ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

ΑΝΑΦΟΡΑ 5ης ΑΣΚΗΣΗΣ

Στοιχεία Ομάδας

- Αναγνωριστικό: parlab05
- Μέλος 1ο: Πέππας Μιχαήλ – Αθανάσιος, Α.Μ: 03121026
- Μέλος 2ο: Σαουνάτσος Ανδρέας, Α.Μ: 03121197
- Ημερομηνία Παράδοσης Αναφοράς: 19.01.2026

▪ Εισαγωγή

Στην παρούσα άσκηση μελετάμε την παραλληλοποίηση και τη βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κατανεμημένης μνήμης, αξιοποιώντας το πρότυπο MPI. Στόχος μας είναι να σχεδιάσουμε και να υλοποιήσουμε αποδοτικές κατανεμημένες εκδόσεις δύο προβλημάτων (K-means, 2D heat-transfer), να αξιολογήσουμε την κλιμακωσιμότητά τους σε πολλαπλές διεργασίες και να ερμηνεύσουμε τα αποτελέσματα με βάση τη θεωρία επικοινωνίας/συγχρονισμού σε συστήματα κατανεμημένης μνήμης, του μαθήματος.

Το πρώτο μέρος αφορά τον αλγόριθμο K-means. Εξετάζουμε πώς διαμοιράζονται τα δεδομένα (points) στις διεργασίες, πώς υπολογίζονται τοπικά οι συνεισφορές για τα νέα centroids και πώς συνδυάζονται μέσω συλλογικών επικοινωνιών (reduce/allreduce) ώστε να ενημερώνονται τα κέντρα ανά επανάληψη. Η ανάλυση εστιάζει τόσο στη λειτουργική ορθότητα όσο και στη συμπεριφορά χρόνου εκτέλεσης όταν αυξάνει ο αριθμός διεργασιών.

Το δεύτερο μέρος αφορά τη διάδοση θερμότητας (2D heat transfer). Σχεδιάζουμε κατανομή του πλέγματος (grid decomposition) και την απαιτούμενη ανταλλαγή οριακών τιμών (halo exchange) μεταξύ γειτονικών διεργασιών, καθώς και μηχανισμό ελέγχου σύγκλισης που βασίζεται σε global reductions. Παρουσιάζουμε μετρήσεις τόσο με ενεργό έλεγχο σύγκλισης όσο και με απενεργοποιημένη σύγκλιση για σταθερό αριθμό επαναλήψεων, ώστε να αξιολογηθεί καθαρά η κλιμάκωση και να απομονωθεί το κόστος επικοινωνίας.

Στην αναφορά καταγράφουμε το πώς υλοποιήθηκαν τα παραπάνω (βασικές σχεδιαστικές επιλογές, κλήσεις MPI, συγχρονισμός), και παρουσιάζουμε πειραματικά αποτελέσματα με διαγράμματα χρόνου και speedup για διαφορετικό αριθμό διεργασιών. Ιδιαίτερη έμφαση δίνεται στη σωστή μεθοδολογία χρονομέτρησης (διάκριση συνολικού χρόνου, χρόνου υπολογισμού και χρόνου επικοινωνίας), καθώς και στην ερμηνεία των τάσεων κλιμάκωσης με βάση έννοιες όπως συλλογικές επικοινωνίες, τοπολογίες/γειτονική επικοινωνία και κόστος συγχρονισμού.

Σημείωση: Έχουμε λάβει υπόψιν μας όλες τις υποδείξεις – διευκρινίσεις της άσκησης. Έτσι, οι χρόνοι που μετρήθηκαν και τα διαγράμματα συνιστούν τον μέσο όρο από τρία runs, όπως ζητούνταν (πιο σημαντική διευκρίνιση).

▪ Ενότητα 4.1 – Αλγόριθμος K-means με MPI

Α. Εισαγωγή

Στο πρώτο μέρος της άσκησης υλοποιούμε μια κατανεμημένη έκδοση του αλγορίθμου K-means, αξιοποιώντας το πρότυπο του MPI. Ο K-means είναι ένας επαναληπτικός αλγόριθμος ομαδοποίησης που:

1. αναθέτει κάθε σημείο (object) στο πλησιέστερο κέντρο συστάδας (centroid)
2. επανυπολογίζει τα centroids ως μέσο όρο των σημείων που τους ανήκουν
3. επαναλαμβάνει μέχρι σύγκλιση ή μέχρι μέγιστο αριθμό επαναλήψεων.

Στόχος μας είναι να παραλληλοποιήσουμε τη ροή σε αρχιτεκτονική και προγραμματιστικό μοντέλο κατανεμημένης μνήμης, ώστε κάθε διεργασία να επεξεργάζεται μόνο ένα τμήμα των δεδομένων και να συνεργάζεται με τις υπόλοιπες μέσω συλλογικών επικοινωνιών (collectives) για τον υπολογισμό των νέων centroids και του κριτηρίου σύγκλισης. Στην αναφορά παρουσιάζουμε:

- τον τρόπο κατανομής των δεδομένων (data decomposition) και την επιλογή των εντολών MPI,
- το πώς συνδυάζονται οι τοπικές συνεισφορές στις global (κοινές) μεταβλητές (global reductions),
- και (στη συνέχεια) τη μελέτη της κλιμάκωσης του προγράμματος, με μετρήσεις και διαγράμματα χρόνου και speedup, για διαφορετικό αριθμό διεργασιών.

Β. Υλοποίηση

Η υλοποίηση του αλγορίθμου βασίζεται σε σαφή διάσπαση του προβλήματος ως προς τα δεδομένα (objects): κάθε rank κρατά μόνο το δικό του υποσύνολο σημείων, ενώ τα centroids παραμένουν κοινά (ίδια τιμή) σε όλα τα ranks, σε κάθε επανάληψη. Ειδικότερα, επισημαίνουμε τα εξής σημεία στην υλοποίησή μας:

1) Παραγωγή/Κατανομή δεδομένων (dataset_generation στο file_io.c)

- Ο συνολικός αριθμός σημείων υπολογίζεται από το dataset_size (MB), το numCoords και το sizeof(double).
- Η κατανομή γίνεται ισότιμα (block distribution) με χρήση της μεταβλητής remainder: κάθε rank παίρνει είτε ⌊N/P⌋ είτε ⌊N/P⌋+1 αντικείμενα, ώστε να μοιράζεται ομοιόμορφα ο φόρτος όταν το N δεν διαιρείται ακριβώς με P. Αυτό αποφεύγει το έντονο load imbalance, ειδικά σε πολλά ranks.
- Το rank 0 δημιουργεί ντετερμινιστικά το dataset (rand_r με seed = i για κάθε object), ώστε να μπορούμε να αναπαράγουμε τα δεδομένα και τις μετρήσεις.
- Η κατανομή των δεδομένων γίνεται με MPI_Scatterv από το rank 0, χρησιμοποιώντας sendcounts/displs (objects × coords). Η χρήση Scatterv (αντί Scatter) είναι κρίσιμη, επειδή τα ranks μπορεί να λαμβάνουν διαφορετικό πλήθος αντικειμένων λόγω του remainder.

2) Αρχικοποίηση κέντρων (main.c)

- Τα αρχικά centroids επιλέγονται από τα πρώτα σημεία numClusters. Η επιλογή γίνεται μόνο από το rank 0 (για να υπάρχει μοναδική πηγή αρχικοποίησης) και στη συνέχεια διαχέεται σε όλα τα ranks με MPI_Bcast.
- Με αυτόν τον τρόπο όλα τα ranks ξεκινούν από τα ίδια centroids, άρα η εξέλιξη του αλγορίθμου παραμένει συνεπής.

3) Κύριος επαναληπτικός βρόχος K-means (kmeans.c)

Σε κάθε επανάληψη κάθε rank εκτελεί καθαρά τοπικό υπολογισμό πάνω στα δικά του objects:

- Ανάθεση (assignment): για κάθε local object, βρίσκουμε το πλησιέστερο centroid (εύρεση ελάχιστης τετραγωνικής Ευκλείδειας απόστασης) και ενημερώνουμε το membership[i].
- Τοπικές συσσωρεύσεις για update centroids:
    1. rank_newClusterSize[c]: πόσα τοπικά σημεία ανατέθηκαν στο cluster c,
    2. rank_newClusters[c, j]: άθροισμα των συντεταγμένων (sums) των τοπικών σημείων του cluster c.

Κατόπιν, συνδυάζουμε τα τοπικά αποτελέσματα στις global μεταβλητές, με reductions:

- MPI_Allreduce για rank_newClusters → newClusters (SUM) και rank_newClusterSize → newClusterSize (SUM). Η κύρια επιλογή εδώ ήταν η χρήση του Allreduce (και όχι Reduce) ώστε:
    1. να αποφεύγεται το bottleneck στο rank 0,
    2. και να έχουν όλα τα ranks άμεσα τα αθροίσματα/δεδομένα ώστε να ενημερώσουν τα centroids τοπικά, χωρίς επιπλέον επικοινωνία.
- Ενημέρωση centroids: για κάθε cluster c, αν newClusterSize[c] > 0, θέτουμε clusters[c, j] = newClusters[c, j] / newClusterSize[c].

4) Κριτήριο σύγκλισης

- Κάθε rank υπολογίζει rank_delta = πλήθος τοπικών αντικειμένων που άλλαξαν cluster σε αυτή την επανάληψη.
- Έπειτα κάνουμε MPI_Allreduce(rank_delta → delta, SUM) ώστε να έχουμε τον συνολικό αριθμό αλλαγών.
- Τέλος, το delta κανονικοποιείται ώστε να προκύψει το κλάσμα των αλλαγών και συγκρίνεται με το threshold, ενώ υπάρχει και ανώτατο όριο επαναλήψεων loop_threshold.

5) Συγκέντρωση αποτελεσμάτων membership (main.c)

- Το τελικό membership ανά αντικείμενο υπολογίζεται τοπικά σε κάθε rank (membership[rank_numObjs]).
- Για να σχηματιστεί το πλήρες διάνυσμα membership για όλα τα αντικείμενα στον root, χρησιμοποιούμε MPI_Gatherv (αντί Gather), ξανά λόγω άνισων μεγεθών (remainder).
- Τα recvcounts/displs υπολογίζονται στο rank 0 σύμφωνα με την ίδια πολιτική κατανομής και μεταδίδονται με MPI_Bcast, ώστε όλα τα ranks να καλέσουν το MPI_Gatherv.

Συνολικά, οι κρίσιμες σχεδιαστικές επιλογές ήταν:

1. Data decomposition ανά object (όχι ανά coords), ώστε κάθε rank να εκτελεί ανεξάρτητο, γραμμικό υπολογισμό πάνω σε συνεχόμενη μνήμη.
2. Χρήση Scatterv/Gatherv, για σωστή διαχείριση N mod P.
3. Χρήση Allreduce για τα κοινά μεγέθη (sums, counts, delta), ώστε να αποφεύγεται το master bottleneck και να κρατάμε τα centroids συγχρονισμένα χωρίς πρόσθετα broadcasts ανά επανάληψη.

Τα αρχεία αυτά (file_io.c, main.c και kmeans.c) που περιέχουν την υλοποίησή μας παρατίθενται ακολούθως:

a6/kmeans/file_io.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>     /* strtok() */
#include <sys/types.h>  /* open() */
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>     /* read(), close() */
#include <mpi.h>

#include "kmeans.h"

double * dataset_generation(int numObjs, int numCoords, long *rank_numObjs)
{
    double * objects = NULL, * rank_objects = NULL;
    long i, j, k;

    // Random values that will be generated will be between 0 and 10.
    double val_range = 10;

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    /*
     * TODO: Calculate number of objects that each rank will examine (*rank_numObjs)
     */
    *rank_numObjs = numObjs / size;
    if (rank < numObjs % size) {
        (*rank_numObjs)++;
    }


    /* allocate space for objects[][] and read all objects */
    int sendcounts[size], displs[size];
    if (rank == 0) {
        objects = (typeof(objects)) malloc(numObjs * numCoords * sizeof(*objects));

        /*
         * TODO: Calculate sendcounts and displs, which will be used to scatter data to each rank.
         * Hint: sendcounts: number of elements sent to each rank
         *       displs: displacement of each rank's data
         */

        int count = 0;
        int remainder = numObjs % size;

        for (k = 0; k < size; k++) {

            int k_numObjs = numObjs / size;
            if (k < remainder) {
                k_numObjs++;
```

```
52              }
53
54              sendcounts[k] = k_numObjs * numCoords;
55              displs[k] = count;
56              count += sendcounts[k];
57          }
58      }
59
60      /*
61       * TODO: Broadcast the sendcounts and displs arrays to other ranks
62       */
63      MPI_Bcast(sendcounts, size, MPI_INT, 0, MPI_COMM_WORLD);
64      MPI_Bcast(displs, size, MPI_INT, 0, MPI_COMM_WORLD);
65
66
67      /* allocate space for objects[][] (for each rank separately) and read all objects */
68      rank_objects = (typeof(rank_objects)) malloc((*rank_numObjs) * numCoords *
    sizeof(*rank_objects));
69
70      /* rank 0 will generate data for the objects array. This array will be used later to
    scatter data to each rank. */
71      if (rank == 0) {
72          for (i=0; i<numObjs; i++)
73          {
74              unsigned int seed = i;
75              for (j=0; j<numCoords; j++)
76              {
77                  objects[i*numCoords + j] = (rand_r(&seed) / ((double) RAND_MAX)) *
    val_range;
78                  if (_debug && i == 0)
79                      printf("object[i=%ld][j=%ld]=%f\n",i,j,objects[i*numCoords + j]);
80              }
81          }
82      }
83
84      /*
85       * TODO: Scatter objects to every rank. (hint: each rank may receive different number
    of objects)
86       */
87      MPI_Scatterv(objects, sendcounts, displs, MPI_DOUBLE, rank_objects, sendcounts[rank],
    MPI_DOUBLE, 0, MPI_COMM_WORLD);
88
89
90      if (rank == 0)
91          free(objects);
92
93      return rank_objects;
94  }
95
```

a6/kmeans/kmeans.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#include "kmeans.h"

// square of Euclid distance between two multi-dimensional points
inline static double euclid_dist_2(int    numdims,  /* no. dimensions */
                                   double * coord1,   /* [numdims] */
                                   double * coord2)   /* [numdims] */
{
    int i;
    double ans = 0.0;

    for(i=0; i<numdims; i++)
        ans += (coord1[i]-coord2[i]) * (coord1[i]-coord2[i]);

    return ans;
}

inline static int find_nearest_cluster(int     numClusters, /* no. clusters */
                                       int     numCoords,   /* no. coordinates */
                                       double * object,      /* [numCoords] */
                                       double * clusters)    /* [numClusters][numCoords]
*/
{
    int index, i;
    double dist, min_dist;

    // find the cluster id that has min distance to object
    index = 0;
    min_dist = euclid_dist_2(numCoords, object, clusters);

    for(i=1; i<numClusters; i++) {
        dist = euclid_dist_2(numCoords, object, &clusters[i*numCoords]);
        // no need square root
        if (dist < min_dist) { // find the min and its array index
            min_dist = dist;
            index    = i;
        }
    }
    return index;
}

void kmeans(double * objects,         /* in: [numObjs][numCoords] */
            int     numCoords,        /* no. coordinates */
            int     numObjs,          /* no. objects */
            int     numClusters,      /* no. clusters */
            double   threshold,       /* minimum fraction of objects that change
membership */
            long    loop_threshold,   /* maximum number of iterations */
            int   * membership,       /* out: [numObjs] */
```

```
51              double * clusters)        /* out: [numClusters][numCoords] */
52  {
53      int i, j;
54      int index, loop=0;
55      double timing = 0;
56
57      /* Every variable has its "rank_" version, which is used to store local data,
58       * and its "new" version, which is used to store global data.
59       */
60      double rank_delta, delta = 0;               // fraction of objects whose clusters
    change in each loop
61      int * rank_newClusterSize, * newClusterSize; // [numClusters]: no. objects assigned in
    each new cluster
62      double * rank_newClusters, *newClusters;     // [numClusters][numCoords]
63
64      // Get rank of this process
65      int rank;
66      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
67
68      // initialize membership
69      for (i=0; i<numObjs; i++)
70          membership[i] = -1;
71
72      // initialize rank_newClusterSize and rank_newClusters to all 0
73      rank_newClusterSize = (typeof(rank_newClusterSize)) calloc(numClusters,
    sizeof(*rank_newClusterSize));
74      rank_newClusters    = (typeof(rank_newClusters))  calloc(numClusters * numCoords,
    sizeof(*rank_newClusters));
75      newClusterSize      = (typeof(newClusterSize)) calloc(numClusters,
    sizeof(*newClusterSize));
76      newClusters         = (typeof(newClusters))  calloc(numClusters * numCoords,
    sizeof(*newClusters));
77
78      timing = wtime();
79      do {
80          // before each loop, set cluster data to 0
81          for (i=0; i<numClusters; i++) {
82              for (j=0; j<numCoords; j++)
83                  rank_newClusters[i*numCoords + j] = 0.0;
84              rank_newClusterSize[i] = 0;
85          }
86
87          rank_delta = 0.0;
88
89          for (i=0; i<numObjs; i++) {
90              // find the array index of nearest cluster center
91              index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords],
    clusters);
92
93              // if membership changes, increase rank_delta by 1
94              if (membership[i] != index)
95                  rank_delta += 1.0;
96
97              // assign the membership to object i
```

```
98              membership[i] = index;
99
100             // update new cluster centers : sum of objects located within
101             rank_newClusterSize[index]++;
102             for (j=0; j<numCoords; j++)
103                 rank_newClusters[index*numCoords + j] += objects[i*numCoords + j];
104         }
105
106         /*
107          * TODO: Perform reduction of cluster data (rank_newClusters, rank_newClusterSize)
    from local arrays to shared.
108          */
109         MPI_Allreduce(rank_newClusters, newClusters, numClusters * numCoords, MPI_DOUBLE,
    MPI_SUM, MPI_COMM_WORLD);
110         MPI_Allreduce(rank_newClusterSize, newClusterSize, numClusters, MPI_INT, MPI_SUM,
    MPI_COMM_WORLD);
111
112
113         // average the sum and replace old cluster centers with newClusters
114         for (i=0; i<numClusters; i++) {
115             if (newClusterSize[i] > 0) {
116                 for (j=0; j<numCoords; j++) {
117                     clusters[i*numCoords + j] = newClusters[i*numCoords + j] /
    newClusterSize[i];
118                 }
119             }
120         }
121
122         /*
123          * TODO: Perform reduction from rank_delta variable to delta variable, that will
    be used for convergence check.
124          */
125         MPI_Allreduce(&rank_delta, &delta, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
126
127         // Get fraction of objects whose membership changed during this loop. This is used
    as a convergence criterion.
128         delta /= numObjs;
129
130         loop++;
131         //printf("\r\tcompleted loop %d", loop);
132         //fflush(stdout);
133     } while (delta > threshold && loop < loop_threshold);
134
135     timing = wtime() - timing;
136     if (rank == 0) fprintf(stdout, "        nloops = %3d   (total = %7.4fs)  (per loop =
    %7.4fs)\n", loop, timing, timing/loop);
137
138     free(rank_newClusters);
139     free(rank_newClusterSize);
140     free(newClusters);
141     free(newClusterSize);
142 }
143
```

