# Παράλληλα Συστήματα Εργασία 3: MPI

## Άσκηση 3.1

**Υλοποίηση:** Εμμανουήλ-Ταξιάρχης Οζίνης (sdi2300147)

### Περιγραφή προβλήματος

Το πρόβλημα μπορεί να χωριστεί στα εξής σημαντικά υποπροβλήματα:

- Την διατύπωση και υλοποίηση του σειριακού αλγορίθμου για πολλαπλασιασμό πολυωνύμων.
- Την **παραλληλοποίηση** του αλγορίθμου σε ικανοποιητικό βαθμό, δηλαδή με τέτοιο τρόπο που το παράλληλο τμήμα του είναι σημαντικά μεγαλύτερο από το μη παραλληλοποιημένο.

### Περιγραφή λύσης

Για την εκτέλεση του προγράμματος, με την βοήθεια του **Makefile** μπορούν να γίνουν δύο βασικές χρήσεις:

- Τοπική εκτέλεση σε ένα κόμβο, με ρητή χρήση των διαθέσιμων hardware threads:

  > make run-loc N=<pol degree> P=<process count>

- Εκτέλεση σε πολλούς κόμβους στο linux lab με ρητή χρήση των διαθέσιμων πυρήνων:

  > make run-lab N=<pol degree> P=<process count>

Για την οργάνωση της λύσης, αρχικά ορίζονται οι βοηθητικές συναρτήσεις:

- `parse_args()` : Διαβάζει τα command line arguments με κατάλληλους ελέγχους. Στην συγκεκριμένη περίπτωση το μόνο argument είναι ο βαθμός των πολυωνύμων, αφού το πλήθος διεργασιών δίνεται μέσω του `mpiexec` .
- `elapsed()` : Επιστρέφει την χρονική διαφορά σε δευτερόλεπτα (έως ακρίβεια nsec) από το start στο end. Σημειώνεται ότι για τις χρονικές μετρήσεις χρησιμοποιείται η δομή struct timespec της *C11*.
- `generate_random_coef()` : Δίνει τυχαίες μη μηδενικές ακέραιες τιμές σε ένα πολυωνυμο, όπου ο κάθε συντελεστής έχει απόλυτη τιμή το πολύ `MAX_ABS_COEFFICIENT_VALUE` .

Έχει οριστεί δομή `Polynomial` και για την διαχείρησή της υπάρχουν οι ακόλουθες συναρτήσεις:

- `pol_init()` : Δεσμεύει δομή Polynomial χρησιμοποιώντας ήδη δεσμευμένους συντελεστές.
- `pol_destroy()` : Αποδεσμεύει το πολυώνυμο και τους συντελεστές.
- `pol_print()` : Τυπώνει ένα πολυώνυμο, για λόγους debugging
- `pol_equals()` : Ελέγχει αν δύο πολυώνυμα είναι ίσα.
- `pol_add()` : Προσθέτει δύο πολυώνυμα. Τα πολυώνυμα και το αποτέλεσμα πρέπει να είναι ήδη δεσμευμένα. Αυτό γίνεται για λόγους ταχύτητας στον πολλαπλασιασμό.
- `pol_multiply()` : Ο σειριακός αλγόριθμος πολλαπλασιασμού πολυωνύμων, ο οποίος εκτελείται μόνο από μία διεργασία. Δεσμεύει ο ίδιος το αποτέλεσμα. Η υλοποίησή του επεξηγείται αργότερα.
- `pol_multiply_parallel()` : Ο παραλληλοποιημένος αλγόριθμος πολλαπλασιασμού. Σε αντίθεση με την `pol_multiply()` , αυτή η συνάρτηση καλείται από όλα τα ranks. Όλα τα διαφορετικά code paths που σχετίζονται με τον αλγόριθμο βρίσκονται εσωτερικά σε αυτή την συνάρτηση. Η ίδια η συνάρτηση δεσμεύει το αποτέλεσμα, το οποίο είναι non-NULL μόνο για το rank 0. Η υλοποίηση του παράλληλου αλγορίθμου επεξηγείται αργότερα.

Για την υλοποίηση του σειριακού αλγορίθμου, έχει γίνει η εξής προσέγγιση:
Έστω ότι πολλαπλασιάζονται τα `pol1` και `pol2` . Τότε, διατηρώντας ένα άθροισμα αρχικοποιημένο σε 0, **για κάθε όρο** του `pol1` πολλαπλασιάζουμε αυτό τον όρο με **ολόκληρο** το `pol2` , πολλαπλασιάζοντας ή μηδενίζοντας τους κατάλληλους συντελεστές και προσαρμόζοντας τις δυνάμεις. Στο τέλος κάθε επανάληψης προσθέτουμε το προσωρινό αποτέλεσμα στο άθροισμα. Έτσι στο τέλος του αλγορίθμου το άθροισμα είναι το αποτέλεσμα του πολλαπλασιασμού.

Για την **παραλληλοποίηση** του σειριακού αλγορίθμου έχει γίνει η εξής προσέγγιση: Παρατηρείται ότι ο σειριακός αλγόριθμος αποτελείται από δύο εμφωλευμένους βρόχους. Η παραλληλοποίηση επιλέχθηκε να γίνει στον εξωτερικό βρόχο επειδή έτσι το μοναδικό σημείο που απαιτεί συνδυασμό τοπικών αποτελεσμάτων είναι η πρόσθεση πολυωνύμων και η εγγραφή σε ένα συνολικό άθροισμα στην μνήμη του rank 0. Αυτό επίσης μπορεί να βελτιωθεί αν οι προσθέσεις των "τοπικών" επαναλήψεων γίνουν σε τοπικά αθροίσματα για κάθε διεργασία, και ο τελικός συνδυασμός περιλαμβάνει μόνο μία πρόσθεση πολυωνύμων για κάθε διεργασία. Ο λόγος που δεν επιλέχθηκε ο εσωτερικός βρόχος για παραλληλοποίηση είναι ότι εκτός της επιμέρους παραλληλοποίησης και της πρόσθεσης, θα απαιτούνταν πιο σύνθετη λογική και πολλές περισσότερες επικοινωνίες. Αυτό θα γινόταν γιατί το τμήμα κώδικα που πολλαπλασιάζει έναν όρο με ένα πολυώνυμο έχει διαφορετικό αριθμό επαναλήψεων από αυτές της πρόσθεσης, πράγμα που σημαίνει ότι (για την πλήρη αξιοποίηση των διεργασιών) ο διαμοιρασμός

τους θα γινόταν με διαφορετικές διεργασίες να χρησιμοποιούν αποτελέσματα άλλων διεργασιών, το οποίο θα απαιτούσε συνεχή επικοινωνία μεταξύ των διεργασιών, με μεγάλες επιβαρύνσεις.

