Λογισμικό και Προγραμματισμός Συστημάτων Υψηλής Επίδοσης

Χειμερινό Εξάμηνο 2024-2025

1ο Σετ Εργαστηριακών Ασκήσεων

Ον/νυμο: Παναγιώτης Καλοζούμης Α.Μ.: 1084560

Ον/νυμο: Δημήτριος Στασινός Α.Μ.: 1084643

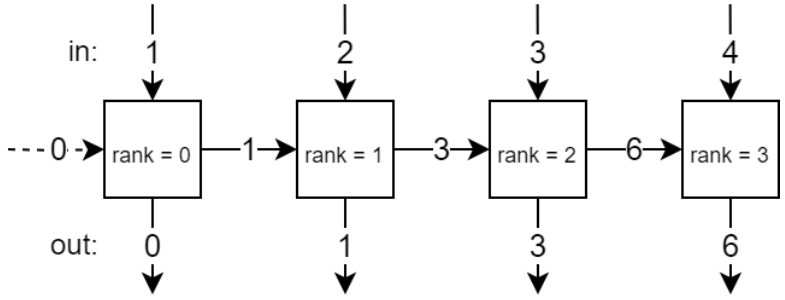
Ερώτημα 1: Υβριδικό Μοντέλο Προγραμματισμού και Παράλληλο I/O

Υλοποίηση Exscan

Η exscan που θα υλοποιήσουμε λειτουργεί ως εξής:

- Κάθε διεργασία δίνει σαν είσοδο μια τοπική τιμή **in**
- Πρέπει στο τέλος της διαδικασίας να έχει αποκτήσει μια άλλη τιμή **out**
- Υπάρχει αλυσίδα από εργασίες, με προτεραιότητα που ορίζεται από το rank
- Κάθε διεργασία δίνει σαν έξοδο την τιμή που έλαβε από την προηγούμενη διεργασία, με εξαίρεση την πρώτη διεργασία που δίνει έξοδο 0
- Κάθε διεργασία στέλνει στην επόμενη το άθροισμα της τοπικής τιμής με την τιμή που έλαβε από την προηγούμενη διεργασία. Η πρώτη διεργασία απλά στέλνει την τοπική της τιμή.

Ένα παράδειγμα εκτέλεσης φαίνεται παρακάτω:



Όλες οι διεργασίες καλούν μαζί την MPI_Exscan_pt2pt. Αρχικά, όλες οι διεργασίες εκτός της 0 πρέπει να μπλοκάρουν περιμένοντας να λάβουν την τιμή prev από την προηγούμενη διεργασία στην αλυσίδα. Επομένως, κάνουν Recv:

```
if (rank != 0)
    MPI_Recv(&prev, 1, MPI_INT, rank - 1, MPI_ANY_TAG ...);
```

Η διεργασία 0, όπως εξηγήσαμε, αρχικοποιεί την τιμή prev σε 0. Κάθε διεργασία που έχει λάβει τιμή αναθέτει τη μεταβλητή εξόδου στην τιμή που έλαβε:

```
*out = prev;
```

Έπειτα, χρειάζεται να στείλουμε τιμή στην επόμενη διεργασία στη σειρά, ώστε να προχωρήσει με την εκτέλεσή της. Επομένως, κάθε διεργασία πλην της τελευταίας κάνει Send την τιμή που περιγράψαμε προηγουμένως:

```
int next = prev + *in;
MPI_Send(&next, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
```

Για να εξασφαλίσουμε ότι παράγουμε ορθά αποτελέσματα, κάθε διεργασία εκτελεί πίσω στη main την κανονική MPI_Exscan, ώστε να συγκρίνει τα αποτελέσματα των δύο υλοποιήσεων. Κάθε διεργασία ανιχνεύει το λάθος, αν έχει συμβεί, μέσω κατάλληλου flag. Έπειτα ο root συγκεντρώνει όλα τα επιμέρους flag σε έναν πίνακα και εκτυπώνει μήνυμα λάθους στον χρήστη.