a6/kmeans/main.c

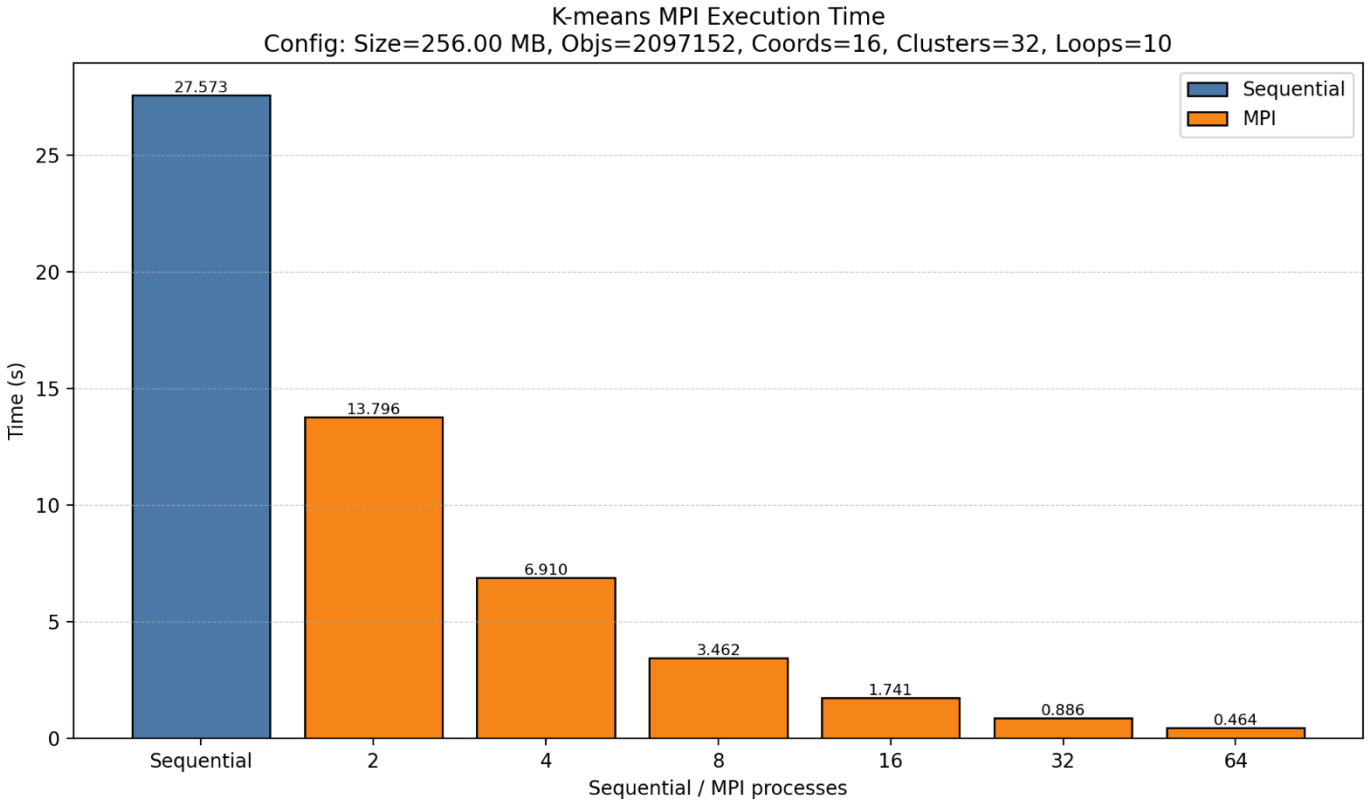
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>     /* strtok() */
#include <sys/types.h>  /* open() */
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>     /* getopt() */
#include <mpi.h>

int _debug;
#include "kmeans.h"

static void usage(char *argv0) {
    char *help =
        "Usage: %s [switches]\n"
        "       -c num_clusters    : number of clusters (must be > 1)\n"
        "       -s size            : size of examined dataset\n"
        "       -n num_coords      : number of coordinates\n"
        "       -t threshold       : threshold value (default : 0.001)\n"
        "       -l loop_threshold  : iterations threshold (default : 10)\n"
        "       -d                 : enable debug mode\n"
        "       -h                 : print this help information\n";
    fprintf(stderr, help, argv0);
    exit(-1);
}

int main(int argc, char **argv)
{
    long i, j, opt;
    extern char* optarg;
    extern int optind;

    long    numClusters=0, numCoords=0, numObjs=0;
    long    rank_numObjs=0;
    int   * membership;     // [rank_numObjs] this array will contain membership information for this rank's objects
    int   * tot_membership; // [numObjs]      this array will contain membership information for all objects
    double * objects;       // [numObjs * numCoords] data  objects
    double * clusters;      // [numClusters * numCoords] cluster center
    double   dataset_size = 0, threshold;
    long    loop_threshold;
    double  io_timing_read;

    /* some default values */
    _debug         = 0;
    threshold      = 0.001;
    loop_threshold = 10;
    numClusters    = 0;

    while ( (opt = getopt(argc,argv,"n:t:l:c:s:dh")) != EOF) {
        switch (opt) {
```

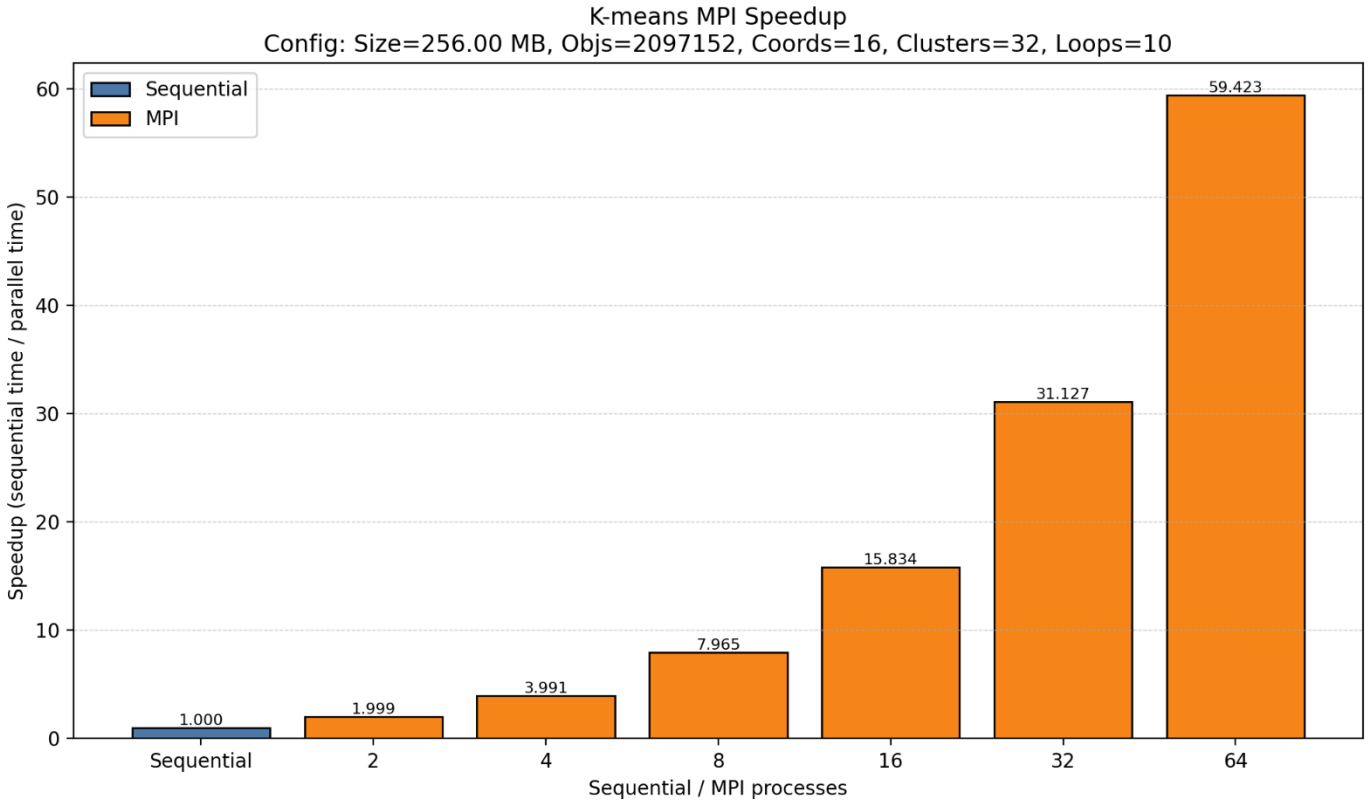
```
51              case 'c': numClusters = atol(optarg);
52                        break;
53              case 't': threshold=atof(optarg);
54                        break;
55              case 'l': loop_threshold=atol(optarg);
56                        break;
57              case 's': dataset_size=atof(optarg);
58                        break;
59              case 'n': numCoords=atol(optarg);
60                        break;
61              case 'd': _debug = 1;
62                        break;
63              case 'h':
64              default: usage(argv[0]);
65                        break;
66          }
67      }
68      if (numClusters <= 1) {
69          usage(argv[0]);
70      }
71
72      int rank, size;
73      MPI_Init(&argc,&argv);
74      MPI_Comm_rank(MPI_COMM_WORLD,&rank);
75      MPI_Comm_size(MPI_COMM_WORLD,&size);
76
77      numObjs = (dataset_size*1024*1024) / (numCoords*sizeof(double));
78
79      if (numObjs < numClusters) {
80          if (rank == 0) printf("Error: number of clusters must be larger than the number of
    data points to be clustered.\n");
81          MPI_Finalize();
82          return 1;
83      }
84      if (rank == 0) printf("dataset_size = %.2f MB    numObjs = %ld    numCoords = %ld
    numClusters = %ld\n", dataset_size, numObjs, numCoords, numClusters);
85
86      objects = dataset_generation(numObjs, numCoords, &rank_numObjs);
87
88      // Allocate space for clusters (coordinates of cluster centers)
89      clusters = (double*)  malloc(numClusters * numCoords * sizeof(double));
90
91      // The first numClusters elements are selected as initial centers. Only rank 0 needs
    to calculate this, and later broadcast it to all ranks.
92      if (rank == 0) {
93          for (i=0; i<numClusters; i++)
94              for (j=0; j<numCoords; j++)
95                  clusters[i*numCoords + j] = objects[i*numCoords + j];
96
97          // check initial cluster centers for repetition
98          if (check_repeated_clusters(numClusters, numCoords, clusters) == 0) {
99              printf("Error: some initial clusters are repeated. Please select distinct
    initial centers\n");
100             MPI_Finalize();
```

```
101             return 1;
102         }
103         /*
104         printf("Initial cluster centers:\n");
105         for (i=0; i<numClusters; i++) {
106             printf("(0) clusters[%ld] =",i);
107             for (j=0; j<numCoords; j++)
108                 printf(" %6.6f", clusters[i*numCoords + j]);
109             printf("\n");
110         }
111         */
112     }
113 
114     /*
115      * TODO: Broadcast initial cluster positions to all ranks
116      */
117     MPI_Bcast(clusters, numClusters * numCoords, MPI_DOUBLE, 0, MPI_COMM_WORLD);
118 
119 
120     // membership: the cluster id for each data object
121     membership = (int*) malloc(rank_numObjs * sizeof(int));
122     tot_membership = (int*) malloc(numObjs * sizeof(int));
123 
124     // start the core computation
125     /*
126      * TODO: Fix number of objects that this kmeans function call will process
127      */
128     kmeans(objects, numCoords, rank_numObjs, numClusters, threshold, loop_threshold,
    membership, clusters);
129 
130     /*
131     if (rank == 0) {
132         printf("Final cluster centers:\n");
133         for (i=0; i<numClusters; i++) {
134             printf("clusters[%ld] = ",i);
135             for (j=0; j<numCoords; j++)
136                 printf("%6.6f ", clusters[i*numCoords + j]);
137             printf("\n");
138         }
139     }
140     */
141 
142     // Gather membership information from all ranks to tot_membership
143     int recvcounts[size], displs[size];
144     if (rank == 0) {
145         /* TODO: Calculate recvcounts and displs, which will be used to gather data from
    each rank.
146          * Hint: recvcounts: number of elements received from each rank
147          *       displs: displacement of each rank's data
148          */
149         int sum_disp = 0;
150         int remainder = numObjs % size;
151 
152         for (j = 0; j < size; j++) {
```