Στην συνάρτηση `pol_multiply_parallel()`, ο παράλληλος αλγόριθμος μπορεί να περιγραφεί με τα εξής βασικά βήματα:

- Ο rank 0 κάνει broadcast τα degrees των δύο πολυωνύμων που θα πολλαπλασιαστούν (διαφορετικά degrees για γενικότητα).
- Εφόσον παραλληλοποιείται ο εξωτερικός βρόχος (που διατρέχει το `pol1`), ο rank 0 πρέπει να στείλει μόνο τμήματα του `pol1` σε κάθε διεργασία (μέσω `MPI_Scatterv()` για να μην χρειάζεται να διαιρούνται τέλεια τα δεδομένα). Από την άλλη το `pol2` πρέπει να γίνει broadcast γιατί διαβάζεται ολόκληρο από όλες τις διεργασίες. Όσον αφορά το `pol1`, στόχος του αλγορίθμου είναι οι συντελεστές του να μοιράζονται κυκλικά σε κάθε διεργασία για καλύτερο load balancing. Επομένως υπάρχει βήμα προεπεξεργασίας των συντελεστών του `pol1`, και επαναδιάταξή τους σε ένα `packed_chunks` buffer που ομαδοποιεί σε contiguous chunks τα ισοϋπόλοιπα indexes (mod rank) για κάθε rank.
- Γίνονται δεσμεύσεις μνήμης (μόνο για τοπική χρήση στον αλγόριθμο), οι οποίες επεξηγούνται αργότερα.
- Εκτελούνται οι εμφωλευμένοι βρόχοι τοπικά σε κάθε διεργασία.
- Κάθε μη μηδενικό rank στέλνει το τοπικό του άθροισμα στο rank 0, το οποίο αναλαμβάνει τις τελικές προσθέσεις και υπολογίζει το συνολικό αποτέλεσμα.

Σημειώνεται ότι στην συνάρτηση `pol_multiply_parallel()` γίνονται πριν την εκτέλεση των βρόχων αρκετές δεσμεύσεις μνήμης, έτσι ώστε να αποφευχθούν όσο το δυνατότερο δεσμεύσεις μέσα σε βρόχους του αλγορίθμου. Ωστόσο αυτές οι δεσμεύσεις θεωρούνται τμήμα του αλγορίθμου όσον αφορά τις μετρήσεις, επειδή ο σειριακός αλγόριθμος δεν έχει κατι αντίστοιχο, και επομένως αυτό πρέπει να θεωρηθεί overhead του παράλληλου αλγορίθμου. Επισης σημειώνεται ότι ο κώδικας για την δέσμευση αυτών των μεταβλητών έχει αλλάξει σε ένα βαθμό σε σχέση με το αντίστοιχο κώδικα της άσκησης 2.1 της εργασίας 2 (χρήση του `SAFE_CALL()` macro). Αυτή η αλλαγή έχει γίνει για λόγους καλύτερης οργάνωσης του κώδικα και δεν έχει επιρροή στον χρόνο εκτέλεσης, που σημαίνει ότι η σύγκριση της άσκησης 3.1 με την 1.1 ή 2.1 μπορεί να γίνει με συνεπή τρόπο για τον χρόνο εκτέλεσης.

## Πειραματικά αποτελέσματα

- **N**: Βαθμός των πολυωνύμων
- **P**: Αριθμός διεργασιών στον παράλληλο αλγόριθμο
- Σε όλα τα παρακάτω αποτελέσματα οι χρόνοι είναι σε *seconds* και δίνεται ακρίβεια *microsecond*.
- Αποφεύγεται η χρήση του N = *10^6* γιατί στο συγκεκριμένο σύστημα μια τέτοια εκτέλεση ολοκληρώνεται σε μη πρακτικό αριθμό λεπτών, καθώς η πολυπλοκότητα είναι τετραγωνική.
- Στα πειράματα με πολλούς κόμβους χρησιμοποιούνται μεγαλύτερα πλήθη από διεργασίες, συγκεκριμένα μέχρι και 64, αφού λόγω του διαχωρισμού των δεδομένων όχι μόνο ανά νήμα αλλά και ανά κόμβο, σε πολύ μεγάλο βαθμό σταματά να υπάρχει contention μεταξύ των διεργασιών. Αυτό σημαίνει ότι το efficiency μειώνεται μόνο σε μεγάλους αριθμούς από processes, και το 64 έχει επιλεχθεί ως ένα όριο με ένα σχετικά χαμηλό efficiency. Σημειώνεται ότι συγκεκριμένα για τα linux του εργαστηρίου (linux01 - linux30) όπου το καθένα έχει 4 πυρήνες-threads, το άνω όριο σε πραγματική παραλληλία θα ήταν 120 processes.
- Στην σύγκριση για μόνο έναν κόμβο, ο μέγιστος αριθμός από processes που χρησιμοποιείται στα πειράματα είναι **15** και όχι 16. Αυτό συμβαίνει λόγω της naive προσέγγισης που έχει γίνει στην άσκηση 1.1 με pthreads, επειδή το main thread που δημιουργεί τα pthreads δεν εκτελεί καμία "χρήσιμη" εργασία, αλλά απλώς περιμένει τα υπόλοιπα threads. Επίσης αυτό σημαίνει ότι τα πειράματα με 16 threads στην άσκηση 1.1 τεχνικά χρησιμοποιούσαν 17 threads, δηλαδή ένα παραπάνω νήμα που δεν εκτελούνταν πραγματικά παράλληλα με όλα τα υπόλοιπα. Αντίθετα, η άσκηση 3.1 με MPI αξιοποιεί όλα τα threads, συμπεριλαμβανομένου του main thread, και μπορεί πράγματι να τρέξει τον αλγόριθμο με 16 hardware threads (όπου αντιστοιχίζονται τα processes). Για να γίνει η σύγκριση σωστά όμως, εδώ θα χρησιμοποιηθούν έως 15 processes/threads και για τις δύο υλοποιήσεις.