Παρακάτω φαίνεται ένα παράδειγμα εκτέλεσης με τυχαίες τιμές εισόδου για 8 διεργασίες:

```
zoukos@aSUS$ mpiexec -n 8 ./ask1a
Processes: 8

Process: 00 Indata: 1 Result: 0
Process: 01 Indata: 3 Result: 1
Process: 02 Indata: 8 Result: 4
Process: 03 Indata: 6 Result: 12
Process: 04 Indata: 6 Result: 18
Process: 05 Indata: 2 Result: 24
Process: 06 Indata: 6 Result: 26
Process: 07 Indata: 5 Result: 32

Successful verification
```

Υλοποίηση Exscan στο υβριδικό μοντέλο MPI + OpenMP

Η exscan που θα υλοποιήσουμε για το υβριδικό μοντέλο λειτουργεί ως εξής:

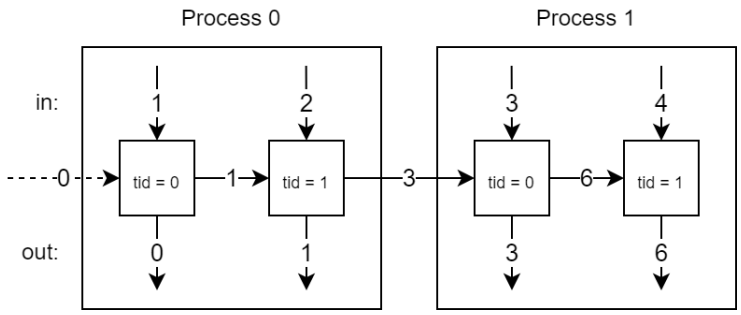
- Καλείται από κάθε νήμα, κάθε διεργασίας, ταυτόχρονα
- Κάθε νήμα δίνει σαν είσοδο μια τοπική τιμή **in**
- Πρέπει στο τέλος της διαδικασίας να έχει αποκτήσει μια άλλη τιμή **out**
- Υπάρχει αλυσίδα από νήματα. Εντός κάθε διεργασίας, τα νήματα έχουν προτεραιότητα που ορίζεται από το thread_num. Οι διεργασίες έχουν προτεραιότητα που ορίζεται από το rank τους. Δεδομένου ότι όλες οι διεργασίες

έχουν το ίδιο πλήθος νημάτων, κάθε νήμα έχει ένα μοναδικό αναγνωριστικό στο παράλληλο πρόγραμμα που δίνεται από την παρακάτω σχέση. Η τιμή αυτή αποτελεί την καθολική προτεραιότητα του νήματος:

$$global\_tid = rank \cdot num\_threads + thread\_num$$

- Κάθε νήμα δίνει σαν έξοδο την τιμή που έλαβε από το προηγούμενο με εξαίρεση το πρώτο νήμα που δίνει έξοδο 0
- Κάθε νήμα στέλνει στο επόμενο το άθροισμα της τοπικής τιμής με την τιμή που έλαβε από το προηγούμενο. Το πρώτο νήμα απλά στέλνει την τοπική του τιμή.

Ένα παράδειγμα εκτέλεσης φαίνεται παρακάτω:



Σε σχέση με πριν, το πρόβλημα τώρα έχει δύο διαστάσεις. Κάθε διεργασία θα έχει μια καθολική τιμή εισόδου την οποία λαμβάνει από την προηγούμενη διεργασία και προωθεί στο πρώτο νήμα, και μια καθολική τιμή εξόδου, η οποία θα είναι η τιμή του τελευταίου νήματος και στέλνεται προς την επόμενη διεργασία. Η διαδικασία ξεκινάει από την πρώτη διεργασία. Κάθε νήμα αυτής της διεργασίας θα χρειαστεί να υπολογίσει για τον εαυτό του την τιμή του Exscan. Επειδή τα νήματα ανήκουν στην ίδια διεργασία, και άρα έχουν κοινή μνήμη, ορίζουμε αρχικά έναν πίνακα με τόσες θέσεις όσα τα νήματα. Κάθε νήμα θα τοποθετήσει την τιμή του στην αντίστοιχη θέση:

```
#pragma omp single
process_data = (int *)malloc(omp_get_num_threads() * sizeof(int));

process_data[threadnum] = in;

#pragma omp barrier
```