```
153             // Υπολογισμός πόσα αντικείμενα περιμένουμε από το rank 'j'
154             int j_numObjs = numObjs / size;
155             if (j < remainder) {
156                 j_numObjs++;
157             }
158 
159             recvcounts[j] = j_numObjs; // Εδώ είναι σκέτα αντικείμενα (int)
160             displs[j] = sum_disp;
161             sum_disp += recvcounts[j];
162         }
163     }
164 
165     /*
166      * TODO: Broadcast the recvcounts and displs arrays to other ranks.
167      */
168     MPI_Bcast(recvcounts, size, MPI_INT, 0, MPI_COMM_WORLD);
169     MPI_Bcast(displs, size, MPI_INT, 0, MPI_COMM_WORLD);
170 
171 
172     /*
173      * TODO: Gather membership information from every rank. (hint: each rank may send different number of objects)
174      */
175     MPI_Gatherv(membership, rank_numObjs, MPI_INT, tot_membership, recvcounts, displs, MPI_INT, 0, MPI_COMM_WORLD);
176 
177 
178     if (_debug && rank == 0)
179         for (i = 0; i < numObjs; ++i)
180             fprintf(stderr, "%d\n", tot_membership[i]);
181 
182     free(objects);
183     free(membership);
184     free(tot_membership);
185     free(clusters);
186 
187     MPI_Finalize();
188     return 0;
189 }
190 
```

Γ. Μετρήσεις και Διαγράμματα





Δ. Συμπεράσματα Μετρήσεων

Από τα διαγράμματα χρόνου εκτέλεσης και speedup (Config: Size=256MB, Objs=2,097,152, Coords=16, Clusters=32, Loops=10) παρατηρούμε εξαιρετικά καλή κλιμάκωση του MPI K-means, σχεδόν γραμμική (ιδανική), σε όλο το εύρος διεργασιών μέχρι και 64. Ο σειριακός χρόνος είναι 27.573 s, ενώ με 2/4/8/16/32/64 διεργασίες ο χρόνος μειώνεται σε 13.796/6.910/3.462/1.741/0.886/0.464 s αντίστοιχα. Αυτό μεταφράζεται σε speedup 1.999×, 3.991×, 7.965×, 15.834×, 31.127× και 59.423×, δηλαδή μικρή απόκλιση από την ιδανική κλιμάκωση, ακόμα και σε 64 διεργασίες.

Η συμπεριφορά αυτή είναι πλήρως αναμενόμενη αν λάβουμε υπόψη τη δομή του αλγορίθμου και τη θεωρία του παραλληλισμού κατανεμημένης μνήμης, καθώς:

1) Κυριαρχία υπολογιστικού τμήματος

Το βασικό κόστος ανά επανάληψη στον K-means είναι η ανάθεση (assignment): για κάθε σημείο υπολογίζονται αποστάσεις από όλα τα centroids. Η πολυπλοκότητα ανά iteration είναι περίπου O(N · K · d) (N objects, K clusters, d coords). Με τις παραμέτρους (N≈2.1M, K=32, d=16) το workload είναι πολύ μεγάλο και έτσι επιτυγχάνεται πολύ υψηλός παραλληλισμός ως προς τα objects. Με data decomposition ανά object, κάθε rank εκτελεί περίπου N/P ανεξάρτητους υπολογισμούς, άρα ο χρόνος υπολογισμού μειώνεται ~1/P.

2) Μικρός όγκος επικοινωνίας ανά iteration

Στο τέλος κάθε iteration απαιτείται συγχρονισμένη ενημέρωση των centroids και του κριτηρίου σύγκλισης. Αυτό υλοποιείται με συλλογικές επικοινωνίες (MPI_Allreduce) πάνω σε σχετικά λίγες και μικρές κοινές μεταβλητές:

- Τα αθροίσματα των νέων centroids έχουν μέγεθος K·d = 32·16 = 512 doubles (μερικά KB).
- Τα counts των clusters έχουν μέγεθος K (λίγες δεκάδες bytes/ints).
- Επιπλέον γίνεται ένα Allreduce για το delta (αριθμός αλλαγών membership).

Με βάση το κλασικό μοντέλο κόστους επικοινωνίας (latency/bandwidth, α–β model), το Allreduce έχει κόστος που αυξάνει περίπου με log(P) στα περισσότερα δέντρα/αλγορίθμους υλοποίησης. Εδώ όμως το μήνυμα είναι μικρό, οπότε (α) το

bandwidth cost είναι πολύ χαμηλό και (β) το πρόσθετο latency/synchronization overhead παραμένει μικρό σε σχέση με το τεράστιο compute phase που κλιμακώνει με 1/P. Το αποτέλεσμα είναι ότι η επικοινωνία δεν κυριαρχεί τον υπολογισμό, άρα διατηρούμε υψηλή επίδοση.

3) Επίδραση συγχρονισμού και συλλογικών επικοινωνιών στην απόκλιση από το ιδανικό

Παρότι η κλιμάκωση είναι σχεδόν ιδανική, το speedup υπολείπεται ελαφρά του P όσο αυξάνουμε τις διεργασίες (π.χ. 64: 59.4× αντί 64×). Αυτό αποδίδεται κυρίως σε:

- Συλλογικές επικοινωνίες (Allreduce) που λειτουργούν ως φράγμα (implicit synchronization): κάθε iteration ολοκληρώνεται μόνο όταν φτάσουν όλα τα ranks στο ίδιο σημείο.
- Αυξανόμενο κόστος latency με το πλήθος διεργασιών (περισσότερα communication steps και πιθανή επικοινωνία όταν απλώνουμε τα ranks σε πολλούς κόμβους).
- Σταθερά κόστη ανά iteration (overheads) που δεν κλιμακώνονται με 1/P (π.χ. bookkeeping, ενημέρωση δομών, overhead MPI runtime).
- Στη λογική του νόμου του Amdahl, η συνολική επιτάχυνση περιορίζεται από οποιοδήποτε «μη παραλληλοποιήσιμο» ή μη κλιμακούμενο κομμάτι. Εδώ το σειριακό κλάσμα είναι εξαιρετικά μικρό, άρα ο μέγιστος παραλληλισμός είναι πολύ υψηλός - κάτι που επιβεβαιώνεται από το ότι η αποδοτικότητα παραμένει μεγάλη ακόμα και σε 64 διεργασίες (περίπου 93%: 59.4/64).

4) Load balance και κατανομή δεδομένων

Η κατανομή των objects έγινε ισοζυγισμένα (block distribution με remainder μέσω Scatterv), οπότε η διαφορά φορτίου ανά rank είναι το πολύ ένα object. Εφόσον το κύριο κόστος είναι ίδιο ανά object (υπολογισμός αποστάσεων προς όλα τα centroids), το load imbalance πρακτικά μηδενίζεται. Αυτό περιορίζει το φαινόμενο όπου κάποια ranks καθυστερούν και κρατούν πίσω τα υπόλοιπα στο Allreduce.

Συνολικά, τα αποτελέσματα δείχνουν ότι ο K-means στο συγκεκριμένο configuration είναι ιδιαίτερα κατάλληλος για strong scaling σε MPI: το μεγαλύτερο μέρος της εργασίας είναι data-parallel, η επικοινωνία ανά iteration είναι μικρού όγκου (κυρίως συλλογικά reductions), και η κατανομή φορτίου είναι καλά

ισοζυγισμένη. Η μικρή πτώση αποδοτικότητας σε μεγάλες τιμές P ερμηνεύεται από την αυξανόμενη σχετική συμμετοχή των σταθερών overheads και των συλλογικών επικοινωνιών (latency/synchronization), όπως προβλέπει η θεωρία των συστημάτων κατανεμημένης μνήμης.

Ε. Σύγκριση με OpenMP

Για τη σύγκριση με την υλοποίηση σε OpenMP και αρχιτεκτονική κοινής μνήμης, χρησιμοποιούμε τα αποτελέσματα της αναφοράς του OpenMP της άσκησης 2 (με τη δική μας σημειολογία, parlab05_report_a2_final.pdf) για τον K-means.

Σημείωση: Οι απόλυτοι σειριακοί χρόνοι των δύο υλοποιήσεων (MPI vs OpenMP) δεν είναι απαραίτητα απευθείας συγκρίσιμοι (διαφορετικός κώδικας/μεθοδολογία timing), όμως οι τάσεις κλιμάκωσης και τα bottlenecks είναι ξεκάθαρα.

1) OpenMP – Naive shared

Στην αφελή αυτή εκδοχή, οι ενημερώσεις των κοινόχρηστων πινάκων (newClusters, newClusterSize και delta) προστατεύονται με atomic operations. Αυτό δημιουργεί έντονο synchronization bottleneck: πολλά νήματα προσπαθούν να ενημερώσουν τα ίδια cache lines, προκαλώντας serialization.

- No affinity: Tseq=12.65 s, ενώ για 2/4/8/16/32/64 νήματα παίρνουμε 9.92/12.70/11.85/10.46/11.31/9.14 s (μη ικανοποιητική κλιμάκωση, ακόμη και χειρότερη από το σειριακό σε σημεία).
- Default affinity: Tseq=12.64 s, και για 4/8 νήματα βελτιώνεται (5.78/3.76 s), αλλά μετά χειροτερεύει (16/32/64: 8.56/12.01/10.07 s).

Η βελτίωση με affinity συμφωνεί με τη θεωρία NUMA: το pinning μειώνει τις remote προσπελάσεις και κρατά καλύτερο locality, όμως δεν λύνει το βασικό πρόβλημα της σχεδίασης (atomics σε shared δομές), άρα το contention παραμένει και περιορίζει την κλιμάκωση.

2) OpenMP – Copied clusters + reduction

Στη βελτιστοποιημένη αυτή εκδοχή, κάθε νήμα γράφει σε ιδιωτικά (thread-local) arrays (local_newClusters/local_newClusterSize) και στη συνέχεια γίνεται συνδυασμός των αποτελεσμάτων (reduction/merge). Έτσι:

- Αφαιρείται το συνεχές atomic contention από τον κυρίως βρόχο.
- Οι ενημερώσεις γίνονται κατά βάση τοπικές (καλύτερη cache locality) και ο συγχρονισμός περιορίζεται σε μικρό/συγκεντρωμένο τμήμα.

Αποτελέσματα: Tseq=12.64 s και χρόνοι 2/4/8/16/32/64: 7.36/3.92/2.13/1.14/0.60/0.48 s. Το speedup φτάνει ~26.3× στα 64 νήματα, με υψηλή αποδοτικότητα μέχρι τα 32 νήματα (~21.1×), και πτώση πέρα από εκεί, κάτι αναμενόμενο από shared-memory περιορισμούς (memory bandwidth saturation, αυξημένο overhead/NUMA effects).

3) MPI – Distributed-memory

Στη δική μας υλοποίηση, το workload διαμοιράζεται ανά object, ενώ η επικοινωνία ανά iteration είναι μικρού όγκου (κυρίως MPI_Allreduce για sums/counts/delta). Αυτό ταιριάζει ιδανικά στο μοντέλο latency/bandwidth: μικρά collectives (μερικά KB) με κόστος που αυξάνει αργά με P (συνήθως ~logP), ενώ το compute μειώνεται περίπου με 1/P.

Μετρήσεις (MPI): Tseq=27.573 s και χρόνοι 2/4/8/16/32/64: 13.796/6.910/3.462/1.741/0.886/0.464 s, με speedup 1.999× έως 59.423×. Η αποδοτικότητα παραμένει πολύ υψηλή ακόμη και στα 64 ranks, επειδή:

- αποφεύγονται shared-memory contention/coherence bottlenecks (ιδιωτική μνήμη ανά process),
- και το communication volume για centroids/delta είναι μικρό σε σχέση με το compute O(N·K·d).

Συμπεράσματα σύγκρισης

1. Το OpenMP μπορεί να είναι εξαιρετικά καλό εντός ενός κόμβου, αλλά απαιτεί προσεκτικό σχεδιασμό για να μειωθούν τα synchronization/NUMA bottlenecks, ιδίως η αποφυγή atomics σε hot paths και η χρήση thread-local δομών + reduction.
2. Η MPI κλιμακώνει πολύ καλύτερα σε πολλούς κόμβους, ειδικά όταν το πρόβλημα είναι compute-intense και η επικοινωνία ανά iteration είναι μικρού μεγέθους (όπως στον K-means με K·d μικρό).
3. Πρακτικά, η θεωρία υποδεικνύει ως βέλτιστη γενίκευση σε clusters έναν υβριδικό σχεδιασμό: MPI μεταξύ κόμβων και OpenMP εντός κόμβου, ώστε να εκμεταλλευόμαστε και την τοπολογία του συστήματος και να περιορίζουμε τόσο το inter-node communication όσο και το intra-node contention.

▪ Ενότητα 4.2 – Διάδοση Θερμότητας σε 2 Διαστάσεις

Α. Εισαγωγή

Στην ενότητα αυτή θα υλοποιήσουμε την επίλυση του δισδιάστατου προβλήματος διάδοσης θερμότητας, σε αρχιτεκτονική κατανεμημένης μνήμης, με χρήση του πρωτοκόλλου MPI. Μετά τη διακριτοποίηση του συνεχούς προβλήματος σε πλέγμα, η ενημέρωση των εσωτερικών κελιών οδηγεί σε ένα επαναληπτικό σχήμα τύπου Jacobi, όπου κάθε νέο κελί προκύπτει από τον μέσο όρο των τεσσάρων γειτονικών τιμών του (πάνω/κάτω/αριστερά/δεξιά). Η μέθοδος αυτή είναι ιδιαίτερα κατάλληλη για την ανάδειξη του παραλληλισμού, καθώς ο υπολογισμός κάθε κελιού εξαρτάται μόνο από τιμές της προηγούμενης επανάληψης και έτσι γίνεται πολύ compute intense.

Ειδικότερα, στόχος μας είναι:

- να κατανείμουμε το δισδιάστατο πλέγμα σε υποπεριοχές και να αναθέσουμε καθεμία από αυτές σε μία διεργασία MPI,
- να υλοποιήσουμε την απαραίτητη ανταλλαγή «halo/ghost» ορίων μεταξύ των γειτονικών διεργασιών, ώστε κάθε διεργασία να μπορεί να υπολογίζει σωστά τα κελιά που βρίσκονται κοντά στα όριά της,
- και να χρονομετρήσουμε τον χρόνο υπολογισμού και τον συνολικό χρόνο (υπολογισμοί + επικοινωνίες) όπως απαιτεί η άσκηση.

Λόγω περιορισμένου χρόνου, δεν υλοποιήθηκαν τα bonus ερωτήματα (Gauss–Seidel SOR και Red–Black SOR), αν και προσπαθήσαμε (όπως φαίνεται και στον scirouter), απλά απέτυχαν στην σύγκλιση και έτσι τα παραλείψαμε, καθώς θεωρήσαμε ότι δεν ήταν ορθά και το debugging ήταν αρκετά χρονοβόρο. Η υλοποίηση καλύπτει την εκδοχή Jacobi με δυνατότητα (μέσω compile-time flags) είτε ελέγχου σύγκλισης ανά C επαναλήψεις είτε εκτέλεσης για σταθερό πλήθος επαναλήψεων (T=256), σύμφωνα με τα ζητούμενα.

Β. Υλοποίηση

Η υλοποίηση πραγματοποιείται στο mpi_jacobi.c και βασίζεται σε μια 2D κατανομή του πλέγματος, με καρτεσιανή τοπολογία MPI, ghost cells, παράγωγους τύπου δεδομένων και συλλογικές επικοινωνίες για σύγκλιση και χρονομέτρηση.

1) Ορίσματα – Διάταξη διεργασιών (process grid)

Το πρόγραμμα δέχεται ως είσοδο τις διαστάσεις του πλέγματος X, Y και τη διάταξη διεργασιών Px, Py. Έτσι, δημιουργείται ένα 2D Cartesian communicator (MPI_Cart_create) με μη περιοδικά όρια (periods={0,0}) και κάθε διεργασία αποκτά τις συντεταγμένες της (MPI_Cart_coords). Η καρτεσιανή αυτή τοπολογία διευκολύνει την εύρεση γειτόνων με MPI_Cart_shift, κάτι που ταιριάζει άμεσα στο μοτίβο επικοινωνίας της εξίσωσης θερμότητας.

2) Padding και τοπικές διαστάσεις

Για να καλυφθούν περιπτώσεις όπου οι διαστάσεις δεν διαιρούνται ακριβώς από το Px,Py, εφαρμόζεται padding:

- local[i] = ceil(global[i] / grid[i])
- global_padded[i] = local[i] * grid[i]

Έτσι όλες οι διεργασίες χειρίζονται ομοιόμορφες τοπικές διαστάσεις, ενώ τα επιπλέον padded κελιά δεν πρέπει να ενημερώνονται ως πραγματικό πεδίο και αντιμετωπίζονται με προσεκτικό ορισμό ορίων επανάληψης.

3) Δομές δεδομένων – Ghost cells

Κάθε διεργασία δεσμεύει δύο τοπικούς πίνακες u_previous και u_current διαστάσεων (localX+2)×(localY+2). Το +2 αντιστοιχεί σε ghost layers (μία γραμμή/στήλη γύρω από τον πραγματικό τοπικό υποπίνακα), όπου αποθηκεύονται οι τιμές που λαμβάνονται από τους γείτονες. Η Jacobi απαιτεί δύο πεδία (παλιό/νέο), ώστε οι ενημερώσεις να βασίζονται αποκλειστικά στην προηγούμενη επανάληψη.

4) Αρχικοποίηση και Scatter του global πεδίου

Το rank 0 δεσμεύει τον global πίνακα U (με padded διαστάσεις) και τον αρχικοποιεί (init2d) σύμφωνα με τις συνοριακές συνθήκες του προβλήματος. Η διανομή των blocks προς τις διεργασίες γίνεται με MPI_Scatterv, ώστε να σταλούν/ληφθούν οι 2D υποπίνακες χωρίς manual packing:

- global_block: MPI_Type_vector πάνω στο padded global (stride = global_padded[1]) και resized extent sizeof(double), ώστε τα displacements να δίνονται σε double units.
- local_block: MPI_Type_vector πάνω στο local buffer (stride = localY+2) και resized, ώστε να γεμίζει το εσωτερικό τμήμα u_previous[1][1].

Τα scattercounts=1 για κάθε διεργασία και τα scatteroffsets υπολογίζονται ώστε να δείχνουν στο σωστό (i,j) block του global πίνακα.

5) Εύρος υπολογισμού (αποφυγή global boundaries και padded κελιών)

Ορίζονται i_min..i_max και j_min..j_max ώστε:

- να μη γίνονται updates στα φυσικά όρια του συνολικού πλέγματος (στα borders κρατάμε τις συνοριακές συνθήκες),
- και στις διεργασίες που βρίσκονται τελευταίες σε κάθε διάσταση να μη γίνονται updates στα κελιά που αντιστοιχούν στο padding.

Έτσι, οι ενημερώσεις περιορίζονται αυστηρά στα πραγματικά εσωτερικά κελιά του domain.

6) Halo exchange

Για την ανταλλαγή ορίων χρησιμοποιούνται δύο communication datatypes:

- row_type: contiguous localY doubles (για αποστολή/λήψη ολόκληρης γραμμής),
- col_type: vector localX στοιχείων με stride (localY+2) (για αποστολή/λήψη στήλης).

Οι γείτονες (north/south/east/west) προκύπτουν μέσω MPI_Cart_shift. Σε κάθε iteration εκτελείται nonblocking ανταλλαγή (MPI_Isend/MPI_Irecv) των εξής:

- Η πρώτη εσωτερική γραμμή προς north και λήψη στο ghost row 0,
- Η τελευταία εσωτερική γραμμή προς south και λήψη στο ghost row localX+1,

- Η πρώτη εσωτερική στήλη προς west και λήψη στο ghost col 0,
- Η τελευταία εσωτερική στήλη προς east και λήψη στο ghost col localY+1,

και ολοκλήρωση με MPI_Waitall. Η χρήση nonblocking αποφεύγει deadlocks και προσφέρει καθαρό συμμετρικό μοτίβο επικοινωνίας μεταξύ γειτόνων.

7) Ενημέρωση Jacobi (υπολογιστικός πυρήνας)

Μετά την ανταλλαγή halo, ενημερώνεται το u_current στα εσωτερικά κελιά:

u_current[i][j] = (u_previous[i-1][j] + u_previous[i+1][j] +

u_previous[i][j-1] + u_previous[i][j+1]) / 4

Χρησιμοποιείται swap των pointers (u_previous ↔ u_current) στην αρχή κάθε iteration, ώστε να αποφεύγονται ακριβές αντιγραφές πινάκων.

8) Έλεγχος σύγκλισης (προαιρετικός, με compile flag)

Όταν είναι ενεργό το TEST_CONV, τότε υπολογίζεται ανά C επαναλήψεις τοπικά το converged και στη συνέχεια γίνεται MPI_Allreduce με MPI_LAND πάνω στο CART_COMM. Έτσι, η σύγκλιση αποφασίζεται globally: το global_converged γίνεται true μόνο αν όλες οι διεργασίες έχουν συγκλίνει. Χωρίς TEST_CONV, εκτελούνται πάντα T=256 επαναλήψεις (όπως ορίζει το σενάριο χωρίς σύγκλιση).

9) Χρονομέτρηση και συλλογή αποτελεσμάτων

Ο συνολικός χρόνος μετράται γύρω από ολόκληρο τον επαναληπτικό βρόχο, ενώ ο χρόνος υπολογισμού μετριέται μόνο γύρω από το double for των ενημερώσεων. Στο τέλος γίνεται MPI_Reduce (MPI_MAX) τόσο για total όσο και για compute time, ώστε να κρατήσουμε το critical-path (η πιο αργή διεργασία καθορίζει τον συνολικό χρόνο). Τέλος, το αποτέλεσμα συγκεντρώνεται στο rank 0 με MPI_Gatherv (ίδια λογική blocks/offsets όπως στο Scatterv) για εκτύπωση του midpoint και (προαιρετικά) την αποθήκευση του πλήρους πλέγματος.

Το αρχείο αυτό (mpi_jacobi.c) που περιέχει την υλοποίησή μας παρατίθεται ακολούθως:

a6/heat_transfer/mpi/mpi_jacobi.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include "mpi.h"
#include "utils.h"

int main(int argc, char ** argv) {
    int rank,size;
    int global[2],local[2]; //global matrix dimensions and local matrix dimensions
    int global_padded[2];   //padded global matrix dimensions
    int grid[2];            //processor grid dimensions
    int i,j,t;
    int global_converged=0,converged=0; //flags for convergence
    MPI_Datatype dummy;     //dummy datatype
    double omega;           //relaxation factor - useless for Jacobi

    struct timeval tts,ttf,tcs,tcf;   //Timers
    double ttotal=0,tcomp=0,total_time,comp_time;
    double t_conv=0.0;
    double t1,t2;

    double ** U, ** u_current, ** u_previous, ** swap;

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

    //----Read arguments----//
    if (argc!=5) {
        fprintf(stderr,"Usage: mpirun .... ./exec X Y Px Py");
        exit(-1);
    }
    else {
        global[0]=atoi(argv[1]);
        global[1]=atoi(argv[2]);
        grid[0]=atoi(argv[3]);
        grid[1]=atoi(argv[4]);
    }

    //----Create 2D-cartesian communicator----//
    MPI_Comm CART_COMM;
    int periods[2]={0,0};
    int rank_grid[2];

    MPI_Cart_create(MPI_COMM_WORLD,2,grid,periods,0,&CART_COMM);
    MPI_Cart_coords(CART_COMM,rank,2,rank_grid);

    //----Compute local dimensions & Padding----//
    for (i=0;i<2;i++) {
        if (global[i]%grid[i]==0) {
```

```
52              local[i]=global[i]/grid[i];
53              global_padded[i]=global[i];
54          }
55          else {
56              local[i]=(global[i]/grid[i])+1;
57              global_padded[i]=local[i]*grid[i];
58          }
59      }
60
61      //Initialization of omega
62      omega=2.0/(1+sin(3.14/global[0]));
63
64      //----Allocate global 2D-domain----//
65      if (rank==0) {
66          U=allocate2d(global_padded[0],global_padded[1]);
67          init2d(U,global[0],global[1]);
68      }
69
70      //----Allocate local 2D-subdomains----//
71      u_previous=allocate2d(local[0]+2,local[1]+2);
72      u_current=allocate2d(local[0]+2,local[1]+2);
73
74      //----Datatypes Definition----//
75      MPI_Datatype global_block;
76      MPI_Type_vector(local[0],local[1],global_padded[1],MPI_DOUBLE,&dummy);
77      MPI_Type_create_resized(dummy,0,sizeof(double),&global_block);
78      MPI_Type_commit(&global_block);
79
80      MPI_Datatype local_block;
81      MPI_Type_vector(local[0],local[1],local[1]+2,MPI_DOUBLE,&dummy);
82      MPI_Type_create_resized(dummy,0,sizeof(double),&local_block);
83      MPI_Type_commit(&local_block);
84
85      //----Scatter parameters----//
86      int * scatteroffset, * scattercounts;
87      if (rank==0) {
88          scatteroffset=(int*)malloc(size*sizeof(int));
89          scattercounts=(int*)malloc(size*sizeof(int));
90          for (i=0;i<grid[0];i++)
91              for (j=0;j<grid[1];j++) {
92                  scattercounts[i*grid[1]+j]=1;
93                  scatteroffset[i*grid[1]+j]=(local[0]*local[1]*grid[1]*i+local[1]*j);
94              }
95      }
96
97      //----Scatter----//
98      MPI_Scatterv(rank == 0 ? &U[0][0] : NULL,
99                   scattercounts, scatteroffset, global_block,
100                  &u_previous[1][1], 1, local_block,
101                  0, MPI_COMM_WORLD);
102
103     // Init u_current
104     for (i = 1; i <= local[0]; i++)
105         for (j = 1; j <= local[1]; j++)
```

```
106             u_current[i][j] = u_previous[i][j];
107
108     if (rank==0)
109         free2d(U);
110
111     //----Communication Datatypes----//
112     MPI_Datatype row_type, col_type;
113     MPI_Type_contiguous(local[1], MPI_DOUBLE, &row_type);
114     MPI_Type_commit(&row_type);
115     MPI_Type_vector(local[0], 1, local[1] + 2, MPI_DOUBLE, &col_type);
116     MPI_Type_commit(&col_type);
117
118     //----Find Neighbors----//
119     int north, south, east, west;
120     MPI_Cart_shift(CART_COMM, 0, 1, &north, &south);
121     MPI_Cart_shift(CART_COMM, 1, 1, &west, &east);
122
123     //---Define iteration ranges-----//
124     int i_min,i_max,j_min,j_max;
125
126     i_min = 1;
127     i_max = local[0];
128     j_min = 1;
129     j_max = local[1];
130
131     if (rank_grid[0] == 0) i_min = 2;
132     if (rank_grid[0] == grid[0] - 1) {
133         i_max = local[0] - (global_padded[0] - global[0]) - 1;
134         if (global_padded[0] == global[0]) i_max = local[0] - 1;
135     }
136
137     if (rank_grid[1] == 0) j_min = 2;
138     if (rank_grid[1] == grid[1] - 1) {
139          j_max = local[1] - (global_padded[1] - global[1]) - 1;
140          if (global_padded[1] == global[1]) j_max = local[1] - 1;
141     }
142
143     //----Computational core----//
144     gettimeofday(&tts, NULL);
145
146     #ifdef TEST_CONV
147     for (t=0;t<T && !global_converged;t++) {
148     #endif
149     #ifndef TEST_CONV
150     #undef T
151     #define T 256
152     for (t=0;t<T;t++) {
153     #endif
154
155         // 1. Swap
156         swap = u_previous;
157         u_previous = u_current;
158         u_current = swap;
159
```

```
160         // 2. Communication
161         MPI_Request reqs[8];
162         MPI_Status stats[8];
163         int req_cnt = 0;
164 
165         MPI_Isend(&u_previous[1][1], 1, row_type, north, 1, CART_COMM, &reqs[req_cnt++]);
166         MPI_Irecv(&u_previous[0][1], 1, row_type, north, 2, CART_COMM, &reqs[req_cnt++]);
167 
168         MPI_Isend(&u_previous[local[0]][1], 1, row_type, south, 2, CART_COMM,
    &reqs[req_cnt++]);
169         MPI_Irecv(&u_previous[local[0]+1][1], 1, row_type, south, 1, CART_COMM,
    &reqs[req_cnt++]);
170 
171         MPI_Isend(&u_previous[1][1], 1, col_type, west, 3, CART_COMM, &reqs[req_cnt++]);
172         MPI_Irecv(&u_previous[1][0], 1, col_type, west, 4, CART_COMM, &reqs[req_cnt++]);
173 
174         MPI_Isend(&u_previous[1][local[1]], 1, col_type, east, 4, CART_COMM,
    &reqs[req_cnt++]);
175         MPI_Irecv(&u_previous[1][local[1]+1], 1, col_type, east, 3, CART_COMM,
    &reqs[req_cnt++]);
176 
177         MPI_Waitall(req_cnt, reqs, stats);
178 
179         // 3. Computation
180         gettimeofday(&tcs, NULL);
181         for (i = i_min; i <= i_max; i++) {
182             for (j = j_min; j <= j_max; j++) {
183                 u_current[i][j] = (u_previous[i-1][j] + u_previous[i+1][j] +
184                                    u_previous[i][j-1] + u_previous[i][j+1]) / 4.0;
185             }
186         }
187         gettimeofday(&tcf, NULL);
188         tcomp += (tcf.tv_sec - tcs.tv_sec) + (tcf.tv_usec - tcs.tv_usec) * 0.000001;
189 
190         // 4. Convergence Check
191         #ifdef TEST_CONV
192         if (t % C == 0) {
193             converged = converge(u_previous, u_current, i_min, i_max, j_min, j_max);
194 
195         t1=MPI_Wtime(); //
196             MPI_Allreduce(&converged, &global_converged, 1, MPI_INT, MPI_LAND, CART_COMM);
197 
198         t2=MPI_Wtime();
199             t_conv+=(t2-t1);
200         }
201         #endif
202 
203     }
204     gettimeofday(&ttf,NULL);
205 
206     ttotal=(ttf.tv_sec-tts.tv_sec)+(ttf.tv_usec-tts.tv_usec)*0.000001;
207 
208     MPI_Reduce(&ttotal,&total_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
209     MPI_Reduce(&tcomp,&comp_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
```

```
210
211     //----Gather results----//
212     if (rank==0) {
213             U=allocate2d(global_padded[0],global_padded[1]);
214     }
215
216     MPI_Gatherv(&u_current[1][1], 1, local_block,
217                 rank == 0 ? &U[0][0] : NULL,
218                 scattercounts, scatteroffset, global_block,
219                 0, MPI_COMM_WORLD);
220
221     //----Printing results----//
222     if (rank==0) {
223         printf("Jacobi X %d Y %d Px %d Py %d Iter %d ComputationTime %lf TotalTime %lf
    ConvergenceTime %lf midpoint %lf\n",
224
    global[0],global[1],grid[0],grid[1],t,comp_time,total_time,t_conv,U[global[0]/2]
    [global[1]/2]);
225
226         #ifdef PRINT_RESULTS
227         char * s=malloc(50*sizeof(char));
228         sprintf(s,"resJacobiMPI_%dx%d_%dx%d",global[0],global[1],grid[0],grid[1]);
229         fprint2d(s,U,global[0],global[1]);
230         free(s);
231         #endif
232     }
233
234     // Free Datatypes before Finalize
235     MPI_Type_free(&row_type);
236     MPI_Type_free(&col_type);
237     MPI_Type_free(&global_block);
238     MPI_Type_free(&local_block);
239     if(rank==0){
240         free(scattercounts);
241         free(scatteroffset);
242     }
243
244     MPI_Finalize();
245     return 0;
246 }
247
```