Για τον παράλληλο αλγόριθμο ο συνολικός χρόνος μπορεί να χωριστεί σε τρεις φάσεις:

1. **Splitting/sending**: Γενικά η φάση αποστολής όλων των απαραίτητων δεδομένων στις διεργασίες από τον rank 0. Αυτό περιλαμβάνει και την κατάλληλη προεπεξεργασία όσων δεδομένων πρέπει να αποσταλούν με συγκεκριμένο τρόπο, όπως είναι η δημιουργία του `packed_chunks`.
2. **Computations**: Η φάση των αυτόνομων υπολογισμών για κάθε διεργασία παράλληλα, που αποτελεί το κυρίαρχο τμήμα του χρόνου εκτέλεσης. Επίσης περιλαμβάνει μερικές δεσμεύσεις μνήμης που χρησιμεύουν στον αλγόριθμο και διατηρούνται μονο τοπικά για κάθε διεργασία.
3. **Receiving/combining**: Η φάση κατά την οποία τα ήδη έτοιμα τοπικά αποτελέσματα αποστέλλονται στο rank 0, και αυτό τα συνδυάζει για να υπολογίσει το συνολικό αποτέλεσμα. Αυτό το τμήμα δεν περιλαμβάνει τις τελικές αποδεσμεύσεις μνήμης. Αυτές συνυπολογίζονται μόνο από τον συνολικό παράλληλο χρόνο.

Οι παρακάτω πίνακες αναφέρονται σε πειράματα που έγιναν χρησιμοποιώντας πολλούς κόμβους του εργαστηρίου linux.

Χρόνοι για **splitting/sending** για κάθε πείραμα με συγκεκριμένο N και P.

| **N\P** | **4** | **8** | **16** | **32** | **64** |
|---|---|---|---|---|---|
| **10000** | 0.000523 | 0.002427 | 0.007347 | 0.012789 | 0.016014 |

| N\P | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|
| **100000** | 0.002330 | 0.011846 | 0.022871 | 0.032288 | 0.035687 |

Χρόνοι για **computations** για κάθε πείραμα με συγκεκριμένο N και P.

| N\P | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|
| **10000** | 0.056805 | 0.034396 | 0.024946 | 0.013559 | 0.003490 |
| **100000** | 5.772085 | 2.829935 | 1.420478 | 0.852273 | 0.402394 |

Χρόνοι για **receiving/combining** για κάθε πείραμα με συγκεκριμένο N και P.

| N\P | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|
| **10000** | 0.000698 | 0.006357 | 0.002050 | 0.242087 | 0.224595 |
| **100000** | 0.020286 | 2.256214 | 1.098981 | 0.637374 | 0.826898 |

Σειριακοί χρόνοι εκτέλεσης για κάθε πείραμα με συγκεκριμένο N και P.

| N\P | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|
| **10000** | 0.189322 | 0.189159 | 0.189530 | 0.187855 | 0.187474 |
| **100000** | 19.184240 | 19.082744 | 19.135596 | 19.755201 | 19.665321 |

Παράλληλοι χρόνοι εκτέλεσης για κάθε πείραμα με συγκεκριμένο N και P, σε αντιστοιχία με τον παραπάνω πίνακα.

| N\P | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|
| **10000** | 0.058205 | 0.046031 | 0.035546 | 0.270807 | 0.256188 |
| **100000** | 5.794912 | 5.102816 | 2.543411 | 1.532030 | 1.283519 |

*Speedup* παράλληλης εκτέλεσης σε σχέση με την σειριακή, σε αντιστοιχία με τους παραπάνω πίνακες.

| N\P | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|
| **10000** | 3.252695 | 4.10938 | 5.331889 | 0.693685 | 0.731786 |
| **100000** | 3.310532 | 3.739650 | 7.523596 | 12.894783 | 15.321404 |

*Efficiency* σε σχέση με το *speedup* και τον αριθμό των νημάτων, σε αντιστοιχία με τους παραπάνω πινακες.

| N\P | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|
| **10000** | 0.827633 | 0.467456 | 0.470225 | 0.021678 | 0.011434 |
| **100000** | 0.827633 | 0.467456 | 0.470225 | 0.402962 | 0.239397 |

## Σχολιασμός αποτελεσμάτων

Για τα παραπάνω αποτελέσματα να σημειωθεί ότι λόγω της συνύπαρξης πολλών χρηστών στα συστήματα linux, δεν ήταν εύκολο να βρεθεί ιδανική στιγμή για μετρήσεις. Επομένως οι παραπάνω τιμές δεν αντιστοιχούν στις πραγματικές δυνατότητες των μηχανημάτων, καθώς σε προηγούμενες δοκιμές έχουν παρατηρηθεί και αισθητά καλύτερες επιδόσεις.