Πριν μπορέσουμε να συνεχίσουμε, θα χρειαστεί το πρώτο νήμα της διεργασίας να διαβάσει την καθολική τιμή εισόδου από την προηγούμενη διεργασία. Αυτό θα αποτελεί το καθολικό offset για τα νήματα εντός της διεργασίας. Η πρώτη διεργασία δεν εκτελεί αυτό το Recv και θα έχει offset 0:

```
if (rank != 0 && threadnum == 0)
    MPI_Recv(&prev, 1, MPI_INT, rank - 1, MPI_ANY_TAG, ...);
```

Για το exscan εντός της διεργασίας αρκεί απλά κάθε νήμα να προσθέσει στο offset τις τιμές του πίνακα που βρίσκονται πριν από τη θέση του, χωρίς να υπάρχει ανάγκη για κάποια περαιτέρω επικοινωνία μεταξύ των νημάτων:

```
*out = prev;

for (int i = 0; i < threadnum; i++)
    *out += process_data[i];
```

Το τελευταίο νήμα κάθε διεργασίας (εκτός από την τελευταία) πρέπει να στείλει την τιμή του στο πρώτο νήμα της επόμενης διεργασίας:

```
if (rank != size - 1 && threadnum == omp_get_num_threads() - 1)
{
    int next = *out + process_data[threadnum];
    MPI_Send(&next, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
}
```

Αφού ολοκληρωθεί η διαδικασία του Exscan, κάθε νήμα τοποθετεί την αρχική και την υπολογισμένη τιμή του σε πίνακες της διεργασίας του. Οι αντίστοιχοι πίνακες απ' όλες τις διεργασίες συγκεντρώνονται (Gather) στον root, ώστε με σταδιακή συσσώρευση των τιμών εισόδου αυτός να επαληθεύσει την εγκυρότητα των αποτελεσμάτων

Παρακάτω φαίνεται ένα παράδειγμα εκτέλεσης με τυχαίες τιμές εισόδου για 4 διεργασίες με 4 νήματα:

```
zoukos@aSUS$ mpiexec -n 4 ./ask1b 4
Processes: 4
Threads: 4

Process: 00 Thread: 00 Indata: 2 Result: 0
Process: 00 Thread: 01 Indata: 2 Result: 2
Process: 00 Thread: 02 Indata: 5 Result: 4
Process: 00 Thread: 03 Indata: 9 Result: 9

Process: 01 Thread: 00 Indata: 2 Result: 18
Process: 01 Thread: 01 Indata: 5 Result: 20
Process: 01 Thread: 02 Indata: 1 Result: 25
Process: 01 Thread: 03 Indata: 5 Result: 26

Process: 02 Thread: 00 Indata: 4 Result: 31
Process: 02 Thread: 01 Indata: 7 Result: 35
Process: 02 Thread: 02 Indata: 1 Result: 42
Process: 02 Thread: 03 Indata: 8 Result: 43

Process: 03 Thread: 00 Indata: 6 Result: 51
Process: 03 Thread: 01 Indata: 7 Result: 57
Process: 03 Thread: 02 Indata: 8 Result: 64
Process: 03 Thread: 03 Indata: 7 Result: 72

Successful verification
```

Παράλληλη εγγραφή μητρώων ίδιου μεγέθους

Κάθε νήμα παράγει ένα τρισδιάστατο μητρώο με τυχαίες πραγματικές τιμές στο εύρος 0-1000. Για να εξασφαλίσουμε τυχαιότητα, κάθε thread αρχικοποιεί ένα ιδιωτικό seed που προκύπτει από το καθολικό αναγνωριστικό του:

```
srand(time(NULL) + rank * omp_get_num_threads() +
omp_get_thread_num());
```