▪ 4.2.1 – Μετρήσεις με Έλεγχο Σύγκλισης

Α. Εισαγωγή

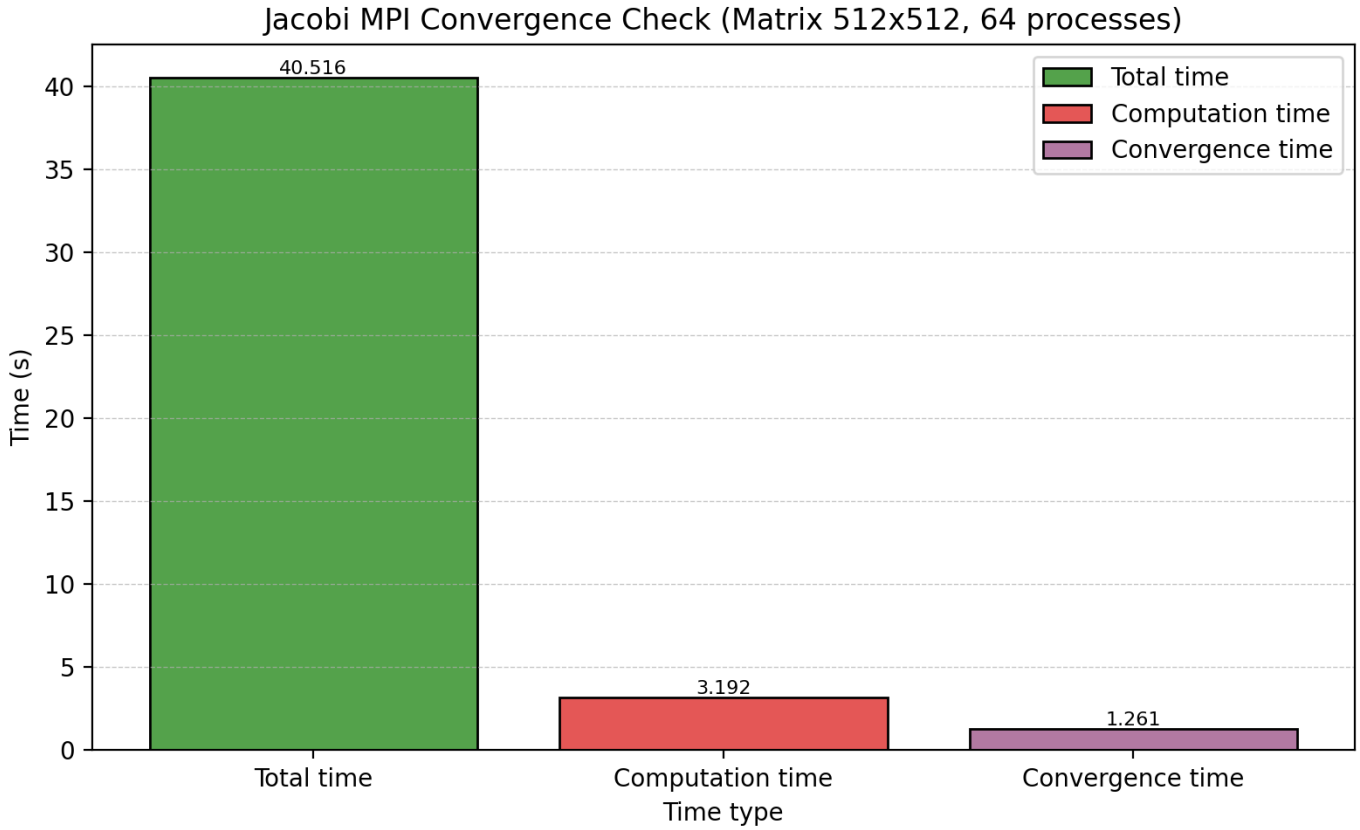
Στην υποενότητα αυτή, θα αξιολογήσουμε την υλοποίηση σε MPI της μεθόδου Jacobi για τη δισδιάστατη εξίσωση θερμότητας, όταν ο αλγόριθμος τερματίζει με βάση τον έλεγχο σύγκλισης. Σύμφωνα με την εκφώνηση, εκτελούμε το σενάριο για πλέγμα 512×512 με 64 MPI διεργασίες και καταγράφουμε ξεχωριστά:

- Συνολικό χρόνο: τον χρόνο του επαναληπτικού τμήματος, όπως καθορίζεται από την πιο αργή διεργασία (critical path).
- Χρόνο υπολογισμών: τον χρόνο που δαπανάται αποκλειστικά στον υπολογιστικό πυρήνα (ενημέρωση των εσωτερικών κελιών).
- Χρόνο ελέγχου σύγκλισης: τον χρόνο που αντιστοιχεί στον περιοδικό έλεγχο σύγκλισης (τοπικός υπολογισμός κριτηρίου + απόφαση μέσω συλλογικής επικοινωνίας).

Η θεωρητική σημασία του πειράματος είναι ότι ο Jacobi είναι ένας εύκολα παραλληλίσιμος αλγόριθμος (χρησιμοποιεί μόνο τιμές της προηγούμενης επανάληψης), όμως σε κατανεμημένη μνήμη απαιτεί ανταλλαγή ορίων σε κάθε iteration και επιπλέον επικοινωνίες για τον τελικό τερματισμό. Άρα, το συνολικό κόστος προκύπτει από την ισορροπία μεταξύ υπολογισμών (που μειώνονται με το πλήθος διεργασιών) και επικοινωνιών/συγχρονισμών (που δεν μειώνονται αντίστοιχα και συχνά αυξάνουν με την κλίμακα).

Σημείωση για τις bonus μεθόδους: η εκφώνηση ζητά σύγκριση και με Gauss–Seidel SOR και Red–Black SOR. Τις προσεγγίσαμε (σε επίπεδο σχεδίασης/δομής επικοινωνίας), αλλά δεν ολοκληρώσαμε την υλοποίησή τους λόγω χρόνου και δύσκολου debugging (αν και έχουμε σχεδόν όλο τον κώδικα στο scirouter), επομένως παραθέτουμε ποιοτική/θεωρητική σύγκριση για το τι θα αναμέναμε στην πράξη.

Β. Μετρήσεις και Διαγράμματα



Γ. Συμπεράσματα

Για το σενάριο 512×512 με 64 διεργασίες, τα μετρούμενα αποτελέσματα είναι:

- Total time: 40.516 s
- Computation time: 3.192 s
- Convergence time: 1.261 s

1) Η εκτέλεση είναι έντονα communication/synchronization-bound

Παρατηρούμε ότι ο χρόνος υπολογισμών αποτελεί μικρό μέρος του συνολικού χρόνου: 3.192 / 40.516 ≈ 7.9% του total. Αντίστοιχα, ο χρόνος ελέγχου σύγκλισης είναι: 1.261 / 40.516 ≈ 3.1% του total. Το υπόλοιπο (~89%) αντιστοιχεί σε κόστη που δεν περιλαμβάνονται στον καθαρό υπολογισμό και στον μετρούμενο έλεγχο σύγκλισης: κυρίως halo exchanges (Isend/Irecv/Waitall), αναμονές/ανισορροπία (όλοι συγχρονίζονται ουσιαστικά ανά iteration), και γενικά MPI overheads ανά επανάληψη.

Αυτό είναι αναμενόμενο από τη θεωρία: όταν το local υποπλέγμα ανά διεργασία γίνεται μικρό (εδώ 512×512 / 64 ≈ 64×64 ανά process), ο λόγος επιφάνειας προς όγκο αυξάνει. Δηλαδή, τα δεδομένα που πρέπει να ανταλλαχθούν (περίμετρος του block) δεν μειώνονται τόσο γρήγορα όσο ο υπολογισμός (εμβαδό του block). Συνεπώς, το κόστος επικοινωνίας γίνεται συγκρίσιμο ή και κυρίαρχο.

2) Ο έλεγχος σύγκλισης προσθέτει παγκόσμιο συγχρονισμό (global reduction)

Παρότι το convergence time (1.261 s) δεν είναι το μεγαλύτερο κομμάτι, παραμένει μη αμελητέο, επειδή:

- ο έλεγχος απαιτεί μια καθολική απόφαση (όλες οι διεργασίες πρέπει να συμφωνήσουν ότι έχει επιτευχθεί σύγκλιση),
- άρα χρησιμοποιείται συλλογική επικοινωνία (π.χ. Allreduce) που επιβάλλει συγχρονισμό και κόστος latency που τυπικά αυξάνει με την κλίμακα (συχνά ~logP).

Πρακτικά, κάθε convergence check λειτουργεί σαν σημείο όπου η πιο αργή διεργασία καθορίζει την πρόοδο όλων (critical path). Αυτό είναι βασική αρχή στη θεωρία: είναι αποδοτική μέθοδος για μικρά μηνύματα, αλλά έχει ένα αναπόφευκτο (και συχνά μεγάλο) synchronization component.

3) Γιατί με 64 διεργασίες ο Jacobi μπορεί να έχει μεγάλο total time παρότι ο compute πυρήνας είναι μικρός

Η Jacobi είναι μεν εύκολα παραλληλοποιήσιμη, αλλά για να προχωρήσει κάθε iteration χρειάζεται ενημερωμένα halo από τους γείτονες. Αυτό σημαίνει ότι:

- υπάρχει επικοινωνία σε κάθε iteration (γειτονική ανταλλαγή γραμμών/στηλών),
- το κόστος της επικοινωνίας καθορίζεται έντονα από latency και όχι μόνο από bandwidth, ειδικά όταν τα blocks είναι μικρά,
- ενώ ταυτόχρονα ο υπολογισμός ανά διεργασία είναι μικρός και δεν κρύβει τον χρόνο επικοινωνίας.

Αποτέλεσμα: όσο μεγαλώνει το P για σταθερό πρόβλημα (strong scaling), το compute μειώνεται γρήγορα, αλλά η επικοινωνία/συγχρονισμοί γίνονται το bottleneck — κλασικό όριο strong scaling (Amdahl-like συμπεριφορά, αλλά εδώ κυρίως λόγω surface/volume και latency).

4) Θεωρητική σύγκριση με τις bonus μεθόδους (χωρίς υλοποίηση)

Παρότι δεν υλοποιήσαμε SOR/Red–Black SOR, η θεωρία υποδεικνύει τα εξής:

- Jacobi: πλήρως παράλληλος ανά iteration, αλλά συνήθως πιο αργή σύγκλιση (περισσότερες επαναλήψεις → περισσότερα halo exchanges και περισσότερα convergence checks).
- Gauss–Seidel SOR: τείνει να συγκλίνει σε λιγότερες επαναλήψεις (άρα πιθανώς μικρότερο συνολικό communication volume), αλλά έχει ισχυρότερες εξαρτήσεις ενημέρωσης που δυσκολεύουν τον καθαρό παραλληλισμό.
- Red–Black SOR: προσφέρει πρακτικό συμβιβασμό, γιατί διατηρεί ταχύτερη σύγκλιση από Jacobi και επιτρέπει παράλληλη ενημέρωση ανά χρώμα (red/black), άρα είναι συχνά πιο κατάλληλο για παραλληλοποίηση από το κλασικό Gauss–Seidel.

Με βάση αυτό, αν οι bonus μέθοδοι ολοκληρώνονταν, θα αναμέναμε μικρότερο total time κυρίως επειδή θα απαιτούνταν λιγότερες επαναλήψεις (άρα λιγότερα halo exchanges και λιγότερα global convergence checks) — δηλαδή θα μειωνόταν το κυρίαρχο κομμάτι του κόστους.

Τελικό συμπέρασμα

Το πείραμα με 512×512 και 64 διεργασίες δείχνει καθαρά ότι, με ενεργό convergence check, η MPI Jacobi γίνεται κυρίως communication/synchronization-bound: ο καθαρός υπολογισμός είναι μικρό ποσοστό του total, ενώ το μεγαλύτερο μέρος προκύπτει από halo exchanges και τα αναπόφευκτα σημεία συγχρονισμού (γειτονική ανταλλαγή ανά iteration και περιοδικές συλλογικές επικοινωνίες για σύγκλιση). Αυτό συνάδει πλήρως με τη θεωρία των εφαρμογών σε κατανεμημένη μνήμη και εξηγεί γιατί οι μέθοδοι με ταχύτερη σύγκλιση (SOR, ιδιαίτερα Red–Black) είναι σημαντικές: μειώνουν τον αριθμό επαναλήψεων, άρα περιορίζουν το επικοινωνιακό κόστος που εδώ κυριαρχεί.

▪ 4.2.2 – Μετρήσεις χωρίς Έλεγχο Σύγκλισης

Α. Εισαγωγή

Στην υποενότητα αυτή θα μετρήσουμε την απόδοση της MPI υλοποίησης του 2D Jacobi για τη διάδοση θερμότητας, χωρίς κριτήριο σύγκλισης. Σύμφωνα με την εκφώνηση, ο αλγόριθμος εκτελείται για σταθερό πλήθος επαναλήψεων (T=256), ώστε όλες οι εκτελέσεις να έχουν ίδιο αριθμό βημάτων και οι συγκρίσεις να μην επηρεάζονται από διαφορετικό χρόνο τερματισμού.

Η απενεργοποίηση του convergence check έχει δύο στόχους:

- Απομονώνει το κόστος του βασικού υπολογισμού και των επικοινωνιών ανά επανάληψη, χωρίς πρόσθετο (καθολικό) συγχρονισμό τύπου Allreduce.
- Επιτρέπει καθαρή αξιολόγηση strong scaling: για κάθε μέγεθος πλέγματος (2048×2048, 4096×4096, 6144×6144) αυξάνουμε τον αριθμό MPI διεργασιών P και μετράμε τον χρόνο.

Μετρήσεις/Μετρικές (όπως ζητούνται):

- Speedup S(P) = Tseq / TP, όπου Tseq ο χρόνος της σειριακής εκτέλεσης για το ίδιο μέγεθος πλέγματος και TP ο χρόνος με P διεργασίες.
- Total time: χρόνος του επαναληπτικού τμήματος, λαμβάνοντας πρακτικά τον χειρότερο (critical path), αφού η συνολική πρόοδος καθορίζεται από την πιο αργή διεργασία.
- Computation time: χρόνος αποκλειστικά των ενημερώσεων χωρίς να μετρά την επικοινωνία/αναμονές.

Θεωρητικά, σε κάθε iteration κάθε διεργασία:

1. ανταλλάσσει halo με τους 4 γείτονες (πάνω/κάτω/αριστερά/δεξιά) και
2. υπολογίζει τοπικά το εσωτερικό της block.