Από τους παραπάνω πίνακες συμπεραίνουμε τα εξής:

- Γενικά υπάρχει αισθητή επιτάχυνση, η οποία είναι πολύ σημαντικότερη για μεγαλύτερες τιμές του *N*.
- Ήδη από τις 8 διεργασίες και άνω παρατηρείται σημαντική πτώση στην αποδοτικότητα. Παρόλα αυτά για ένα μεγάλο διάστημα, μέχρι περίπου τις 32 διεργασίες η αποδοτικότητα μειώνεται ελάχιστα, και η επόμενη σημαντική μείωση παρατηρείται μετά από αυτά. Το συγκεκριμένο φαινόμενο πιθανότατα οφείλεται στην συνύπαρξη πολλών χρηστών, διότι σε μεμονωμένες δοκιμές (όχι συνολικά) πριν από την τελική μέτρηση έχει παρατηρηθεί και πιο αναμενόμενη μείωση του χρόνου από 4 σε 8 διεργασίες.
- Στην φάση splitting/sending, όπως είναι αναμενόμενο, οι χρόνοι είναι σχετικά μικροί όμως αυξάνονται είτε με την αύξηση των διεργασιών είτε με την αύξηση του N.

- Στην φάση computations οι χρόνοι μειώνονται πιο προβλέψιμα από τους συνολικούς, και συγκεκριμένα για N=10^5, οι χρόνοι φαίνονται περίπου να υποδιπλασιάζονται, όσο οι διεργασίες διπλασιάζονται.
- Στην φάση receiving/combining οι χρόνοι για N=10^5 φαίνονται αρκετά απρόβλεπτοι, και ίσως αυτό οφείλεται σε internal υλοποιήσεις των μηχανισμών επικοινωνίας μεταξύ των διεργασιών. Μια αιτία θα μπορούσε να είναι, για παράδειγμα, το αν μια επικοινωνία μεταξύ δύο διεργασιών πρέπει να γίνει μέσω τοπικού δικτύου (σε περίπτωση διαφορετικών κόμβων) ή μέσω μηχανισμών IPC του λειτουργικού συστήματος (σε περίπτωση που είναι στον ίδιο κόμβο).
- Στην φάση receiving/combining, για N=10^5 υπάρχει μια ξαφνική αύξηση του χρόνου απο 4 σε 8 διεργασίες. Αυτή η ξαφνική αύξηση είναι ο άμεσος λόγος για τον οποίο το συνολικό speedup από 4 σε 8 διεργασίες δεν αλλάζει σχεδόν καθόλου. Παρόλα αυτά, ο έμμεσος λόγος, όπως αναφέρθηκε και πριν, είναι πιθανότατα ο ανταγωνισμός μεταξύ διεργασιών διαφορετικών χρηστών για πόρους.

### Σύγκριση με υλοποίηση `pthreads`

Εφόσον η υλοποίηση με pthreads ήταν αποδοτικότερη από την υλοποίηση με OpenMP, η σύγκριση της υλοποίησης με MPI θα συγκριθεί με αυτή των pthreads.

Οι παρακάτω πίνακες αναφέρονται σε πειράματα που έγιναν τοπικά σε έναν μόνο κόμβο, τα χαρακτηριστικά του οποίου παρατίθονται στο τέλος της αναφοράς 3.1.

Υλοποίηση **pthreads**: *Speedup* παράλληλης εκτέλεσης σε σχέση με την σειριακή

| **N\Threads** | **1** | **2** | **4** | **8** | **15** |
|---|---|---|---|---|---|
| **1000** | 0.791199 | 1.239546 | 1.288399 | 1.229979 | 0.969816 |
| **10000** | 1.103307 | 2.099371 | 3.287014 | 3.734003 | 4.312176 |
| **100000** | 1.118343 | 2.216025 | 3.914853 | 6.188193 | 7.613948 |

Υλοποίηση **pthreads**: *Efficiency* σε αντιστοιχία με τον προηγούμενο πίνακα.

| **N\Threads** | **1** | **2** | **4** | **8** | **15** |
|---|---|---|---|---|---|
| **1000** | 0.791199 | 0.619773 | 0.322099 | 0.153747 | 0.064654 |
| **10000** | 1.103307 | 1.049685 | 0.821753 | 0.466750 | 0.287478 |
| **100000** | 1.118343 | 1.108012 | 0.978713 | 0.773524 | 0.507596 |

Υλοποίηση **MPI**: *Speedup* παράλληλης εκτέλεσης σε σχέση με την σειριακή

| **N\P** | **1** | **2** | **4** | **8** | **15** |
|---|---|---|---|---|---|
| **1000** | 0.962349 | 1.796098 | 2.728352 | 0.724516 | 0.546225 |
| **10000** | 1.039938 | 1.923327 | 3.557693 | 3.725310 | 5.765541 |
| **100000** | 1.061416 | 2.096624 | 3.778442 | 4.972481 | 6.018304 |

Υλοποίηση **MPI**: *Efficiency* σε αντιστοιχία με τον προηγούμενο πίνακα.

| **N\P** | **1** | **2** | **4** | **8** | **15** |
|---|---|---|---|---|---|
| **1000** | 0.962349 | 0.898049 | 0.682088 | 0.090565 | 0.036415 |
| **10000** | 1.039938 | 0.961663 | 0.889423 | 0.465664 | 0.384369 |
| **100000** | 1.061416 | 1.048312 | 0.944610 | 0.621560 | 0.401220 |

Παρατηρούμε ότι μεταξύ των δύο υλοποιήσεων, η υλοποίηση με **pthreads** είναι αποδοτικότερη από αυτή του MPI, αφού για μεγαλύτερα N οι επιταχύνσεις με pthreads είναι σχεδόν όλες μεγαλύτερες από τις αντίστοιχες με MPI. Αυτό είναι αναμενόμενο γιατί το MPI είναι σχεδιασμένο για παραλληλία σε μεγαλύτερη κλίμακα (και όχι απλά μεταξύ νημάτων σε ένα κόμβο) και απαιτεί αναγκαία overheads, το σημαντικότερο από τα οποία είναι η επικοινωνία μεταξύ διεργασιών. Αντίθετα, τα pthreads είναι πιο low level και εστιάζουν στην χρήση νημάτων για ένα μόνο σύστημα κοινόχρηστης μνήμης, με αποτέλεσμα η χρήση των νημάτων να είναι προσδιορισμένη με περισσότερη ακρίβεια, που συνεπάγεται καλύτερη αποδοτικότητα.