Σκοπός είναι να γράψουμε όλα αυτά τα παράλληλα μητρώα σε ένα κοινό δυαδικό αρχείο. Όλα οι διεργασίες πρέπει να γράφουν παράλληλα στο αρχείο χρησιμοποιώντας συναρτήσεις I/O του MPI. Κάθε νήμα κάθε διεργασίας μπορεί να γράφει ανεξάρτητα από τα άλλα νήματα. Οι πίνακες των νημάτων πρέπει να γραφτούν σύμφωνα με την καθολική διάταξη των νημάτων. Επομένως, για να ξέρει κάθε νήμα πού να ξεκινήσει την εγγραφή, πρέπει πρώτα να γνωρίζει πόσο έχουν γράψει όλα τα προηγούμενα νήματα

της υβριδικής εφαρμογής. Για τον σκοπό αυτό θα χρησιμοποιήσουμε την Exscan που υλοποιήσαμε προηγουμένως για το υβριδικό μοντέλο.

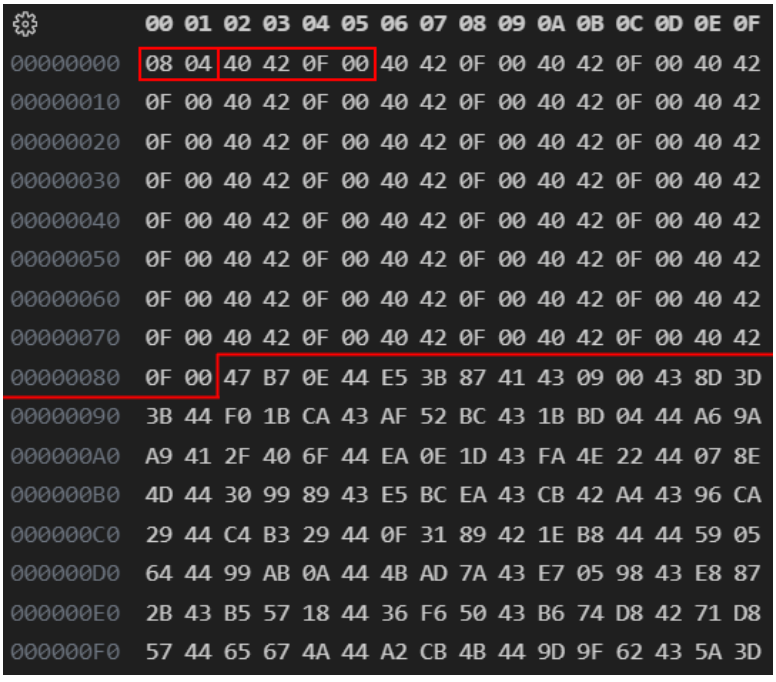
Αρχικά, το βασικό νήμα της πρώτης διεργασίας δημιουργεί το κοινό αρχείο και γράφει σ' αυτό κάποια μεταδεδομένα που αφορούν το περιβάλλον στο οποίο έγινε η εγγραφή. Συγκεκριμένα, γράφουμε:

- Το πλήθος των διεργασιών που χρησιμοποιήθηκαν (byte)
- Το πλήθος των νημάτων ανά διεργασία (byte)
- Έναν πίνακα ακεραίων (int) που σε κάθε θέση περιέχει το πλήθος στοιχείων που έγραψε το αντίστοιχο νήμα. Εδώ γνωρίζουμε από πριν ότι όλα τα νήματα γράφουν το ίδιο πλήθος στοιχείων

Έπειτα ξεκινάμε την παράλληλη περιοχή, όπου γίνεται η αρχικοποίηση των μητρώων. Γνωρίζοντας πόσο θα γράψει το κάθε νήμα για το μητρώο του, καλούν όλα τα νήματα την MPI_Exscan_omp με είσοδο το πλήθος στοιχείων (N^3). Θα λάβουν σαν έξοδο τη θέση όπου πρέπει να γίνει η εγγραφή. Επειδή η exscan ουσιαστικά θα δώσει για κάθε νήμα σαν έξοδο το άθροισμα των τιμών όλων των προηγούμενων νημάτων στη διάταξη, με την κλήση αυτή κάθε νήμα μαθαίνει πού πρέπει να γράψει στο αρχείο (ξεκινώντας μετά από το header που γράψαμε είδη). Έπειτα όλα τα νήματα κάνουν παράλληλα εγγραφή με την MPI_File_write_at.

