn_tasks.c και παραμετροποιώ το ίδιο κομμάτι κώδικα που αλλαξαμε στο προηγούμενο ερώτημα με οδηγίες OpenMP ως εξής:

```
t0 = omp_get_wtime();
   #pragma omp parallel
   {
       #pragma omp single nowait
       {
           for (int i=0;i<QUERYELEMS;i++) {
               #pragma omp task firstprivate(i) private(err)
               {
                   double yp = find_knn_value(xtest[i], PROBDIM, NNBS);
                   err = 100.0*fabs((yp-ytest[i])/ytest[i]);
                   #pragma omp atomic
                   sse += (ytest[i]-yp)*(ytest[i]-yp);

                   #pragma omp atomic
                   err_sum += err;

               }
           }
       }
   }
   t1 = omp_get_wtime();
   t_sum = (t1-t0);
```

Με το #pragma omp parallel ανοίγω μια παράλληλη περιοχή και με το #pragma omp single εξασφαλίζω ότι ένα thread θα δημιουργήσει τα tasks με την εντολή #pragma omp task. Δίνω το clause firstprivate(i) για να έχει το κάθε task το δικό του μετρητή με την τιμή που είχε αμέσως πριν και private(err) ώστε το κάθε task να υπολογίζει το δικό του σφάλμα. Με τις οδηγίες #pragma omp atomic εξασφαλίζω ότι δε θα υπάρχουν race condition στον υπολογισμό του συνολικού τετραγωνικού και σχετικού σφάλματος.
Παρακάτω βλέπουμε την εκτέλεση του παράλληλου κώδικα με tasks με 2 threads.

```
gskondras@Peregrine-Falcon:~/Desktop/HPC-course/C_Project$ ./myknn_tasks x.txt q.txt
Results for 1024 query points
APE = 4.82 %
MSE = 1.782709
R2 = 1 - (MSE/Var) = 0.921801
Total time = 6651.263463 ms
Average time/query = 6.501724 ms
```

Παραλληλοποίηση με MPI

Για την παραλληλοποίηση με χρήση mpi, δημιουργώ το αρχείο myknn_mpi.c. Κάνω τις κατάλληλες αρχικοποιήσεις στο MPI:

```
MPI_Init(&argc, &argv);
   int rank;
   int size;
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&size);
```

Παραμετροποιώ και πάλι το ίδιο κομμάτι κώδικα όπως φαίνεται παρακάτω:

```
   int batch = QUERYELEMS/size;
   int start = rank*batch;
   int end = start+batch;

   double t1 = MPI_Wtime();
   for (int i=start;i<end;i++) {
       double yp = find_knn_value(xtest[i], PROBDIM, NNBS);
       local_sse += (ytest[i]-yp)*(ytest[i]-yp);
       err = 100.0*fabs((yp-ytest[i])/ytest[i]);
       local_err_sum += err;

   }

   double t2 = MPI_Wtime();
   double t_local = t2-t1;

   MPI_Reduce(&local_sse, &global_sse, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
   MPI_Reduce(&local_err_sum, &global_err_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Η μεταβλητή batch περιέχει τον αριθμό των queries που θα εκτελέσει ο κάθε node από το στοιχείο start μέχρι το end. Ο αριθμός batch προκύπτει από τη διαίρεση του πλήθους των query elements με το πλήθος των nodes(χρησιμοποιούμε αριθμούς ώστε η διαίρεση να μην έχει υπόλοιπο). Τώρα μετρώ το χρόνο με τη χρήση της MPI_Wtime(). Χρησιμοποιώ την MPI_Reduce() με operation MPI_SUM ώστε να συγκεντρώσω τα συνολικά σφάλματα στους υπολογισμούς στο root node. Στο τέλος μόνο το root node τυπώνει τα αποτελέσματα. Παρακάτω βλέπουμε την εκτέλεση του παράλληλου κώδικα με MPI με 2 threads.