Για τα πειράματα σε ένα κόμβο (MPI/pthreads) χρησιμοποιήθηκε σύστημα με τα εξής χαρακτηριστικά:

- **Μοντέλο laptop:** MSI Katana GF66 12UC
- **Μοντέλο CPU:** 12th Gen Intel(R) Core(TM) i7-12650H
- **Logical processors:** 16

- **OS:** WSL2 (Ubuntu) πάνω σε Windows11
- **C compiler:** gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0

**Σημείωση:** Η hardware μνήμη RAM είναι 16GB όμως το WSL την περιορίζει σε ~8GB. Παρόλα αυτά **τονίζεται ότι το WSL δεν περιορίζει την χρήση των hardware threads**.

# Άσκηση 3.2

## Χαρακτηριστικά υλοποίησης

- Υλοποίηση προγράμματος από: Κωνσταντίνο Γεώργιο Βλαζάκη (sdi2300017)
- Τα προγράμματα εκτελέστηκαν στα εργαστηριακά συστήματα Linux.

## Παραδοχές:

- Όλες οι ακέραιες τιμές σε πίνακες και διανύσματα βρίσκονται στο διάστημα [0, 100], ώστε να είναι ευανάγνωστα τα δεδομένα.
- Το πρόγραμμα δεν μπορεί να λάβει τιμή διάστασης του πίνακα μεγαλύτερη από $10^4$, καθώς καταλήγει σε υπερχείλιση (overflow) ακεραίων.
- Υπάρχει ενδεχόμενο να χρειαστεί η εκτέλεση του setup script "**askdate.sh**". Υπό αυτή την περίπτωση, μπορεί να εκτελεστεί με την εντολή "make setup"

## Υλοποιήσεις

### main: Κύριο πρόγραμμα προς εκτέλεση:

- Ξεκινάει θέτοντας τα αναγκαία δεδομένα για την εκτέλεση του προγράμματος. Αυτά αποτελούν τα ορίσματα από τη γραμμή εντολών (Διάσταση πίνακα, ποσοστό μηδενικών στοιχείων πίνακα, επαναλήψεις πολλαπλασιασμού πίνακα), καθώς και τα παράγωγά τους (συνολικό αριθμό τιμών που θα περιέχει ο πίνακας, αριθμό μηδενικών στοιχείων στον πίνακα). Υπολογίζει επίσης τον αριθμό μη μηδενικών στοιχείων που θα υπάρχουν στον πίνακα, για τη δημιουργία και χρήση της Compressed Sparse Row (CSR) αναπαράστασης αραιού πίνακα.
- Αρχικοποιεί τις "**timespec**" μεταβλητές που θα χρειαστούν για τη χρονομέτρηση της εκτέλεσης.
- Αρχικοποιεί το πρωτόκολλο MPI, καθώς και διάφορες τιμές πινάκων, διανυσμάτων και αναπαραστάσεων CSR, οι οποίες θα συμπληρωθούν κατά την εκτέλεση του προγράμματος.
- Δημιουργεί έναν δισδιάστατο πίνακα χρησιμοποιώντας δυναμική δέσμευση μνήμης (με αριθμό γραμμών "dimension" και αριθμό στηλών "dimension", καθώς είναι τετραγωνικός) και θέτει όλα τα στοιχεία του σε τυχαίους ακέραιους αριθμούς. Αφότου γεμίσει τον πίνακα με ακέραιους, μηδενίζει τόσους από αυτούς όσους έχει ορίσει ο χρήστης, σε τυχαίες θέσεις του πίνακα. Έτσι παράγεται ένας αραιός πίνακας. Ταυτοχρόνως, παράγει μία εκδοχή του ίδιου πίνακα, σε μονοδιάστατη μορφή. Αυτή η εκδοχή χρησιμοποιείται με τις MPI συναρτήσεις, ώστε τα στοιχεία να βρίσκονται σε συνεχόμενες θέσεις μνήμης.
- Δημιουργεί ένα διάνυσμα χρησιμοποιώντας δυναμική δέσμευση μνήμης (με αριθμό γραμμών "dimension"), και θέτει όλα τα στοιχεία του σε τυχαίους ακέραιους αριθμούς.
- Παράγει σειριακά την CSR αναπαράσταση του πίνακα (μέσω της σειριακής συνάρτησης CSR_create) και μετράει τον χρόνο εκτέλεσής της.
- Πολλαπλασιάζει σειριακά τον πίνακα με το διάνυσμα τόσες φορές όσες έχει ορίσει ο χρήστης (με κάθε επανάληψη να λαμβάνει ως διάνυσμα εισόδου το διάνυσμα εξόδου της προηγούμενης), και μετράει τον χρόνο εκτέλεσης όλων των επαναλήψεων.
- Πολλαπλασιάζει σειριακά την CSR αναπαράσταση του πίνακα με το διάνυσμα τόσες φορές όσες έχει ορίσει ο χρήστης (με κάθε επανάληψη να λαμβάνει ως διάνυσμα εισόδου το διάνυσμα εξόδου της προηγούμενης), και μετράει τον χρόνο εκτέλεσης όλων των επαναλήψεων.
- Για την παράλληλη εκτέλεση, αρχικοποιεί μεταβλητές για την κατανομή των γραμμών του πίνακα προς κάθε διεργασία, ώστε να μπορούν όλες να επεξεργάζονται ξεχωριστά κομμάτια (blocks) του πίνακα.
- Παράγει παράλληλα την CSR αναπαράσταση του πίνακα (μέσω της παράλληλης συνάρτησης CSR_create_mpi) και μετράει τον χρόνο εκτέλεσής της.
    - Για να επιτευχθεί αυτό με κατανομή γραμμών (του πίνακα "**mat_mpi**") ανά διεργασία, χρησιμοποιούνται οι συναρτήσεις MPI_Scatterv και MPI_Gatherv. Αρχικοποιούνται οι λίστες "**sendcounts**"/"**send_displacements**" για να οριστεί ακριβώς ο αριθμός των γραμμών που θα σταλθεί σε κάθε διεργασία, και οι λίστες "**idx_recvcounts**"/"**idx_displacements**" για να οριστεί ο αριθμός των δεικτών (από τη λίστα start_idx του CSR_t αντικειμένου) που θα παραληφθεί από κάθε διεργασία στο τέλος της εκτέλεσης. Επιπλέον, αρχικοποιούνται οι λίστες "**nzcounts**"/"**nz_displacements**", που παραλαμβάνουν (μέσω της MPI_Gather) τους αριθμούς των μη-μηδενικών στοιχείων που έχει επεξεργαστεί κάθε διεργασία. Τέλος οι "**nzcounts**"/"**nz_displacements**" χρησιμοποιούνται για την παραλαβή των λιστών "**val_array**"/"**col_array**", ώστε να συμπληρωθεί ολοκληρωτικά η CSR αναπαράσταση.
- Πολλαπλασιάζει παράλληλα τον πίνακα με το διάνυσμα τόσες φορές όσες έχει ορίσει ο χρήστης και μετράει τον χρόνο εκτέλεσης.
    - Για να επιτευχθεί αυτό με κατανομή γραμμών (του πίνακα "**mat_mpi**") ανά διεργασία, χρησιμοποιούνται οι συναρτήσεις MPI_Scatterv και MPI_Gatherv. Αρχικοποιούνται τα διανύσματα "**priv_vec**" και "**priv_res**" για τα κομμάτια των διανυσμάτων πολλαπλασιασμού και αποτελέσματος που θα παραλάβει/αποστείλει κάθε διεργασία. Αρχικοποιούνται και οι λίστες