Για να επιβεβαιώσομε ότι η εγγραφή έγινε επιτυχώς, κάθε νήμα έπειτα διαβάζει τις τιμές που έγραψε σε ένα ξεχωριστό buffer με την MPI_File_read_at. Έπειτα συγκρίνουμε τις αρχικές τιμές με τις τιμές του buffer για να σιγουρευτούμε ότι η εγγραφή και η ανάγνωση έγινε επιτυχώς.

Παρακάτω φαίνεται το αρχείο που προκύπτει μετά την εκτέλεση του προγράμματος με 8 διεργασίες, 4 νήματα ανά διεργασία και N=100. Παρατηρούμε ότι στα πρώτα δύο bytes γράφονται το πλήθος των διεργασιών και τα νήματα. Έπειτα ακολουθούν 32 ακέραιοι που δείχνουν πόσο έγγραψε κάθε νήμα. Εδώ γράφεται σε κάθε θέση το 0x000F4240, το οποίο είναι 1.000.000, αφού όλοι γράφουν το ίδιο. Μετά από την κεφαλίδα, γράφονται τα δεδομένα του αρχείου:



	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	08	04	40	42	0F	00	40	42	0F	00	40	42	0F	00	40	42
00000010	0F	00	40	42	0F	00	40	42	0F	00	40	42	0F	00	40	42
00000020	0F	00	40	42	0F	00	40	42	0F	00	40	42	0F	00	40	42
00000030	0F	00	40	42	0F	00	40	42	0F	00	40	42	0F	00	40	42
00000040	0F	00	40	42	0F	00	40	42	0F	00	40	42	0F	00	40	42
00000050	0F	00	40	42	0F	00	40	42	0F	00	40	42	0F	00	40	42
00000060	0F	00	40	42	0F	00	40	42	0F	00	40	42	0F	00	40	42
00000070	0F	00	40	42	0F	00	40	42	0F	00	40	42	0F	00	40	42
00000080	0F	00	47	B7	0E	44	E5	3B	87	41	43	09	00	43	8D	3D
00000090	3B	44	F0	1B	CA	43	AF	52	BC	43	1B	BD	04	44	A6	9A
000000A0	A9	41	2F	40	6F	44	EA	0E	1D	43	FA	4E	22	44	07	8E
000000B0	4D	44	30	99	89	43	E5	BC	EA	43	CB	42	A4	43	96	CA
000000C0	29	44	C4	B3	29	44	0F	31	89	42	1E	B8	44	44	59	05
000000D0	64	44	99	AB	0A	44	4B	AD	7A	43	E7	05	98	43	E8	87
000000E0	2B	43	B5	57	18	44	36	F6	50	43	B6	74	D8	42	71	D8
000000F0	57	44	65	67	4A	44	A2	CB	4B	44	9D	9F	62	43	5A	3D

Παράλληλη εγγραφή μητρώων διαφορετικού μεγέθους

Επαναλαμβάνουμε την ίδια διαδικασία, αλλά τώρα κάθε τυχαίο μητρώο που παράγεται συμπιέζεται πριν γραφτεί στο αρχείο. Αυτό έχει ως αποτέλεσμα το ότι τα δεδομένα που γράφουν τα νήματα πιθανότατα έχουν διαφορετικά μεγέθη. Για τη συμπίεση χρησιμοποιήθηκε η βιβλιοθήκη ZFP.