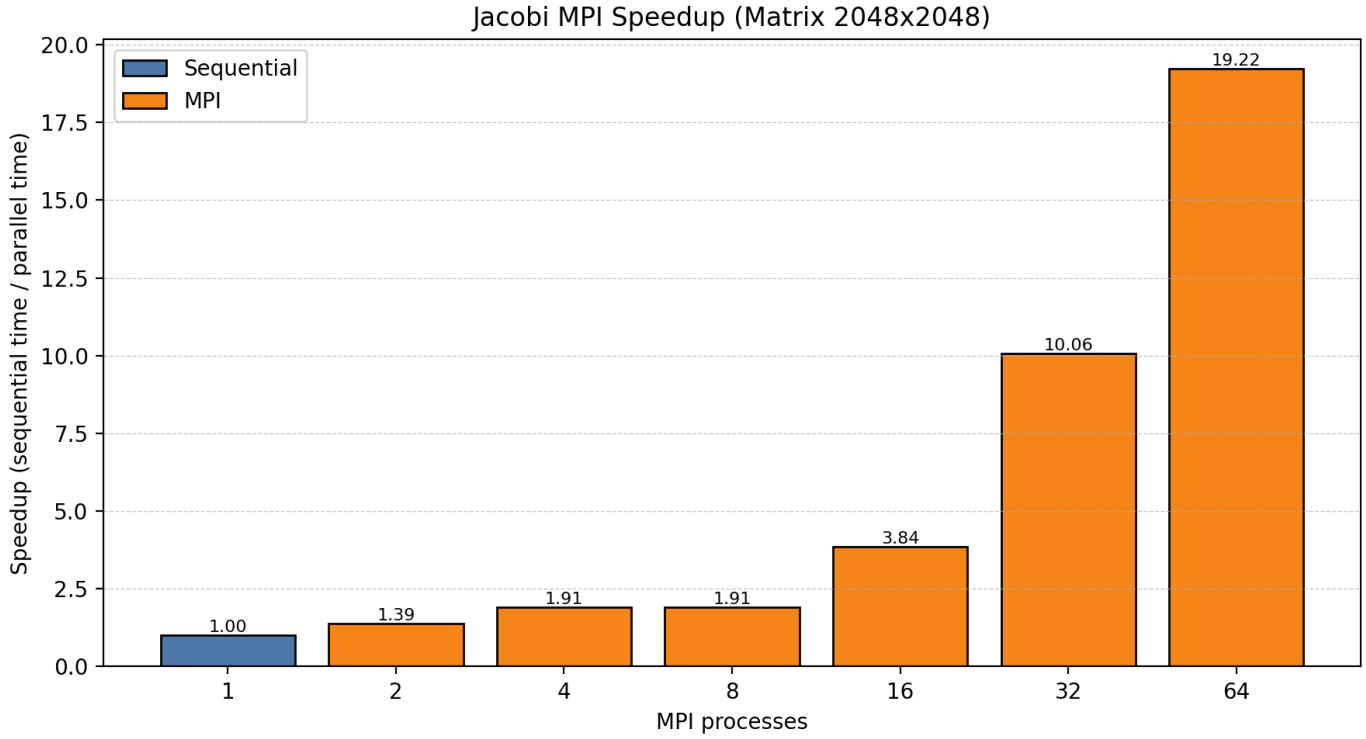
Άρα ο συνολικός χρόνος ανά iteration προσεγγίζεται από:

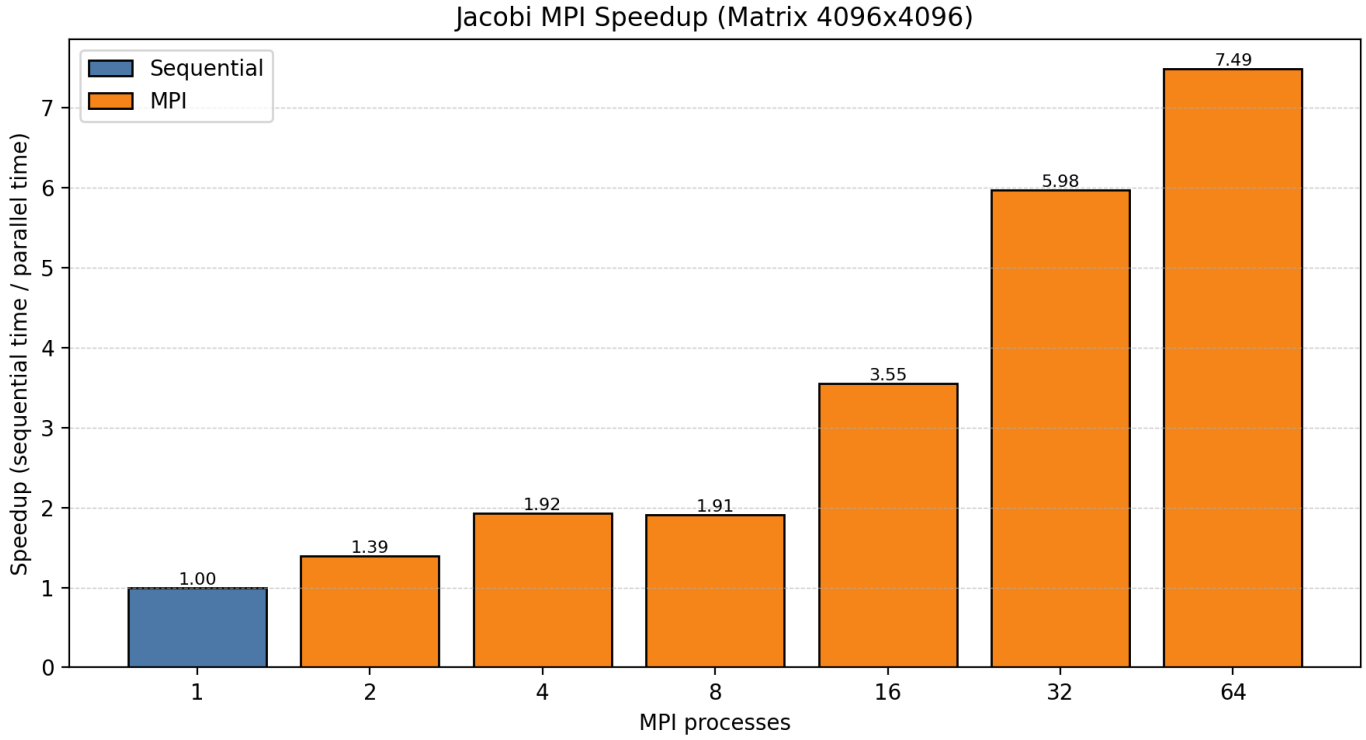
$T_{iter} \approx T_{comp} + T_{comm}$,

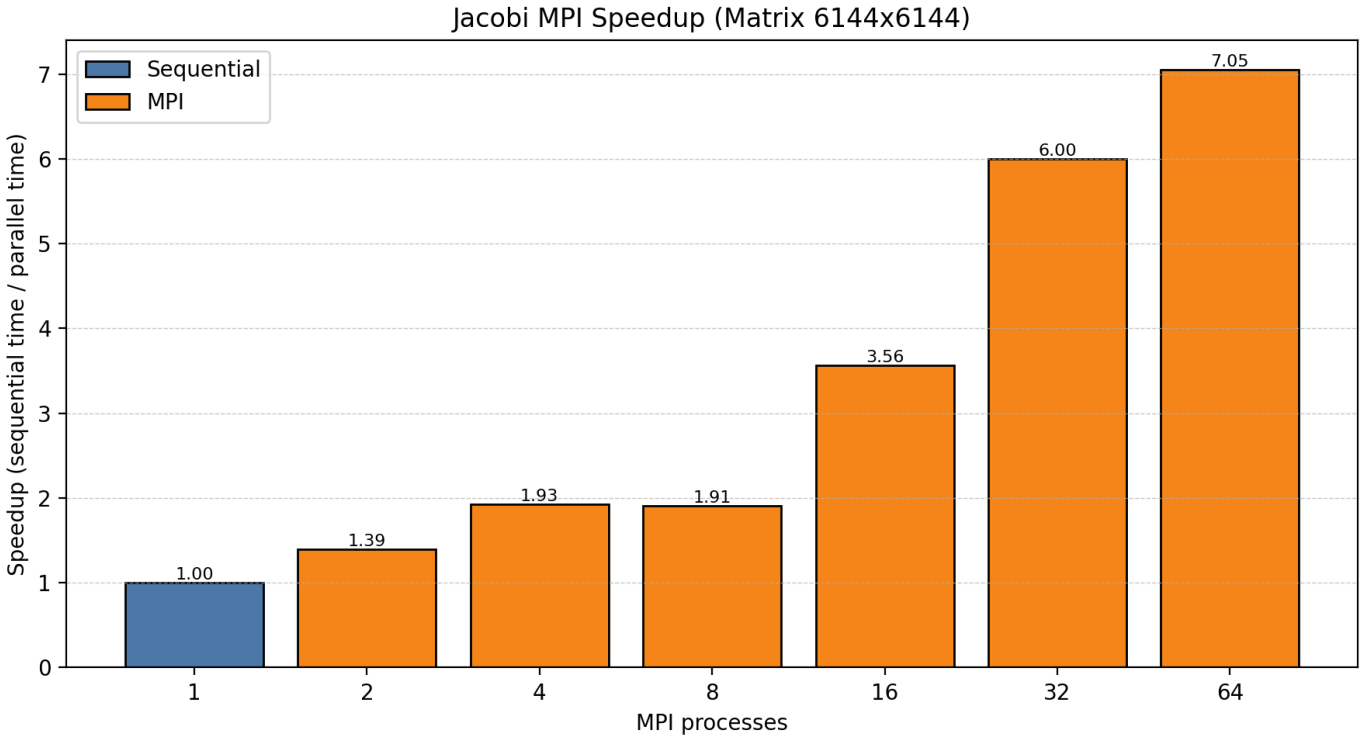
όπου T_{comm} περιλαμβάνει latency + bandwidth κόστος ανταλλαγών και τυχόν αναμονές (Waitall). Στην strong scaling περίπτωση (σταθερό πρόβλημα, αυξανόμενο P), το T_{comp} μειώνεται περίπου ως $O(1/P)$, ενώ το T_{comm} δεν μειώνεται αντίστοιχα και συχνά γίνεται το bottleneck λόγω αυξημένου surface/volume ratio και latency-dominated μικρών μηνυμάτων.

Β. Μετρήσεις και Διαγράμματα

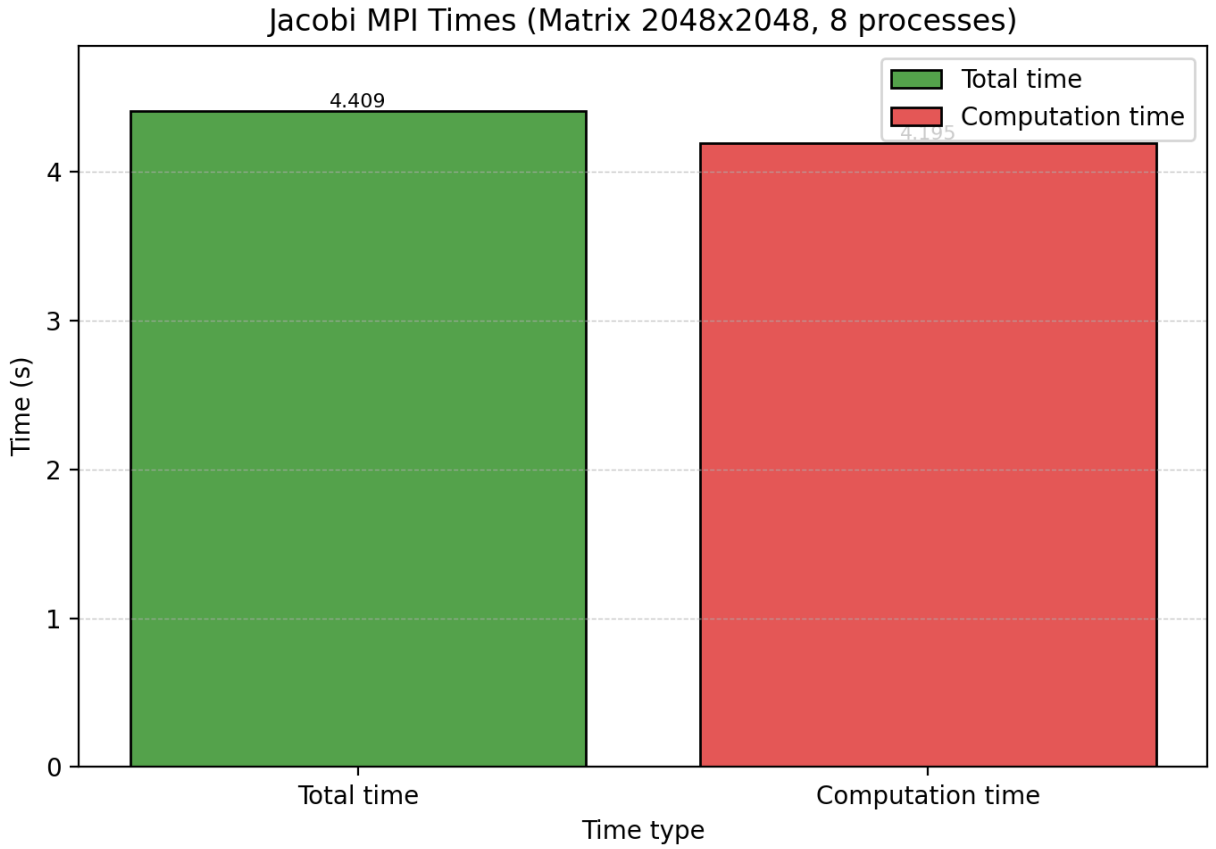
Διαγράμματα Επιτάχυνσης

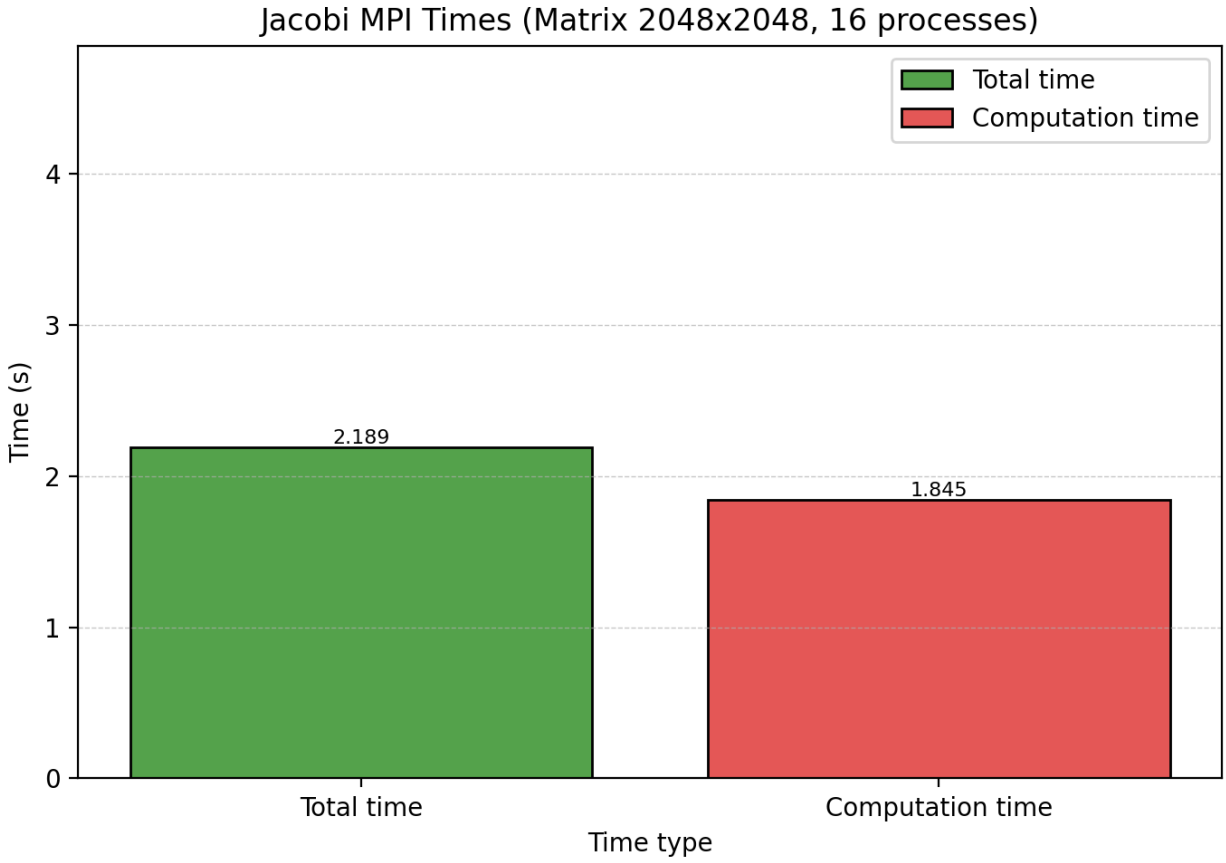


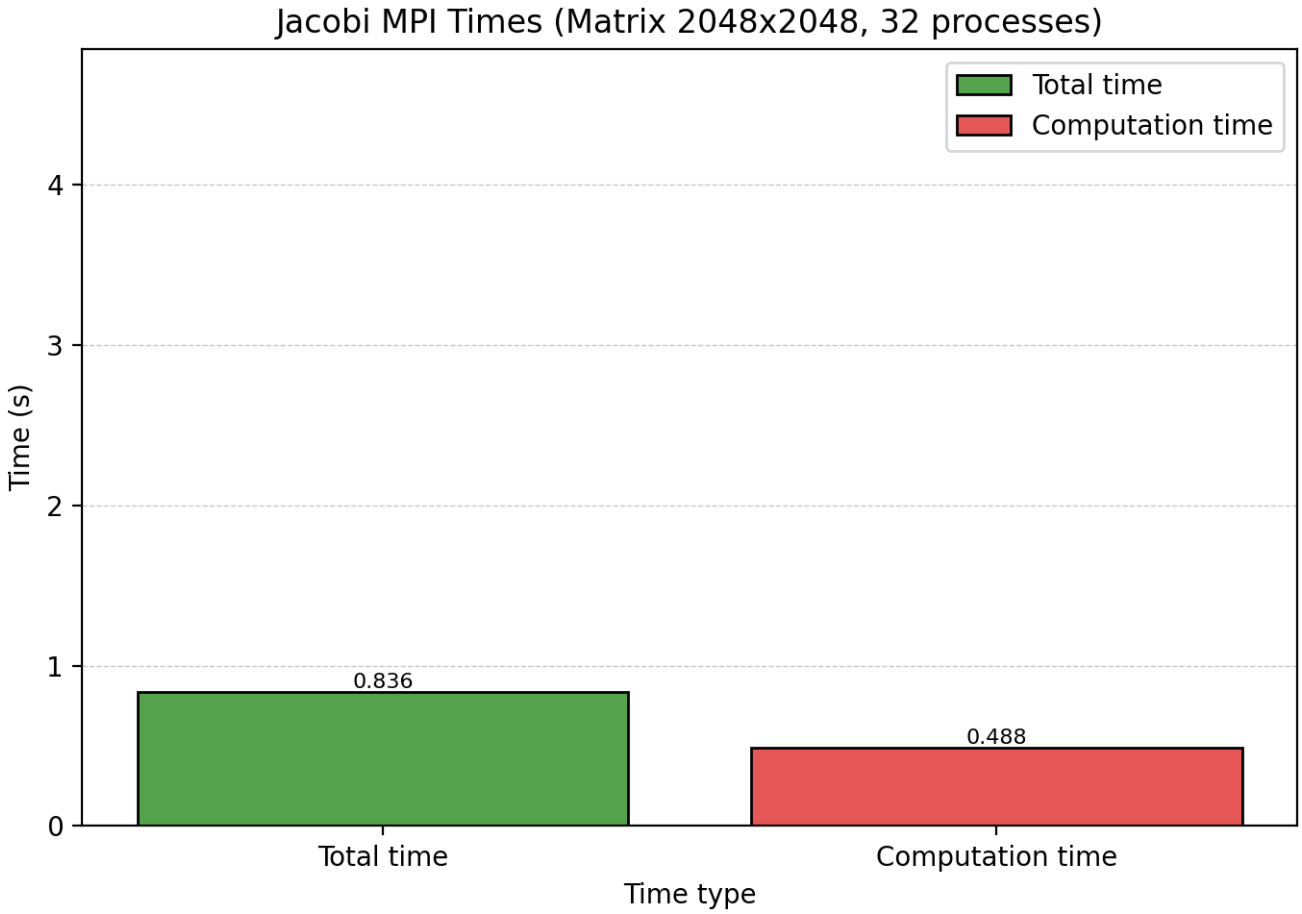


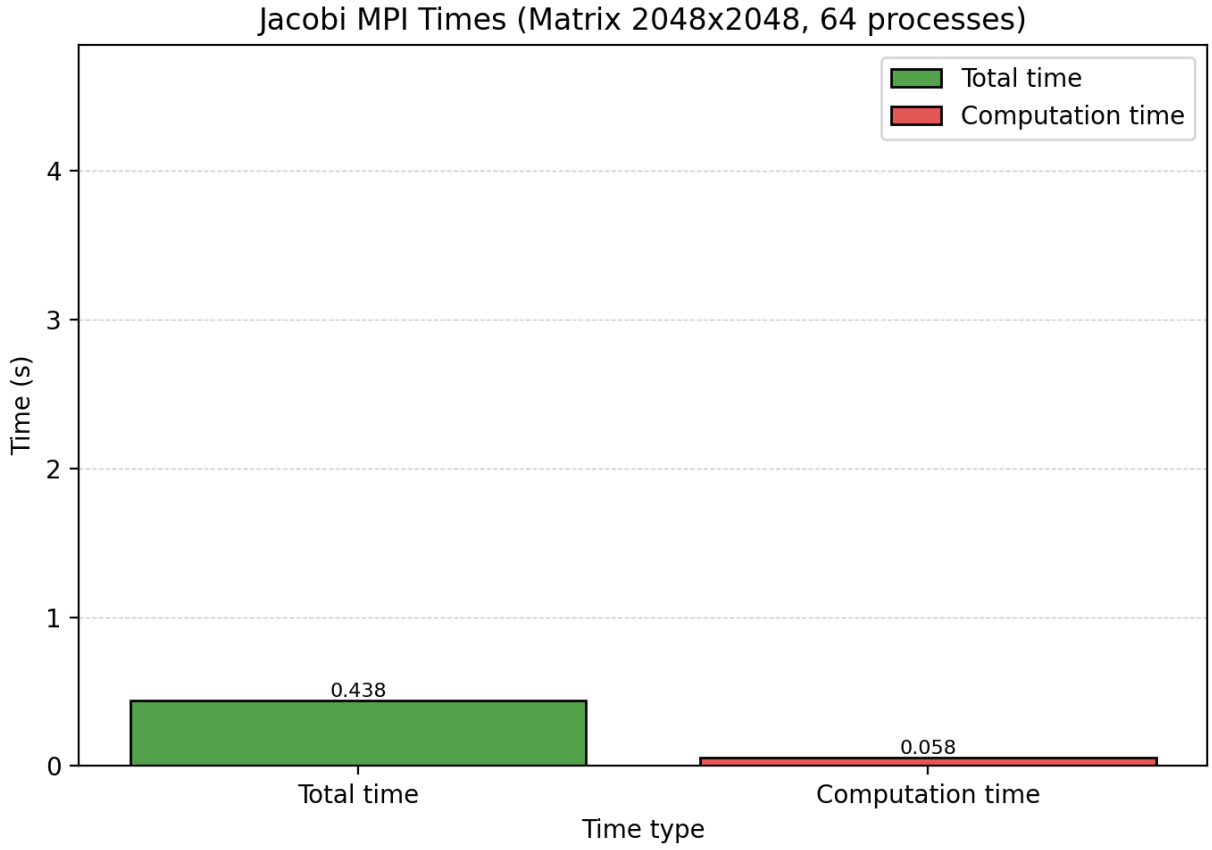

Jacobi MPI Speedup (Matrix 6144x6144)
Sequential
MPI
Speedup (sequential time / parallel time)
MPI processes
1.00
1.39
1.93
1.91
3.56
6.00
7.05
0
1
2
3
4
5
6
7
1
2
4
8
16
32
64