"**recvcounts**"/"**recv_displacements**" για την κατανομή (με "**MPI_Scatterv**") του διανυσματος πολλαπλασιασμού και τη συμπλήρωση (με "**MPI_Gatherv**") του διανύσματος αποτελέσματος.

- Πολλαπλασιάζει παράλληλα την CSR αναπαράσταση του πίνακα με το διάνυσμα τόσες φορές όσες έχει ορίσει ο χρήστης και μετράει τον χρόνο εκτέλεσης.
  - Αρχικοποιούνται οι λίστες "**rowcounts**"/"**row_displacements**" για την κατανομή (με "**MPI_Scatterv**") της λίστας "**start_idx**" του CSR_t αντικειμένου, και χρησιμοποιούνται οι λίστες "**nzcounts**"/"**nz_displacements**" για την κατανομή των λιστών "**val_array**"/"**col_array**". Χρησιμοποιούνται επίσης οι λίστες "**recvcounts**"/"**recv_displacements**" για την κατανομή (με "**MPI_Scatterv**") του διανύσματος πολλαπλασιασμού, καθώς και για τη συμπλήρωση (με "**MPI_Gatherv**") του διανύσματος αποτελέσματος.
  - **Επισήμανση**: Κατά την κατανομή των λιστών του CSR_t αντικειμένου, είναι αναγκαίο οι τιμές της λίστας "**start_idx**" να μετατραπούν με σχετική βάση το μηδέν. Δηλαδή, εάν π.χ. μία διεργασία λάβει τις "**start_idx**" τιμές [5, 8], αυτές θα πρέπει να μετατραπούν σε [0, 3], το οποίο επιτυγχάνεται απλά αφαιρώντας την πρώτη τιμή (εδώ το 5) από κάθε στοιχείο. Εδώ δεν επηρεάζεται το τελευταίο στοιχείο κάθε λίστας "**start_idx**", εφόσον έχει ήδη τεθεί στην κατάλληλη τιμή (τον συνολικό αριθμό μη-μηδενικών τιμών που θα επεξεργαστεί η συνάρτηση).
- Τέλος, συλλέγει όλα τα δεδομένα που είναι σημαντικά για τον χρήστη (τις παραμέτρους του προγράμματος και τους χρόνους εκτέλεσης της κάθε διαδικασίας) σε αντίστοιχες λίστες, και τα αποθηκεύει σε ένα εξωτερικό αρχείο με όνομα "test_data.txt".