Όπως και πριν, γράφουμε στην αρχή του αρχείου ένα header με πληροφορίες. Το header αυτό διαφέρει ελαφρώς από το προηγούμενο. Συγκεκριμένα, περιέχει:

- Το πλήθος των διεργασιών που χρησιμοποιήθηκαν (byte)
- Το πλήθος των νημάτων ανά διεργασία (byte)
- Έναν πίνακα ακεραίων (int) που σε κάθε θέση, αντί για το πλήθος float που γράφονται από το νήμα, περιέχει το πλήθος των bytes, καθώς μετά τη συμπίεση παύουν να υφίστανται float στοιχεία, και απλά εργαζόμαστε με bytes. Προφανώς σε αντίθεση με πριν, η πληροφορία αυτή δεν μπορεί να γραφτεί παρά μόνο από το νήμα της παράλληλης περιοχής που έκανε τη συμπίεση

Μέσα στην παράλληλη περιοχή, αρχικοποιούμε τα μητρώα όπως και πριν και έπειτα τα συμπιέζουμε με τη συνάρτηση compress_array. Η συνάρτηση αυτή ορίζει ένα τρισδιάστατο ZFP πεδίο (zfp_field) αριθμών κινητής υποδιαστολής, το οποίο αναπαριστά τον προς συμπίεση πίνακα. Έπειτα δημιουργεί μια ροή δεδομένων (zfp_stream) η οποία περιέχει τις παραμέτρους της συμπίεσης. Εδώ μπορούμε να ρυθμίσουμε κατάλληλα την επιθυμητή ακρίβεια, μέσω του μέγιστου επιτρεπόμενου σφάλματος, εξισορροπώντας έτσι τον ρυθμό συμπίεσης και την ακρίβεια των αποτελεσμάτων. Αφού εκτιμήσουμε το μέγεθος του συμπιεσμένου πίνακα, δεσμεύουμε την απαιτούμενη μνήμη, δημιουργούμε ένα bitstream για εγγραφή δεδομένων πάνω στη μνήμη και λέμε στο ZFP να χρησιμοποιήσει τη μνήμη για εγγραφή του συμπιεσμένου μητρώου. Τέλος, η συμπίεση γίνεται με τη συνάρτηση zfp_compress, η οποία επιστρέφει και το μέγεθος του συμπιεσμένου μητρώου σε bytes.

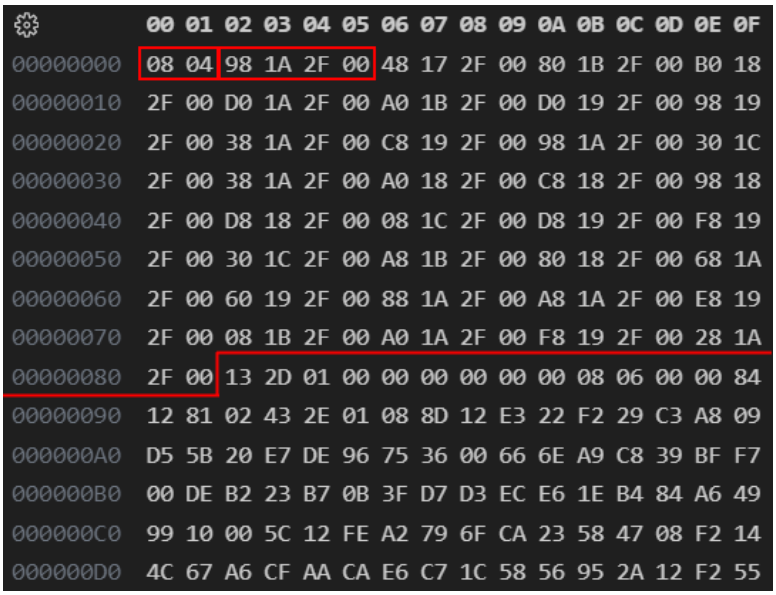
Αφού κάθε νήμα συμπιέσει το μητρώο, γράφουμε τα νέα μεγέθη στο header του αρχείου και έπειτα καλούμε την MPI_Exscan_omp για να μάθουμε, όπως και πριν, πού πρέπει κάθε νήμα να ξεκινήσει την εγγραφή του μητρώου. Έπειτα όλα τα νήματα κάνουν παράλληλα εγγραφή με την MPI_File_write_at.