```
gskondras@Peregrine-Falcon:~/Desktop/HPC-course/C_Project$ mpiexec -n 2 myknn_mpi x.txt q.txt
rank 0:
Results for 1024 query points
APE = 4.82 %
MSE = 1.782709
R2 = 1 - (MSE/Var) = 0.921801
Total time = 6037.953571 ms
Average time/query = 5.896439 ms
```

Μετρήσεις απόδοσης

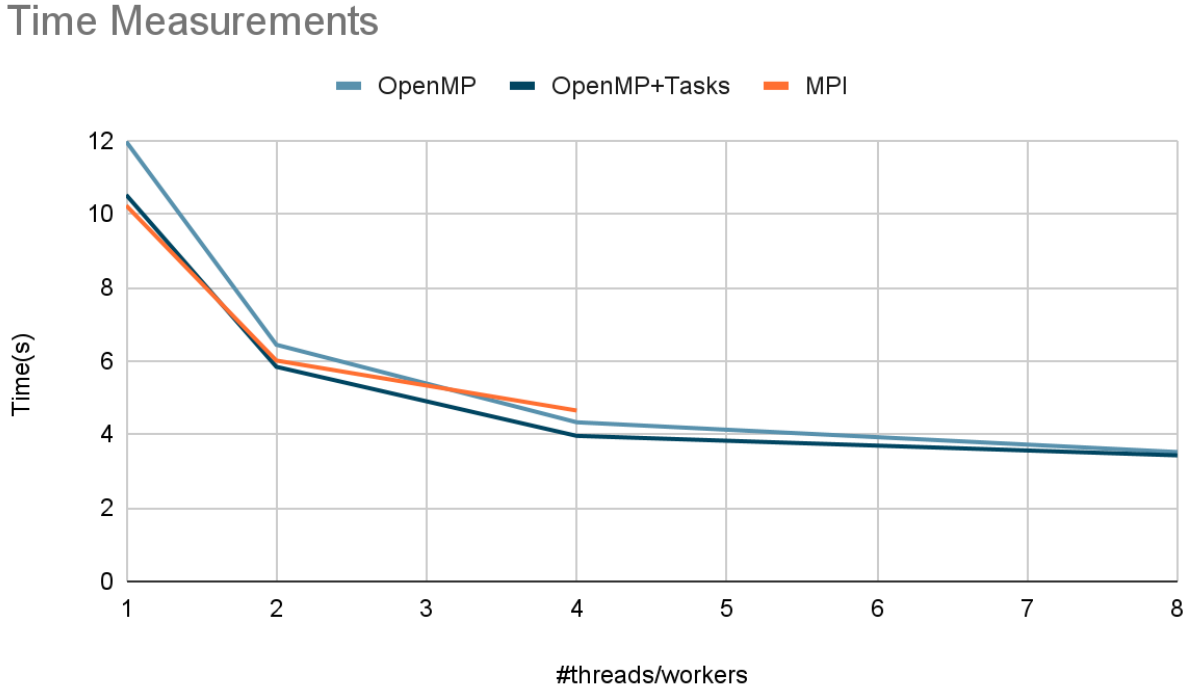
Για τις μετρήσεις της απόδοσης έγιναν πειράματα για τις 3 μεθόδους παραλληλοποίησης που χρησιμοποιήθηκαν(OpenMP, OpenMP+Tasks, MPI) με τη χρήση 1,2,4 και 8 workers. Η περίπτωση του MPI με τη χρήση 8 nodes δεν γινόταν να εκτελεστεί καθώς το σύστημα έχει ακριβώς 8 threads και ο mpicc μου έδινε το μήνυμα *“There are not enough slots available in the system to satisfy the 8 slots that were requested by the application”.* Τα πειράματα επαναλήφθηκαν 5 φορές και η τελική τιμή του χρόνου είναι ο μέσος όρος των χρόνων που προέκυψαν. Αρχικά έγιναν πειράματα με τις αρχικές παραμέτρους που δίνονται στο Makefile:

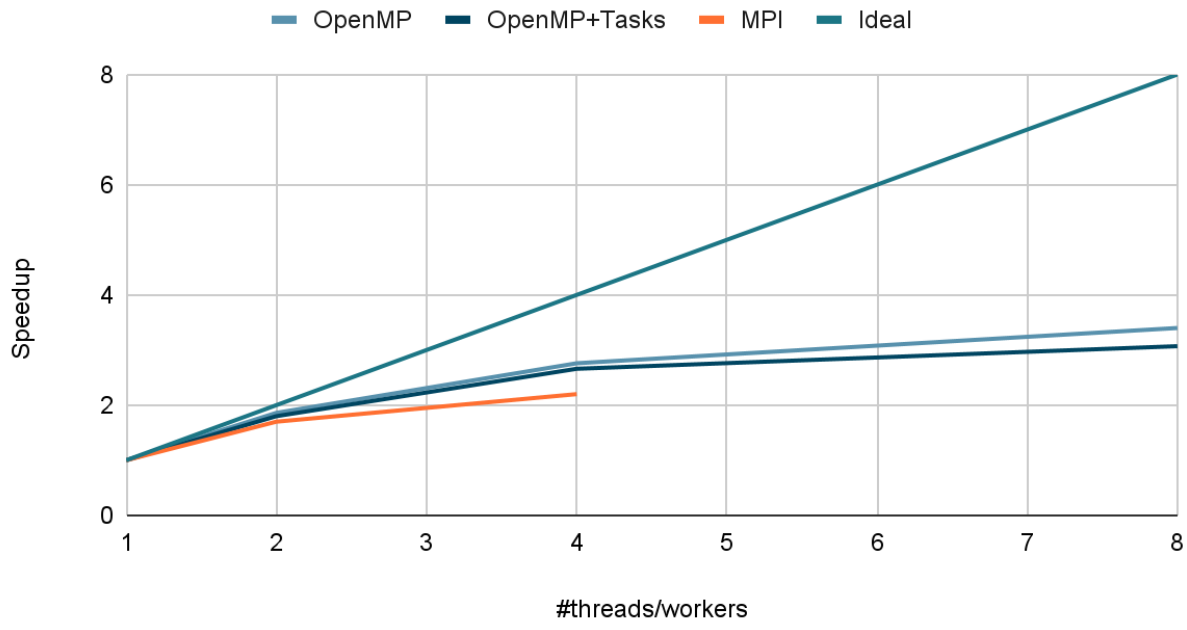
```
DIM ?= 16
KNN ?= 32
TRA ?= 1048576
QUE ?= 1024
```

Οι χρόνοι που μετρήθηκαν είναι οι εξής:

#threads/nodes	OpenMP	OpenMP+Task	MPI
1	11.97s	10.52s	10.23
2	6.44s	5.84s	6.01
4	4.33s	3.96s	4.65
8	3.52s	3.43s	-

Παρακάτω φαίνεται η γραφική αναπαράσταση των χρόνων εκτέλεσης και του speedup[Speedup(p) = T(1)/T(p)]





Παρατηρούμε καλή επιτάχυνση για 2 και 4 threads, ενώ για 8 threads η επιτάχυνση είναι χαμηλότερη. Πέραν της βελτιστοποιησης του κώδικα, ένας από τους λόγους που υπάρχει μικρή επιτάχυνση για μεγάλο αριθμό threads ίσως είναι και το γεγονός ότι τα πειράματα γίνονται σε

προσωπικό φορητό υπολογιστή με συνολικά 8 threads και κάποια από τα threads τρέχουν και διεργασίες του λειτουργικού παράλληλα. Επίσης κατά τη διάρκεια πειραμάτων με αρκετά threads η θερμοκρασία του μηχανήματος αυξάνεται σημαντικά ενδεχομένως οδηγώντας σε χαμηλότερη απόδοση.