## matrixlib: Βιβλιοθήκη συναρτήσεων που σχετίζονται με πίνακες και λίστες:

- Δομή **CSR_t**: Χρησιμοποιείται για την αποθήκευση της μορφής CSR ενός πίνακα. Αποτελείται από μία λίστα ακεραίων για τις μη μηδενικές τιμές του αρχικού πίνακα (**val_array**), μία λίστα ακεραίων για τις στήλες όπου βρίσκονται αυτές οι τιμές (**col_array**), και μία λίστα "δεικτών" στις θέσεις της "**col_array**" όπου βρίσκονται τα πρώτα μη μηδενικά στοιχεία κάθε γραμμής του αρχικού (αραιού) πίνακα (**start_idx**).
- **CSR_create**: Δημιουργεί την CSR αναπαράσταση ενός δισδιάστατου πίνακα ακεραίων. Λαμβάνει έναν δισδιάστατο πίνακα ακεραίων, τους αριθμούς στηλών και γραμμών του πίνακα, καθώς και τον αριθμό μη μηδενικών στοιχείων. Αρχικοποιεί τις λίστες μίας δομής CSR_t, και προσθέτει τον αριθμό των μη μηδενικών στοιχείων στην τελευταία θέση της λίστας "**start_idx**". Διατρέχει σειριακά κάθε γραμμή του αρχικού πίνακα, όπου αναθέτει τιμή στο αντίστοιχο στοιχείο της λίστας "**start_idx**" και διατρέχει κάθε στήλη του αρχικού πίνακα ώστε να προσθέσει τις μη μηδενικές τιμές του στη λίστα "**val_array**" και τις αντίστοιχες στήλες τους στην "**col_array**". Τέλος, επιστρέφει τη συμπληρωμένη δομή CSR_t.
- **mat_vec**: Πολλαπλασιάζει έναν δισδιάστατο πίνακα ακεραίων με ένα διάνυσμα. Λαμβάνει τον αρχικό πίνακα ακεραίων, το διάνυσμα με το οποίο αυτός θα πολλαπλασιαστεί, και τους αριθμούς γραμμών και στηλών του πίνακα. Αρχικοποιεί το διάνυσμα αποτελέσματος. Διατρέχει σειριακά κάθε γραμμή και στήλη του πίνακα και αθροίζει τα γινόμενα των στοιχείων κάθε γραμμής με τα στοιχεία κάθε στήλης του διανύσματος. Τέλος, επιστρέφει το συμπληρωμένο διάνυσμα αποτελέσματος.
- **CSR_mat_vec**: Εφαρμόζει τον πολλαπλασιασμό δισδιάστατου πίνακα ακεραίων με διάνυσμα, χρησιμοποιώντας την CSR αναπαράσταση του πίνακα. Λαμβάνει τη δομή CSR_t που περιέχει την CSR αναπαράσταση, το διάνυσμα με το οποίο θα πολλαπλασιαστεί και τη διάσταση που αντιστοιχεί στον πίνακα και το διάνυσμα. Αρχικοποιεί το διάνυσμα αποτελέσματος. Διατρέχει σειριακά τις γραμμές, βρίσκει το πρώτο μη μηδενικό στοιχείο κάθε γραμμής του αρχικού πίνακα, καθώς και της ακριβώς επόμενης γραμμής. Διατρέχει όλα τα μη μηδενικά στοιχεία ανάμεσα σε αυτά που έχει βρει και αθροίζει τα γινόμενά τους με τα στοιχεία του διανύσματος. Τέλος, επιστρέφει το συμπληρωμένο διάνυσμα αποτελέσματος.
- **print_matrix**: Εκτυπώνει έναν δισδιάστατο πίνακα ακεραίων στην κονσόλα. Λαμβάνει τον δισδιάστατο πίνακα και τους αριθμούς γραμμών και στηλών του.
- **print_CSR**: Εκτυπώνει μία αναπαράσταση CSR στην κονσόλα. Λαμβάνει τη δομή CSR_t και τον αριθμό γραμμών του αρχικού πίνακα.
- **print_array**: Εκτυπώνει μία λίστα σε σειρά. Χρήσιμη για εμφάνιση μικρών διανυσμάτων/μονοδιάστατων πινάκων.
- **CSR_create_mpi**: Εκτελεί την ίδια λειτουργία με την CSR_create. Εδώ η υλοποίηση έχει αλλαχτεί ώστε να επιτραπεί η παραλληλοποίησή της. Χρησιμοποιεί τρεις ξεχωριστούς βρόχους for, ώστε να αφαιρεθεί η εξάρτηση που υπήρχε (κατά την εκτέλεση της CSR_create) πάνω στη μεταβλητή "**list_idx**". Ο πρώτος εκτελεί μία απλή μέτρηση των μη μηδενικών στοιχείων κάθε γραμμής του αρχικού πίνακα, όπου κάθε αριθμός αποθηκεύεται στην επόμενη θέση από αυτή που αντιστοιχεί στη γραμμή του. Ο δεύτερος αθροίζει κάθε αριθμό (μετά από την πρώτη θέση, καθώς πρέπει να παραμείνει "0") με τον ακριβώς προηγούμενό του. Πλέον, η λίστα περιέχει "δείκτες" στις ακριβείς θέσεις όπου υπάρχουν τα πρώτα μη μηδενικά στοιχεία κάθε γραμμής του αρχικού πίνακα. Ο τρίτος βρόχος διατρέχει τις λίστες "**val_array**" και "**col_array**", χρησιμοποιώντας τη συμπληρωμένη λίστα "**start_idx**", και αποθηκεύει όλα τα μη μηδενικά στοιχεία και τις αντίστοιχες στήλες όπου βρίσκονται.
  - Για παραλληλοποίηση μέσω MPI, η συνάρτηση λαμβάνει ένα τμήμα (block) του πίνακα (μέσω της MPI_Scatterv) που πρέπει να μετατρέψει σε CSR αναπαράσταση, το οποίο επεξεργάζεται και αποθηκεύει σε ένα ιδιωτικό CSR_t αντικείμενο. Το αντικείμενο που παράγεται από κάθε διεργασία συνδυάζεται σε μία τελική, ολοκληρωμένη CSR αναπαράσταση στην "**main**".
- **mat_vec_mpi**: Εκτελεί την ίδια λειτουργία με την "**mat_vec**", αλλά ο βρόχος που διατρέχει τις γραμμές του αρχικού πίνακα έχει παραλληλοποιηθεί μέσω του πρωτοκόλλου MPI.
  - Για παραλληλοποίηση μέσω MPI, η συνάρτηση λαμβάνει ένα τμήμα (block) του πίνακα και του διανύσματος που πρέπει να πολλαπλασιάσει (μέσω της "**MPI_Scatterv**"). Χρησιμοποιείται η "**MPI_Allgatherv**" για να συμπληρωθεί ολόκληρο το διάνυσμα πολλαπλασιασμού (είναι προτιμότερο αυτό από το να δοθεί το διάνυσμα πολλαπλασιασμού ολόκληρο σαν όρισμα σε κάθε διεργασία, σε περιπτώσεις όπου μπορεί να είναι το αποτέλεσμα ενός προηγούμενου πολλαπλασιασμού). Ο πολλαπλασιασμός εφαρμόζεται πάνω στο τμήμα (block) του πίνακα και το διάνυσμα πολλαπλασιασμού που έχει συμπληρωθεί από την

"**MPI_Allgatherv**". Προτιμούνται οι "**MPI_Scatterv**"/"**MPI_Allgatherv**" από τις "**MPI_Scatter**"/"**MPI_Allgather**", σε περιπτώσεις όπου ο συνολικός αριθμός γραμμών του πίνακα δεν διαιρείται από τον αριθμό διεργασιών.