Διαγράμματα χρόνου – Πίνακας 2048x2048

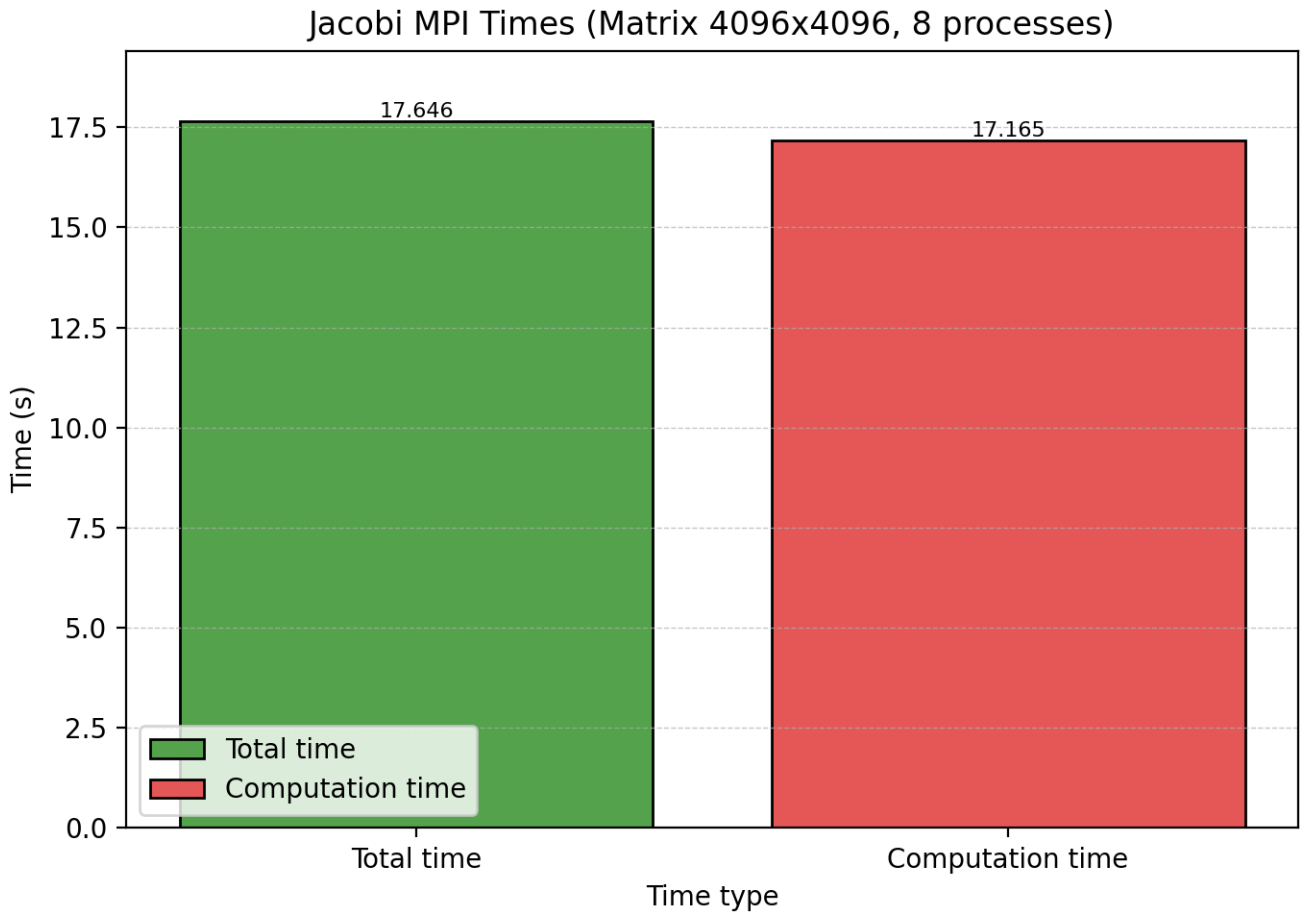


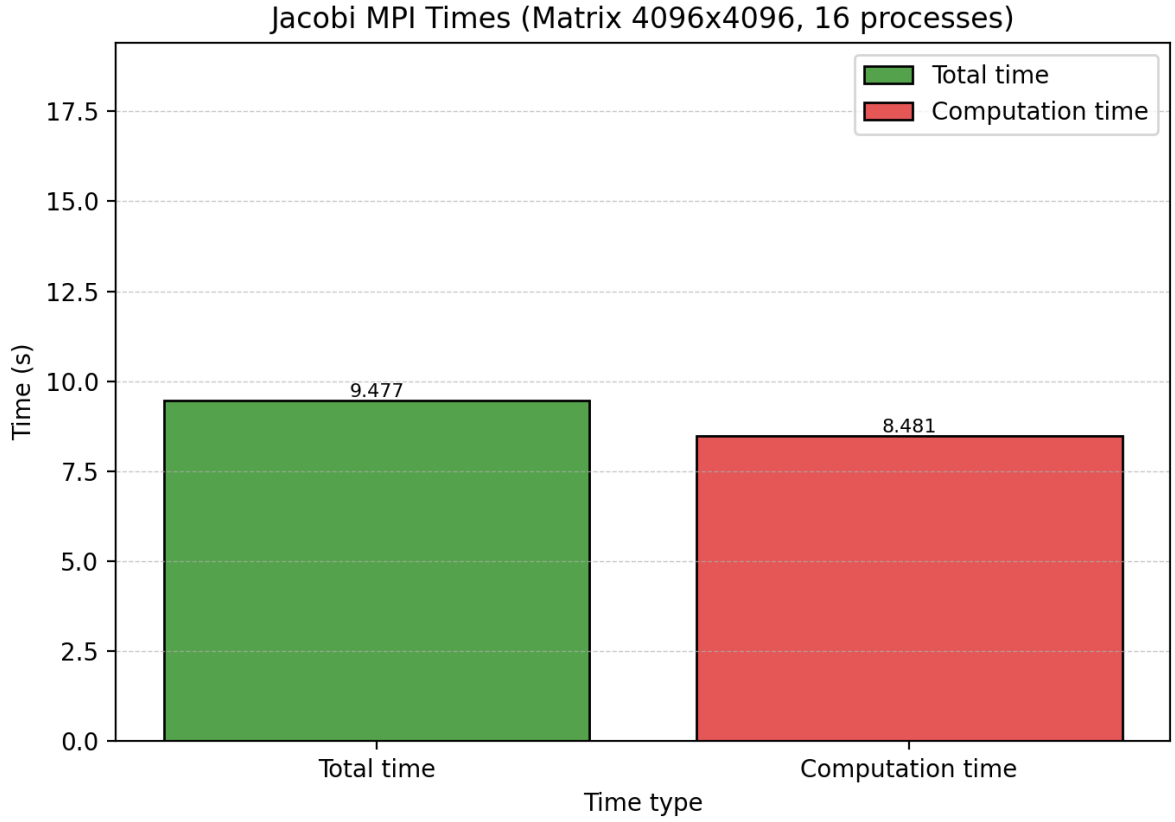


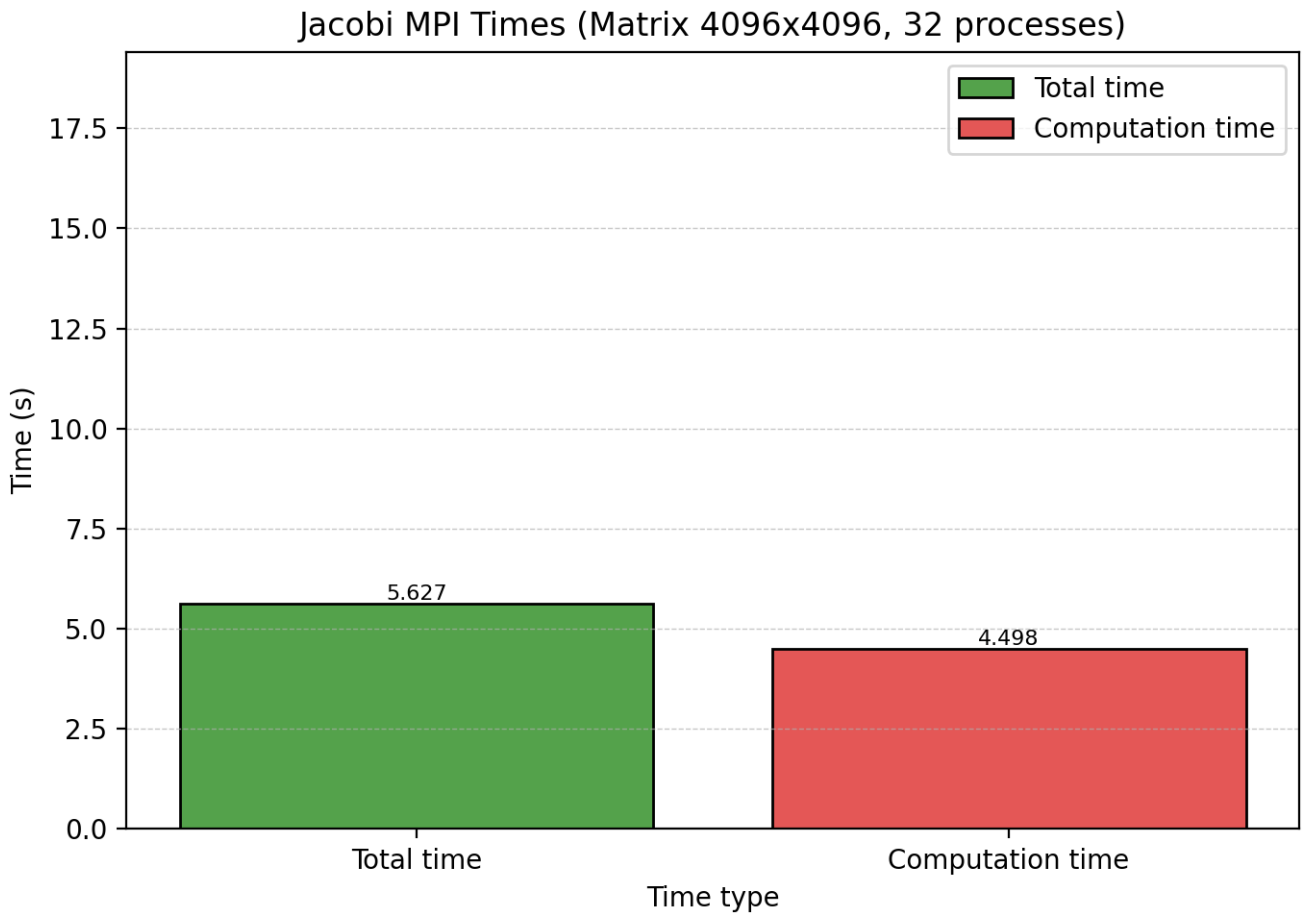

Jacobi MPI Times (Matrix 2048x2048, 32 processes)
Total time
Computation time
Time (s)
0
1
2
3
4
0.836
0.488
Total time
Computation time
Time type

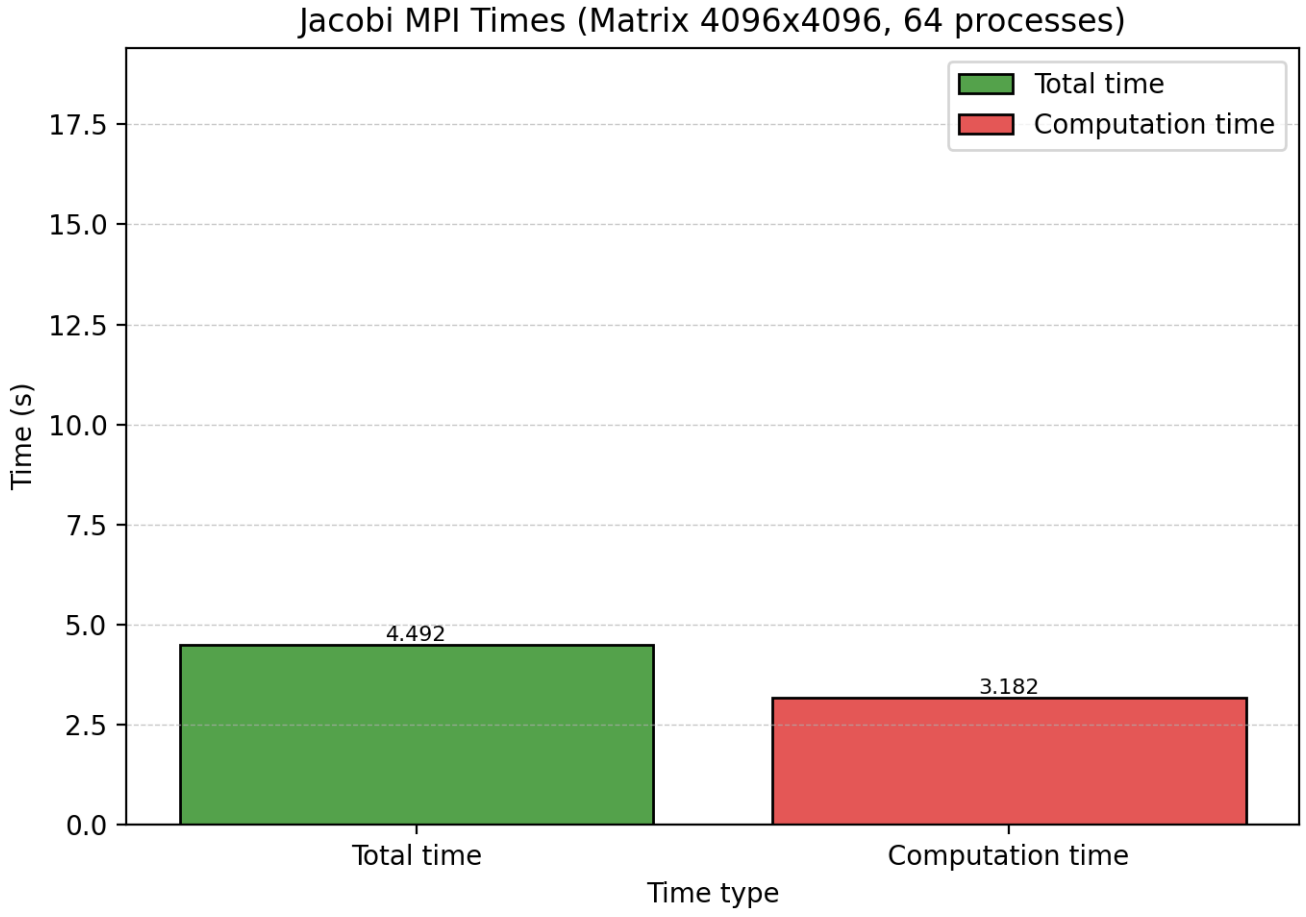

Jacobi MPI Times (Matrix 2048x2048, 64 processes)
Total time
Computation time
Time (s)
0
1
2
3
4
0.438
0.058
Total time
Computation time
Time type