Για να επιβεβαιώσομε ότι η εγγραφή έγινε επιτυχώς, κάθε νήμα έπειτα διαβάζει τις τιμές που έγγραψε με την MPI_File_read_at. Για να ανακτήσουμε τα αρχικά μητρώα, περνάμε το buffer ανάγνωσης στη συνάρτηση decompress_array, η οποία εκτελεί παρόμοια λειτουργία με την compress_array, αλλά για την αποσυμπίεση. Αφού επιστραφούν τα αποσυμπιεσμένα μητρώα, συγκρίνουμε τις τιμές με τις αρχικές. Επειδή υπάρχει πιθανότητα να υπάρχει κάποιο σφάλμα μεταξύ των αποσυμπιεσμένων αριθμών και των αρχικών, η σύγκριση γίνεται υποθέτοντας ένα μέγιστο επιτρεπτό σφάλμα.

Παρακάτω φαίνεται ένα παράδειγμα εκτέλεσης για N = 100 και 8 διεργασίες με 4 νήματα. Παρατηρούμε ότι το αρχικό μέγεθος κάθε μητρώου είναι 4.000.000 bytes, ενώ μετά τη συμπίεση το μέσο μέγεθος έχει μειωθεί στα 3.086.909 bytes. Γενικά, για να δούμε έναν ικανοποιητικό λόγο συμπίεσης, θα πρέπει το μέγεθος των αρχικών μητρώων να είναι αρκετά μεγάλο:

```
zoukos@aSUS$ mpiexec -n 8 ./ask1d 4 100
Array size: 4000000 bytes
Average Compression Size: 3086909.00 bytes
Successful verification.
```

Παρακάτω φαίνεται και τμήμα του αρχείου μετά την εγγραφή. Όπως και πριν, στα πρώτα δύο bytes γράφονται το πλήθος των διεργασιών και τα νήματα. Έπειτα ακολουθούν 32 ακέραιοι που δείχνουν πόσο έγγραψε κάθε νήμα. Σε αντίθεση με πριν, οι αριθμοί εδώ είναι διαφορετικοί, καθώς κάθε νήμα γράφει το συμπιεσμένο μητρώο, και υποδεικνύουν bytes αντί για πλήθος floats. Μετά από την κεφαλίδα, γράφονται τα δεδομένα του αρχείου:



	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	08	04	98	1A	2F	00	48	17	2F	00	80	1B	2F	00	B0	18
00000010	2F	00	D0	1A	2F	00	A0	1B	2F	00	D0	19	2F	00	98	19
00000020	2F	00	38	1A	2F	00	C8	19	2F	00	98	1A	2F	00	30	1C
00000030	2F	00	38	1A	2F	00	A0	18	2F	00	C8	18	2F	00	98	18
00000040	2F	00	D8	18	2F	00	08	1C	2F	00	D8	19	2F	00	F8	19
00000050	2F	00	30	1C	2F	00	A8	1B	2F	00	80	18	2F	00	68	1A
00000060	2F	00	60	19	2F	00	88	1A	2F	00	A8	1A	2F	00	E8	19
00000070	2F	00	08	1B	2F	00	A0	1A	2F	00	F8	19	2F	00	28	1A
00000080	2F	00	13	2D	01	00	00	00	00	00	00	08	06	00	00	84
00000090	12	81	02	43	2E	01	08	8D	12	E3	22	F2	29	C3	A8	09
000000A0	D5	5B	20	E7	DE	96	75	36	00	66	6E	A9	C8	39	BF	F7
000000B0	00	DE	B2	23	B7	0B	3F	D7	D3	EC	E6	1E	B4	84	A6	49
000000C0	99	10	00	5C	12	FE	A2	79	6F	CA	23	58	47	08	F2	14
000000D0	4C	67	A6	CF	AA	CA	E6	C7	1C	58	56	95	2A	12	F2	55

Ερώτημα 2: Παράλληλη αναζήτηση παραμέτρων στη Μηχανική Μάθηση

Για το ερώτημα αυτό τροποποιήσαμε τον αρχικό κώδικα ώστε να υπάρχει η δυνατότητα να λαμβάνει διαφορετικά δεδομένα για το πρόβλημα (samples, features, τιμές υπερπαραμέτρων) από το terminal ως arguments αλλά και να έχει προεπιλεγμένες τιμές εάν αυτές δεν δοθούν. Επίσης εκτυπώνουμε τον χρόνο που χρειάστηκε κάθε διεργασία να ολοκληρωθεί. Την ίδια διαδικασία ακολουθήσαμε και για τις υπόλοιπες τρεις υλοποιήσεις.