- **CSR_mat_vec_mpi**: Εκτελεί την ίδια λειτουργία με την "**CSR_mat_vec**", αλλά ο βρόχος που διατρέχει τις γραμμές έχει παραλληλοποιηθεί μέσω του πρωτοκόλλου.
  - Χρησιμοποιείται η "**MPI_Allgatherv**" για να συμπληρωθεί ολόκληρο το διάνυσμα πολλαπλασιασμού (ακριβώς όπως στην "**mat_vec_mpi**"). Ο πολλαπλασιασμός εφαρμόζεται πάνω στα τμήματα (blocks) των λιστών του CSR_t αντικειμένου και το διάνυσμα πολλαπλασιασμού που έχει συμπληρωθεί από την "**MPI_Allgatherv**".
- **compare_array**: Συγκρίνει όλα τα στοιχεία δύο λιστών ανά θέση και επιστρέφει τον αριθμό των στοιχείων που δεν ταιριάζουν.
- **CSR_destroy**: Αποδεσμεύει τη μνήμη που καταλαμβάνουν οι λίστες ενός αντικειμένου CSR_t (μέσω free).

## Αποτελέσματα εκτελέσεων

- Προϋποθέσεις:
  - Ο χρόνος κάθε διαδικασίας μετριέται σε δευτερόλεπτα.
  - Κάθε εκτέλεση έχει επαναληφθεί 5 φορές ώστε να μετρηθεί ο μέσος όρος του χρόνου κάθε διαδικασίας.
  - Τα δεδομένα αυτά λήφθηκαν μέσω του bash script "**test_script.sh**" και της εντολής "**make test**".
  - Κάθε γραμμή περιγράφει τους χρόνους εκτέλεσης μίας διαδικασίας, κάθε στήλη την παράμετρο που έχει αλλάξει κατά τη συγκεκριμένη εκτέλεση.
  - Οι παράμετροι που αλλάζουν είναι οι εξής:
    - D -> Διάσταση πίνακα (αποτελεί αριθμό γραμμών και στηλών για τετραγωνικό πίνακα).
    - Z -> Ποσοστό μηδενικών στοιχείων στον πίνακα.
    - R -> Επαναλήψεις πολλαπλασιασμού πίνακα-διανύσματος.
    - N -> Αριθμός διεργασιών MPI προς εκτέλεση.
  - Οι προεπιλεγμένες (Default) παράμετροι είναι:
    - D = 1000.
    - Z = 60.
    - R = 5.
    - N = 4.
- Σειριακή εκτέλεση συναρτήσεων/διαδικασιών:

| **Function/Parameters:** | **Default** | **D=10000** | **Z=0** | **Z=99** | **R=1** | **R=20** |
|---|---|---|---|---|---|---|
| **Serial Product** | 0.097695 | 6.502394 | 0.061166 | 0.060509 | 0.012186 | 0.243252 |
| **Serial CSR Creation** | 0.065001 | 3.011539 | 0.023824 | 0.008086 | 0.033072 | 0.033648 |
| **Serial CSR Product** | 0.043985 | 4.052386 | 0.045874 | 0.000677 | 0.003726 | 0.074041 |

- Σχόλια:
  - Η αλλαγή της διάστασης πίνακα από 1000 σε 10000 παράγει μεγάλη διαφορά στον υπολογισμένο χρόνο εκτέλεσης, σε σύγκριση με κάθε άλλη αλλαγή.
  - Υπάρχει σημαντική επιτάχυνση στον πολλαπλασιασμό πίνακα-διανύσματος, όταν συγκρίνουμε την κοινή σειριακή υλοποίηση με την CSR σειριακή υλοποίηση.
- Παράλληλη εκτέλεση συναρτήσεων/διαδικασιών:

| **Function/Parameters:** | **Default** | **D=10000** | **Z=0** | **Z=99** | **R=1** | **R=20** | **N=3** | **N=2** | **N=1** |
|---|---|---|---|---|---|---|---|---|---|
| **Parallel Product** | 0.480716 | 0.740505 | 0.006644 | 0.006571 | 0.001387 | 0.026206 | 0.008566 | 0.012216 | 0.023594 |
| **Parallel CSR Creation** | 0.025411 | 0.666740 | 0.006093 | 0.001282 | 0.005742 | 0.005820 | 0.007067 | 0.009649 | 0.017384 |
| **Parallel CSR Product** | 1.042956 | 3.931365 | 0.041960 | 0.001057 | 0.008841 | 0.045440 | 0.019763 | 0.025627 | 0.043713 |

- Επιταχύνσεις από σειριακή σε παράλληλη εκτέλεση συναρτήσεων/διαδικασιών:

| **Function/Parameters:** | **Default** | **D=10000** | **Z=0** | **Z=99** | **R=1** | **R=20** | **N=3** | **N=2** | **N=1** |
|---|---|---|---|---|---|---|---|---|---|
| **Product Speedup** | 0.2032 | 8.7810 | 9.2062 | 9.2085 | 8.7859 | 9.2823 | 6.9281 | 4.7479 | 2.4811 |
| **CSR Creation Speedup** | 2.5580 | 4.5168 | 3.9101 | 6.3073 | 5.7597 | 5.7814 | 4.5765 | 3.2705 | 2.0397 |
| **CSR Product Speedup** | 0.0422 | 1.0308 | 1.0933 | 0.6405 | 0.4214 | 1.6294 | 0.9079 | 0.6821 | 0.3895 |

- Σχόλια:

- Η αλλαγή της διάστασης πίνακα από 1000 σε 10000 εξακολουθεί να παράγει μεγάλη διαφορά στον υπολογισμένο χρόνο εκτέλεσης, σε σύγκριση με κάθε άλλη αλλαγή.
- Στην παράλληλη εκτέλεση, ο πολλαπλασιασμός CSR-διανύσματος φέρνει χειρότερα αποτελέσματα από την κοινή υλοποίηση.
- Όσο μειώνεται ο αριθμός MPI διεργασιών, τόσο μειώνεται και η επιτάχυνση των συναρτήσεων γινομένου πίνακα-διανύσματος και δημιουργίας CSR, εφόσον έχει μειωθεί η παραλληλία. Το γινόμενο CSR-διανύσματος λαμβάνει αυξημένη επιτάχυνση για τον ίδιο λόγο.