Διαγράμματα χρόνου – Πίνακας 4096x4096

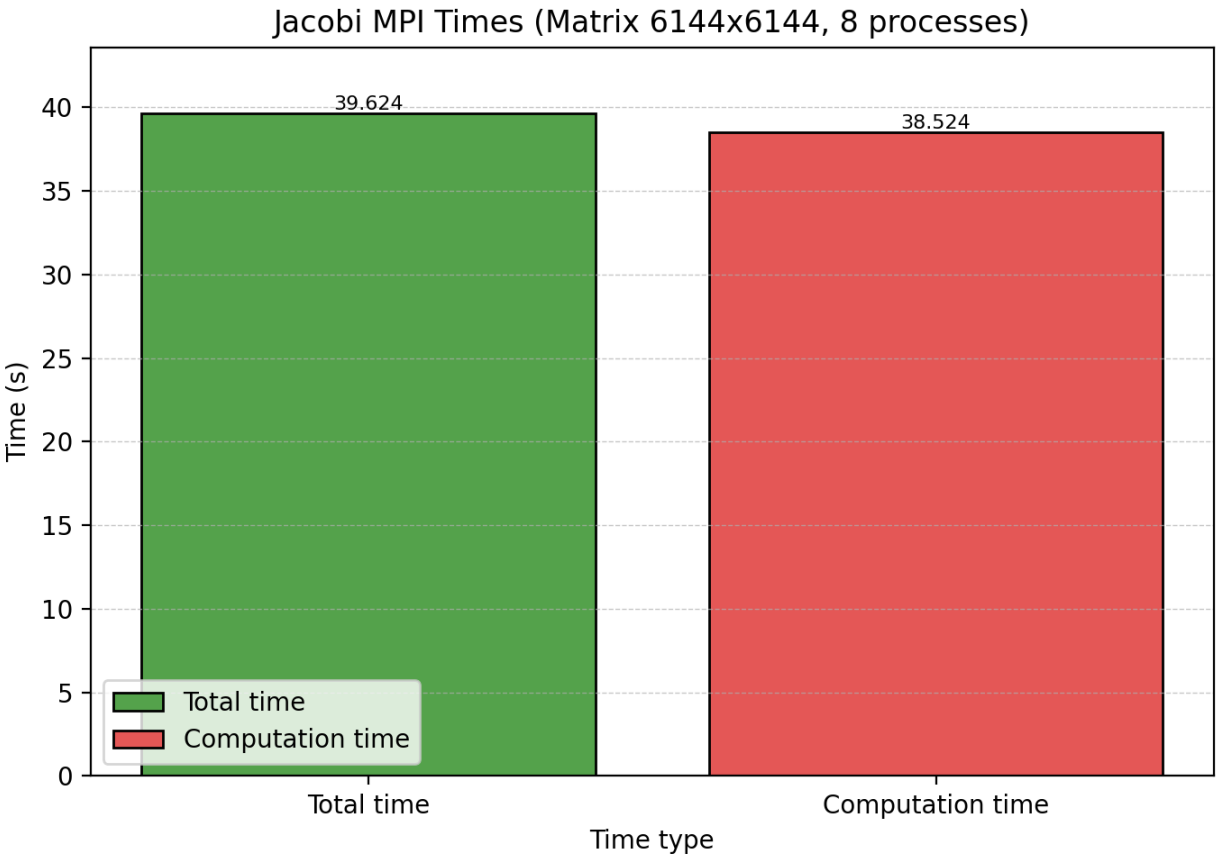


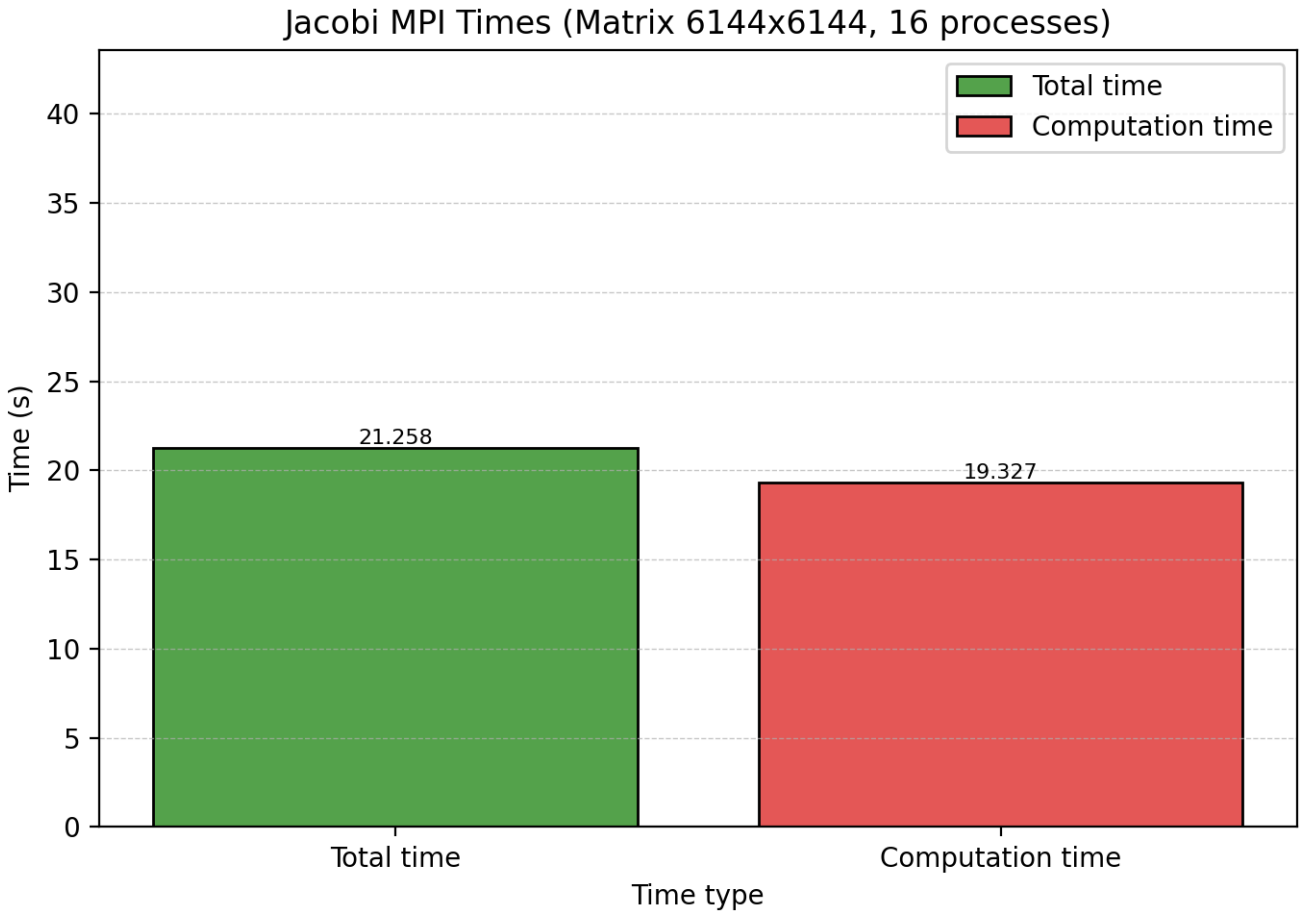


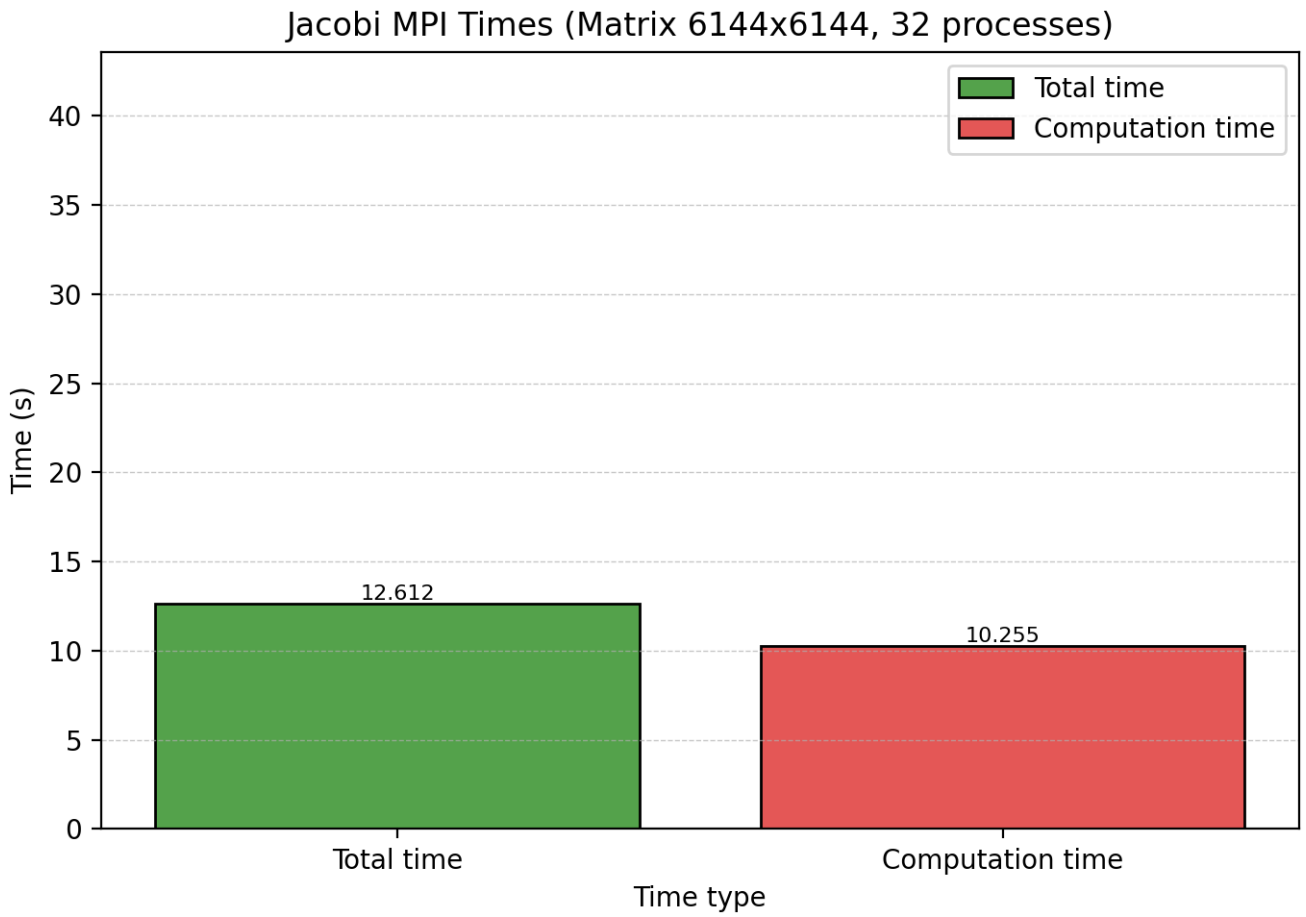

Jacobi MPI Times (Matrix 4096x4096, 32 processes)
Total time
Computation time
Time (s)
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
5.627
4.498
Total time
Computation time
Time type

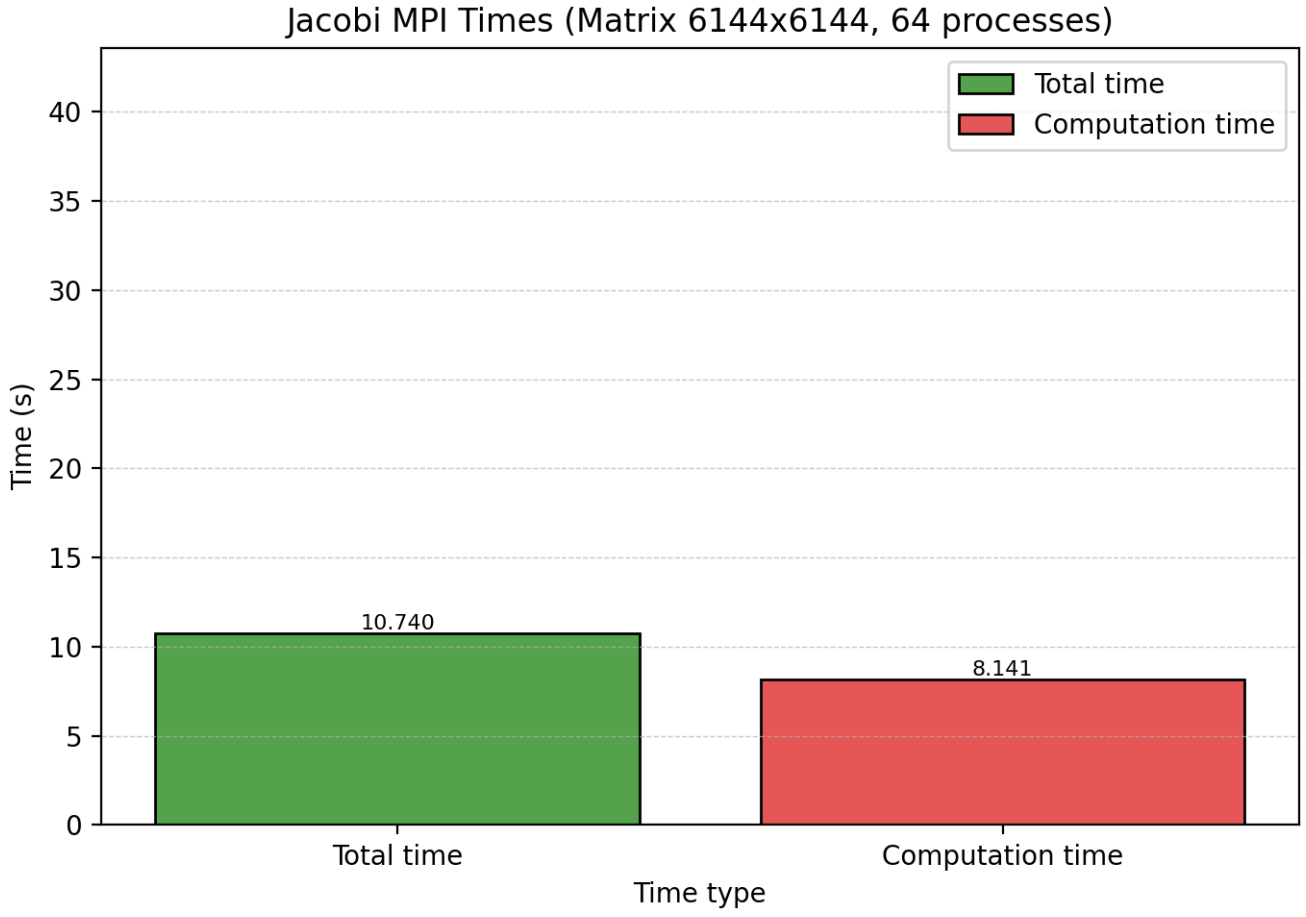

Jacobi MPI Times (Matrix 4096x4096, 64 processes)
Total time
Computation time
Time (s)
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
4.492
3.182
Total time
Computation time
Time type

Διαγράμματα χρόνου – Πίνακας 4096x4096






Jacobi MPI Times (Matrix 6144x6144, 32 processes)
Total time
Computation time
Time (s)
0
5
10
15
20
25
30
35
40
12.612
10.255
Total time
Computation time
Time type


Jacobi MPI Times (Matrix 6144x6144, 64 processes)
Total time
Computation time
Time (s)
0
5
10
15
20
25
30
35
40
10.740
8.141
Total time
Computation time
Time type

Γ. Συμπεράσματα

Παρακάτω σχολιάζουμε συνολικά τα speedups και, με βάση τα διαγράμματα Total vs Computation time, τεκμηριώνουμε πού ξοδεύεται ο χρόνος και γιατί εμφανίζεται κορεσμός.

1) Speedup – Strong scaling τάση ανά μέγεθος πλέγματος

(α) Matrix 2048×2048

Speedup: 1.39 (P=2), 1.91 (P=4), 1.91 (P=8), 3.84 (P=16), 10.06 (P=32), 19.22 (P=64).

Παρατηρείται μικρό κέρδος μέχρι P=16, ενώ από P=32 και P=64 εμφανίζεται απότομη επιτάχυνση, με superlinear συμπεριφορά σε σχέση με τα μικρότερα P (δηλ. το speedup αυξάνει δυσανάλογα όταν διπλασιάζουμε τις διεργασίες). Αυτό μπορεί να εξηγηθεί από memory/cache effects: όσο αυξάνει το P, το τοπικό υποπλέγμα ανά διεργασία μικραίνει και χωράει καλύτερα σε caches (ή/και σε τοπικές NUMA περιοχές), μειώνοντας σημαντικά cache misses και πιέσεις στο memory bandwidth. Έτσι, δεν κερδίζουμε μόνο από τον καθαρό διαμοιρασμό του υπολογισμού, αλλά και από το ότι κάθε διεργασία εκτελεί έναν πιο cache-friendly υπολογισμό με χαμηλότερο κόστος πρόσβασης στη μνήμη, άρα ο χρόνος πέφτει περισσότερο από όσο θα περίμενε κανείς με απλή γραμμική κλιμάκωση. Παρόλα αυτά, το τελικό speedup 19.22 σε P=64 απέχει από το ιδανικό 64 (άρα το efficiency παραμένει χαμηλό), κάτι που δείχνει ότι το κόστος επικοινωνίας/συγχρονισμών και τα σταθερά overheads (halo exchanges, wait, πιθανό contention στο interconnect) εξακολουθούν να περιορίζουν τη συνολική κλιμάκωση, ειδικά σε τόσο μικρό πρόβλημα.

(β) Matrix 4096×4096

Speedup: 1.39 (P=2), 1.92 (P=4), 1.91 (P=8), 3.55 (P=16), 5.98 (P=32), 7.49 (P=64).

Στο μέγεθος 4096×4096 το speedup αυξάνει πιο ομαλά σε σχέση με το 2048×2048, δηλαδή βλέπουμε την αναμενόμενη τάση χωρίς απότομα άλματα: όσο αυξάνονται οι διεργασίες, ο χρόνος μειώνεται και το speedup μεγαλώνει. Αυτό είναι συμβατό με το ότι το πρόβλημα έχει πλέον αρκετό υπολογιστικό φορτίο ώστε το κόστος ανά iteration (υπολογισμός στο εσωτερικό) να παραμένει σημαντικό και να αποσβένει τα επικοινωνιακά overheads στα μικρότερα P.

Ωστόσο, μετά το P=32 εμφανίζεται σαφής κορεσμός: από 32 σε 64 διεργασίες κερδίζουμε σχετικά λίγο (5.98 → 7.49). Η συμπεριφορά αυτή εξηγείται από τη θεωρία: καθώς αυξάνεται το P, το τοπικό υποπλέγμα κάθε διεργασίας μικραίνει, άρα ο λόγος οριακών κελιών προς εσωτερικά αυξάνει και η σχετική σημασία του halo exchange μεγαλώνει. Επιπλέον, σε υψηλά P το κόστος της επικοινωνίας γίνεται όλο και πιο latency-dominated (πολλά μικρά μηνύματα/συχνές ανταλλαγές) και οι συγχρονισμοί (MPI_Waitall, implicit sync ανά iteration) επιβάλλουν έναν bulk-synchronous ρυθμό, όπου η συνολική πρόοδος περιορίζεται από τις καθυστερήσεις των πιο αργών διεργασιών και από πιθανό contention στο interconnect. Άρα, παρότι συνεχίζουμε να μειώνουμε τον καθαρό υπολογισμό ανά διεργασία, το μη-κλιμακούμενο μέρος (επικοινωνία + αναμονές) αρχίζει να κυριαρχεί και η επιπλέον παραλληλία μετά το P=32 αποδίδει φθίνον κέρδος. Γενικά, απέχουμε πολύ από την ιδανική (γραμμική) κλιμάκωση.

(γ) Matrix 6144×6144

Speedup: 1.39 (P=2), 1.93 (P=4), 1.91 (P=8), 3.56 (P=16), 6.00 (P=32), 7.05 (P=64).

Στο μεγαλύτερο πλέγμα 6144×6144 παρατηρούμε παρόμοια εικόνα με το 4096×4096: το speedup αυξάνει αρχικά με τρόπο αναμενόμενο (strong scaling), αλλά μετά το P=32 εμφανίζεται καθαρός κορεσμός και στο P=64 καταλήγουμε σε τελικό speedup περίπου 7× (6.00 → 7.05 από 32 σε 64). Η βασική ερμηνεία είναι ότι, παρότι το μεγαλύτερο πλέγμα είναι πιο compute-intense (άρα ευνοεί γενικά τον παραλληλισμό στα μικρότερα P), σε υψηλές τιμές P το τοπικό block ανά διεργασία μικραίνει αρκετά ώστε να αυξηθεί ο λόγος επιφάνειας/όγκου και να ενισχυθεί η επίδραση της επικοινωνίας ορίων.

Συγκεκριμένα, η επικοινωνία halo ανά iteration παραμένει υποχρεωτική και συνοδεύεται από synchronization (Waitall), οπότε όσο μειώνεται ο υπολογισμός ανά διεργασία, το σταθερό/μη-κλιμακούμενο κόστος (latency των μικρών μηνυμάτων, κόστος δικτύου, και αναμονές λόγω slowest rank) αρχίζει να κυριαρχεί. Έτσι, η μετάβαση από 32 σε 64 διεργασίες δεν μπορεί να διπλασιάσει την απόδοση: το compute μειώνεται, αλλά το communication+sync δεν ακολουθεί, οδηγώντας σε φθίνον κέρδος. Με άλλα λόγια, στο 6144×6144 ο παραλληλισμός είναι καλύτερος από άποψη διαθέσιμου υπολογιστικού έργου, όμως στο πολύ υψηλό P μπαίνουμε στην περιοχή όπου το πρόβλημα περιορίζεται κυρίως από επικοινωνία και συγχρονισμούς, άρα εμφανίζεται ο ίδιος strong-scaling κορεσμός.

Συμπέρασμα speedup

Καθώς αυξάνεται το μέγεθος του πλέγματος, το πρόβλημα γίνεται συνολικά πιο compute-intense: ανά διεργασία αυξάνεται ο αριθμός αριθμητικών πράξεων που αντιστοιχεί σε κάθε ανταλλαγή ορίων (halo exchange), άρα βελτιώνεται ο λόγος υπολογισμού προς επικοινωνία. Αυτό θεωρητικά ευνοεί την κλιμάκωση, επειδή τα κόστη επικοινωνίας (latency + bandwidth) και οι συγχρονισμοί δεν μειώνονται με τον ίδιο ρυθμό όπως ο υπολογισμός όταν αυξάνει το P, ενώ σε μεγαλύτερα πλέγματα ο υπολογισμός παραμένει αρκετά βαρύς ώστε να αποσβένονται καλύτερα τα σταθερά επικοινωνιακά κόστη ανά iteration.

Συμπέρασμα από τα διαγράμματα: και στα τρία μεγέθη (2048×2048, 4096×4096, 6144×6144) παρατηρούμε strong scaling μέχρι ένα σημείο, δηλαδή η αύξηση των διεργασιών μειώνει τον χρόνο εκτέλεσης. Ωστόσο η κλιμάκωση δεν είναι γραμμική, κάτι αναμενόμενο από τη θεωρία strong scaling (Amdahl + communication overhead): υπάρχει ένα μη-κλιμακούμενο μέρος που σχετίζεται με ανταλλαγές ορίων, latency-dominated μηνύματα και αναμονές/συγχρονισμούς (π.χ. Waitall ανά iteration), το οποίο αποκτά ολοένα μεγαλύτερη σχετική βαρύτητα όσο το per-process υπολογιστικό φορτίο μειώνεται.

Για τα μεγαλύτερα πλέγματα (4096×4096 και 6144×6144) εμφανίζεται καθαρός κορεσμός από P≥32 και μετά (μικρό πρόσθετο κέρδος από 32→64). Η εικόνα αυτή ταιριάζει με τη θεωρία: όταν το τοπικό υποπλέγμα ανά διεργασία μικραίνει, αυξάνεται ο λόγος περιμέτρου προς εμβαδόν, άρα η σχετική επίδραση των halo exchanges και των συγχρονισμών μεγαλώνει και περιορίζει το επιπλέον όφελος της παραλληλίας. Με άλλα λόγια, ενώ ο υπολογισμός συνεχίζει να μειώνεται, το communication+sync δεν ακολουθεί, οδηγώντας σε φθίνον κέρδος.

Στο μικρότερο πλέγμα (2048×2048) παρατηρείται απότομη βελτίωση στα P=32 και P=64, με superlinear συμπεριφορά σε σχέση με τα μικρότερα P. Αυτό είναι συμβατό με τα cache/memory effects: το τοπικό block ανά διεργασία γίνεται αρκετά μικρό ώστε να χωράει καλύτερα σε caches και/ή να βελτιώνεται η NUMA locality, μειώνοντας δυσανάλογα τα cache misses και τις αργές προσπελάσεις στη μνήμη. Έτσι δεν κερδίζουμε μόνο από τον καθαρό επιμερισμό του compute, αλλά και από πιο αποδοτική εκτέλεση του ίδιου υπολογισμού λόγω καλύτερης ιεραρχίας μνήμης. Παρόλα αυτά, το speedup παραμένει αρκετά κάτω από το ιδανικό P σε υψηλά P, κάτι που δείχνει ότι τα επικοινωνιακά/συγχρονιστικά κόστη (halo exchange, latency, wait) εξακολουθούν να θέτουν το τελικό όριο στην κλιμάκωση.

2) Total vs Computation time

Από τα time-plots μπορούμε να εκτιμήσουμε το communication + synchronization overhead ως:

Toverhead = Ttotal − Tcomp.

(α) Matrix 2048×2048

- P=8: Ttotal=4.409s, Tcomp=4.195s → Toverhead=0.214s (~4.9% του total)
- P=16: Ttotal=2.189s, Tcomp=1.845s → Toverhead=0.344s (~15.7%)
- P=32: Ttotal=0.836s, Tcomp=0.488s → Toverhead=0.348s (~41.6%)
- P=64: Ttotal=0.438s, Tcomp=0.058s → Toverhead=0.380s (~86.8%)

Εδώ φαίνεται καθαρά η strong-scaling συμπεριφορά: όσο αυξάνει το P, ο υπολογισμός ανά διεργασία μειώνεται έντονα, ενώ το κόστος των halo exchanges και των αναμονών (Waitall) δεν μειώνεται με τον ίδιο ρυθμό. Παράλληλα, στο 2048×2048 εμφανίζεται και το φαινόμενο superlinear βελτίωσης κυρίως στο άλμα 16→32→64, που συμβαδίζει με την παρατήρηση ότι το Tcomp πέφτει δυσανάλογα (ιδίως στο P=64: 0.058s). Η πιο πιθανή αιτία είναι τα cache/memory effects: με αρκετά μικρό local block, ο υπολογισμός γίνεται πολύ πιο cache-friendly (λιγότερα cache misses, καλύτερη temporal/spatial locality, πιθανώς καλύτερη NUMA τοπικότητα), οπότε ο ίδιος αριθμός ενημερώσεων εκτελείται ταχύτερα από ό,τι θα προέβλεπε ο απλός επιμερισμός 1/P.

Ωστόσο, το ίδιο διάγραμμα δείχνει και το τελικό όριο της strong scaling κλιμάκωσης: ενώ το Tcomp σχεδόν μηδενίζεται στο P=64, το Toverhead παραμένει ~σταθερό (≈0.35–0.38s από P=16 και μετά) και τελικά κυριαρχεί στο συνολικό χρόνο (≈86.8% στο P=64). Άρα, στο μεγάλο P το σύστημα γίνεται καθαρά communication/synchronization-bound: η επίδοση δεν περιορίζεται πλέον από τον υπολογιστικό πυρήνα, αλλά από latency, ανταλλαγές ορίων και τον συγχρονισμό που επιβάλλει το Waitall ανά iteration.

(β) Matrix 4096×4096

- P=8: 17.646 − 17.165 = 0.481s (~2.7%)
- P=16: 9.477 − 8.481 = 0.996s (~10.5%)
- P=32: 5.627 − 4.498 = 1.129s (~20.1%)
- P=64: 4.492 − 3.182 = 1.310s (~29.2%)

Για το 4096×4096, τα time-plots δείχνουν ότι το overhead (Ttotal−Tcomp) αυξάνει τόσο σε απόλυτη τιμή όσο και ως ποσοστό του συνολικού χρόνου, κάτι που εξηγεί άμεσα τον κορεσμό του speedup μετά το P=32. Συγκεκριμένα, το Toverhead ανεβαίνει από ~0.48s (P=8) σε ~1.00s (P=16), ~1.13s (P=32) και ~1.31s (P=64), ενώ παράλληλα το ποσοστό του overhead αυξάνει περίπου από ~2.7% σε ~10.5%, ~20.1% και ~29.2%. Αυτό σημαίνει ότι, όσο αυξάνουμε τις διεργασίες, οι ανταλλαγές halo και οι αναμονές/synchronization ανά iteration καταναλώνουν όλο και μεγαλύτερο μέρος του συνολικού χρόνου.

Παρότι ο υπολογισμός παραμένει ακόμη το κυρίαρχο κομμάτι (το Tcomp δεν είναι αμελητέο), η μείωσή του από P=32 → P=64 είναι περιορισμένη (4.498s → 3.182s), άρα ο διπλασιασμός διεργασιών δεν μπορεί να φέρει αντίστοιχο κέρδος. Η επιπλέον παραλληλία μεταφράζεται κυρίως σε: (i) υψηλότερο σχετικό κόστος halo exchange (surface/volume effect καθώς μικραίνει το local block), (ii) latency-dominated επικοινωνία σε μικρότερα μηνύματα και (iii) αυξημένες αναμονές στο Waitall/critical path (η πιο αργή διεργασία καθορίζει τον ρυθμό). Έτσι, μετά το P=32 η εκτέλεση μπαίνει σταδιακά σε μια περιοχή όπου το communication/synchronization και η μειωμένη αποδοτικότητα πόρων (π.χ. contention σε interconnect/μνήμη) περιορίζουν το πρόσθετο όφελος και οδηγούν σε κορεσμό του speedup.

(γ) Matrix 6144×6144

- P=8: 39.624 − 38.524 = 1.100s (~2.8%)
- P=16: 21.258 − 19.327 = 1.931s (~9.1%)
- P=32: 12.612 − 10.255 = 2.357s (~18.7%)
- P=64: 10.740 − 8.141 = 2.599s (~24.2%)

Για το 6144×6144 παρατηρούμε την ίδια βασική εικόνα με το 4096×4096, αλλά σε απόλυτα μεγαλύτερους χρόνους: το overhead (Ttotal−Tcomp) αυξάνει συστηματικά με το P, ενώ η μείωση του υπολογιστικού χρόνου από P=32 → P=64

είναι πλέον σχετικά περιορισμένη. Από τα δεδομένα προκύπτει ότι το Toverhead ανεβαίνει από ~1.10s (P=8) σε ~1.93s (P=16), ~2.36s (P=32) και ~2.60s (P=64), ενώ το ποσοστό του overhead αυξάνει αντίστοιχα (~2.8% → ~9.1% → ~18.7% → ~24.2%). Άρα, όσο αυξάνεται το πλήθος διεργασιών, οι ανταλλαγές halo και οι αναμονές/synchronization γίνονται ολοένα πιο σημαντικό τμήμα του συνολικού χρόνου.

Παρότι το πρόβλημα είναι πιο compute-intense (άρα αντέχει καλύτερα την κλιμάκωση στα μικρότερα P), σε υψηλά P κυριαρχεί ξανά το surface/volume effect: το τοπικό block μικραίνει, η σχετική περίμετρος αυξάνει, και ο χρόνος επικοινωνίας γίνεται πιο latency-dominated. Επιπλέον, το Waitall ανά iteration επιβάλλει bulk-synchronous ρυθμό, όπου τυχόν καθυστερήσεις (π.χ. λόγω contention στο interconnect ή ανομοιομορφιών στους κόμβους) μετατρέπονται σε πραγματικό κόστος αναμονής για όλους. Έτσι, ο διπλασιασμός από 32→64 διεργασίες μειώνει μεν το Tcomp (10.255s → 8.141s), αλλά το Toverhead παραμένει σημαντικό και αυξάνει, με αποτέλεσμα το συνολικό κέρδος να είναι μικρό (12.612s → 10.740s) και το speedup να εμφανίζει κορεσμό μετά το P=32 — ακριβώς όπως προβλέπει η θεωρία strong scaling όταν τα communication/synchronization κόστη γίνονται συγκρίσιμα με τον υπολογισμό.

3) <u>Περαιτέρω ερμηνεία</u>

- Surface/volume effect: Με 2D decomposition, το compute ανά διεργασία είναι περίπου O(N^2/P), ενώ η επικοινωνία halo είναι περίπου O(N/√P) ανά διεργασία (περίμετρος του block). Ο λόγος επικοινωνίας/υπολογισμού αυξάνει όσο αυξάνει το P (και όσο μειώνεται το τοπικό block), άρα το efficiency πέφτει.
- Latency dominance: Σε μεγάλα P, τα μηνύματα γίνονται μικρότερα. Τότε το latency (α) κυριαρχεί έναντι του bandwidth (β·m), άρα δεν μπορούμε να κερδίσουμε όσο θα περιμέναμε από το 1/P του υπολογισμού.
- Συγχρονισμός/critical path: Παρότι χωρίς convergence δεν έχουμε Allreduce, κάθε iteration έχει πρακτικά συγχρονισμό μέσω Waitall (και μέσω της ανάγκης να έχουν φτάσει τα halo πριν τον υπολογισμό). Έτσι η πιο αργή διεργασία καθορίζει την ταχύτητα όλων, ειδικά σε περιπτώσεις μικρής ανισορροπίας ή θορύβου από το σύστημα.
- Περιορισμοί πόρων (μνήμη/NUMA/oversubscription): Σε υψηλά P, μπορεί να εμφανιστεί κορεσμός bandwidth μνήμης, contention σε shared πόρους, ή/και μειωμένο locality. Αυτό εξηγεί γιατί σε 4096/6144 το compute δεν συνεχίζει να μειώνεται ανάλογα με το P και γιατί το speedup μετά το 32 δεν διπλασιάζεται.

Σε αντίθεση με την 4.2.1, εδώ δεν υπάρχει κόστος global convergence check (τοπικό κριτήριο + Allreduce). Άρα, οποιαδήποτε απόκλιση από ιδανικό speedup οφείλεται κυρίως σε:

- halo exchanges ανά iteration,
- synchronization/αναμονές,
- και γενικότερη strong-scaling αναποτελεσματικότητα (surface/volume, latency, πόροι).

Αυτό επιβεβαιώνεται από το γεγονός ότι το Toverhead = Ttotal−Tcomp αυξάνει συστηματικά με το P και τελικά περιορίζει τον ρυθμό βελτίωσης.

Τελικό συμπέρασμα

Χωρίς έλεγχο σύγκλισης (σταθερό T=256), η υλοποίηση αυτή παρουσιάζει strong scaling (αλλά πάντα μακριά από το ιδανικό) μέχρι ~32 διεργασίες, αλλά από εκεί και πάνω εμφανίζεται κορεσμός, ιδιαίτερα στα μεγαλύτερα πλέγματα 4096×4096 και 6144×6144. Τα time-plots δείχνουν καθαρά ότι, όσο αυξάνεται το P, ο υπολογισμός μειώνεται, αλλά το επικοινωνιακό/συγχρονιστικό overhead αυξάνει (σε ποσοστό και συχνά και σε απόλυτη τιμή), οδηγώντας σε χαμηλό efficiency και περιορισμένο επιπλέον κέρδος στα P=64.

▪ Γενικά Συμπεράσματα

Στην Άσκηση 4 μελετήσαμε την παραλληλοποίηση και βελτιστοποίηση σε αρχιτεκτονικές κατανεμημένης μνήμης με MPI, πάνω σε δύο διαφορετικά μοτίβα υπολογισμού: (i) K-means (κυρίως data-parallel υπολογισμός με συλλογικές επικοινωνίες ανά επανάληψη) και (ii) Jacobi 2D (υπολογισμός με γειτονική επικοινωνία και halo exchange ανά επανάληψη, και προαιρετικό έλεγχο σύγκλισης με global reduction). Τα αποτελέσματα επιβεβαιώνουν τη θεωρία: όσο αυξάνουμε τον αριθμό διεργασιών για σταθερό πρόβλημα, ο καθαρός υπολογισμός ανά διεργασία μειώνεται, αλλά το κόστος επικοινωνίας/συγχρονισμών δεν μειώνεται αντίστοιχα και τελικά γίνεται ο κύριος περιοριστικός παράγοντας (Amdahl + communication overhead).

1) K-means με MPI

Ο K-means παρουσίασε ιδιαίτερα καλή κλιμάκωση στο συγκεκριμένο configuration (Size=256MB, Coords=16, Clusters=32, Loops=10), με χρόνο από 27.573s (sequential) έως 0.464s στα 64 processes και speedup ~59.4×. Αυτό είναι αναμενόμενο επειδή:

- Το κυρίαρχο κόστος είναι η ανάθεση σημείων σε κέντρα (assignment step), που είναι πλήρως data-parallel και διαμοιράζεται σχεδόν ιδανικά στα processes.
- Η επικοινωνία ανά iteration είναι κυρίως συλλογική με μικρό μήνυμα (π.χ. αθροίσματα/μετρήσεις για K×D, και counts για K). Άρα η επιβάρυνση είναι περισσότερο latency-dominated αλλά μικρής συνολικής διάρκειας σε σχέση με τον υπολογισμό πάνω σε εκατομμύρια objects.
- Η συγκέντρωση/συγχρονισμός ανά iteration δημιουργεί σημείο συγχρονισμού (bulk synchronous pattern), όμως η αναλογία υπολογισμού/επικοινωνίας παραμένει ευνοϊκή, για αυτό και η απόδοση παραμένει σχεδόν γραμμική μέχρι υψηλά P.

2) Jacobi 2D χωρίς έλεγχο σύγκλισης (σταθερές επαναλήψεις)

Χωρίς έλεγχο σύγκλισης, οι μετρήσεις απομονώνουν καλύτερα το καθαρό κόστος. Παρατηρούμε ότι:

- Για μικρότερο πλέγμα (2048×2048), ο χρόνος επικοινωνίας/αναμονών γίνεται συγκρίσιμος ή και μεγαλύτερος από τον υπολογισμό όταν αυξάνει το P, επειδή το τοπικό υποπλέγμα μικραίνει και ο λόγος επιφάνειας/όγκου (surface/volume) χειροτερεύει. Αυτό φαίνεται χαρακτηριστικά στα 64 processes, όπου ο υπολογισμός πέφτει πολύ (0.058s) αλλά ο συνολικός χρόνος δεν ακολουθεί ανάλογα (0.438s), άρα κυριαρχεί πλέον το κόστος επικοινωνίας και συγχρονισμού.
- Για μεγαλύτερα πλέγματα (4096×4096 και 6144×6144) η κλιμάκωση βελτιώνεται αρχικά, γιατί αυξάνει το computation per process. Ωστόσο, μετά από ένα σημείο (π.χ. από 32→64 processes) το όφελος μειώνεται αισθητά: ο συνολικός χρόνος συνεχίζει να μειώνεται, αλλά με ολοένα μικρότερο ρυθμό, επειδή οι ανταλλαγές ορίων και τα Wait/Sync κόστη αυξάνουν με το P και με τον αριθμό γειτόνων ανά iteration.

Θεωρητικά, το halo exchange έχει κόστος που προσεγγίζει α + β·m ανά μήνυμα (latency + bandwidth term), όπου m είναι το μέγεθος των γραμμών/στηλών που ανταλλάσσονται. Καθώς m μειώνεται με το partitioning, ο bandwidth όρος μικραίνει, αλλά ο latency όρος (α) και οι συγχρονισμοί (Waitall) παραμένουν, οπότε το ποσοστό μη-χρήσιμου χρόνου μεγαλώνει.

3) Jacobi 2D με έλεγχο σύγκλισης

Με ενεργό έλεγχο σύγκλισης προστίθεται επιπλέον κόστος από global reductions (π.χ. MPI_Allreduce) κάθε C επαναλήψεις. Επειδή το μήνυμα είναι πολύ μικρό (flag/int), το κόστος είναι σχεδόν εξ ολοκλήρου latency-dominated και αυξάνει περίπου με O(logP) σε βήματα επικοινωνίας, ενώ επιβάλλει και καθολικό συγχρονισμό (όλοι τα ranks πρέπει να φτάσουν στο σημείο ελέγχου). Άρα:

- Η συχνότητα ελέγχου (C) επηρεάζει άμεσα τον χρόνο: μικρό C → περισσότερα global syncs → χειρότερο strong scaling.
- Παράλληλα, ο έλεγχος μπορεί να μειώσει το συνολικό πλήθος επαναλήψεων όταν υπάρχει έγκαιρη σύγκλιση, άρα υπάρχει trade-off λιγότερες επαναλήψεις έναντι ακριβότερης επανάληψης.

Στο συγκεκριμένο πείραμα (512×512, 64 processes), ο συνολικός χρόνος είναι πολύ μεγαλύτερος από τον καθαρό υπολογισμό, γεγονός που επιβεβαιώνει ότι σε μικρό πρόβλημα/υψηλό P κυριαρχούν επικοινωνίες/συγχρονισμοί και όχι το υπολογιστικό kernel.

4) Σύγκριση MPI με OpenMP (σε επίπεδο συμπεριφοράς/θεωρίας)

Η σύγκριση με OpenMP αναδεικνύει την ουσιαστική διαφορά κοινής ενάντια σε κατανεμημένη μνήμη:

- Στο OpenMP, η επικοινωνία μεταξύ threads είναι implicit μέσω κοινής μνήμης. Αυτό μειώνει δραστικά το latency σε σχέση με MPI, αλλά εισάγει άλλους περιορισμούς: contention σε shared δεδομένα, memory bandwidth saturation, false sharing και NUMA effects. Επιπλέον, απαιτούνται μηχανισμοί reduction/atomics/critical για αθροίσεις (π.χ. K-means), που μπορεί να περιορίσουν την κλιμάκωση σε πολλά threads.
- Στο MPI, η κλιμάκωση σε πολλούς κόμβους είναι φυσική και το working set ανά process μειώνεται, αλλά πληρώνουμε ρητή επικοινωνία (latency/bandwidth) και συγχρονισμούς (collectives, Waits). Για τέτοια προβλήματα, η απόδοση εξαρτάται έντονα από το surface/volume και την τοπολογία επικοινωνίας (neighbors), ενώ για data-parallel με μικρά collectives (K-means) η κλιμάκωση μπορεί να είναι πολύ καλή.

Συνολικά, σε single-node περιβάλλον το OpenMP συχνά υπερέχει για μικρό/μεσαίο P λόγω χαμηλού communication overhead, ενώ σε multi-node/μεγάλα δεδομένα η MPI προσφέρει την απαραίτητη κατανεμημένη κλιμάκωση. Πρακτικά, η βέλτιστη προσέγγιση σε σύγχρονα συστήματα είναι συχνά υβριδική (MPI μεταξύ κόμβων + OpenMP εντός κόμβου), ώστε να μειωθούν τα MPI ranks και τα halo/collective κόστη.

Σ.Η.Μ.Μ.Υ. Ε.Μ.Π.
Ιανουάριος 2026