Υλοποίηση Multiprocessing Pool

Σε αυτή την υλοποίηση χρησιμοποιήσαμε το multiprocessing πακέτο της Python, το οποίο δημιουργεί διεργασίες οι οποίες εκτελούν tasks με την ίδια συνάρτηση test_params και παραμέτρους τους διαφορετικούς συνδυασμούς που δημιουργήθηκαν από την ParameterGrid. Εκτός από τα δεδομένα του προβλήματος, μπορούν να οριστούν και πόσες διεργασίες θα δημιουργηθούν ως arguments στο terminal. Κάθε φορά που μια διεργασία τερματίζει, της ανατίθεται ένας καινούργιος συνδυασμός, αν υπάρχει, και τα αποτελέσματα της αποθηκεύονται στην μεταβλητή results.

Υλοποίηση MPI Futures

Εδώ χρησιμοποιήσαμε το πακέτο MPI4py για να παραλληλοποιήσουμε την παραμετρική αναζήτηση. Αυτό το πακέτο πρέπει να εκτελεστεί μέσω του terminal μέσω της εντολής mpiexec όπου ορίζουμε και πόσες διεργασίες θα δημιουργηθούν και θα εκτελούν το πρόγραμμα. Αρχικά προετοιμάζουμε τα δεδομένα του προβλήματος και η διεργασία 0 δημιουργεί τους πιθανούς συνδυασμούς. Έπειτα χρησιμοποιούμε το MPIPoolExecutor με το οποίο η master διεργασία έχει πρόσβαση στο MPIPoolExecutor και υποβάλει τις διεργασίες στους workers, κάνοντας map τη συνάρτηση test_params στις διαφορετικές παραμέτρους. Η master διεργασία δεν παίρνει μέρος στην εκτέλεση των διεργασιών. Αν το script το εκτελεί μόνο μια διεργασία δεν έχουμε παραλληλισμό όπως και με δύο διεργασίας αφού η master θα περιμένει την worker να τελειώσει για να της δώσει την επόμενη διεργασία κάτι που φαίνεται και στο διάγραμμα παρακάτω.

Υλοποίηση μοντέλου Master-Worker με χρήση MPI

Για αυτή την υλοποίηση δημιουργήσαμε μια δική μας εκδοχή του Master-Worker μοντέλου χρησιμοποιώντας το πακέτο MPI4py και φτιάχνοντας κλάσεις για τον Master και τον Worker. Για να τρέξει το Script χρειάζεται τουλάχιστον να εκτελεστεί με δύο διεργασίες. Η διεργασία 0 δημιουργεί ένα στιγμιότυπο της κλάσης Master, ενώ όλες οι άλλες είναι Worker.

Όπως και πριν, η διεργασία 0 δημιουργεί όλους τους συνδυασμούς που πρέπει να εκτελεστούν. Για κάθε συνδυασμό δημιουργεί ένα στιγμιότυπο της κλάσης Work, τα οποία τοποθετεί σε μία ουρά. Με τη μέθοδο «distribute_work» ο Master εκτελεί την αρχική διαμοίραση των εργασιών στους Workers, δίνοντας μια εργασία σε κάθε Worker:

```
for i in range(1, self.comm.Get_size()):
    self.send_work_to(i)
```

Η μέθοδος send_work_to θα δώσει στον Worker i την επόμενη εργασία στην ουρά. Αν δεν υπάρχει εργασία, τότε απλά επιστρέφει. Ο Master καταγράφει την αποστολή εργασίας, ώστε να ξέρει κάθε χρονική στιγμή πόσοι απασχολημένοι Worker υπάρχουν:

```
work = self._get_next_request()
if work is None: return
comm.isend(work, rank, WORKTAG)
self.sent_requests += 1
```

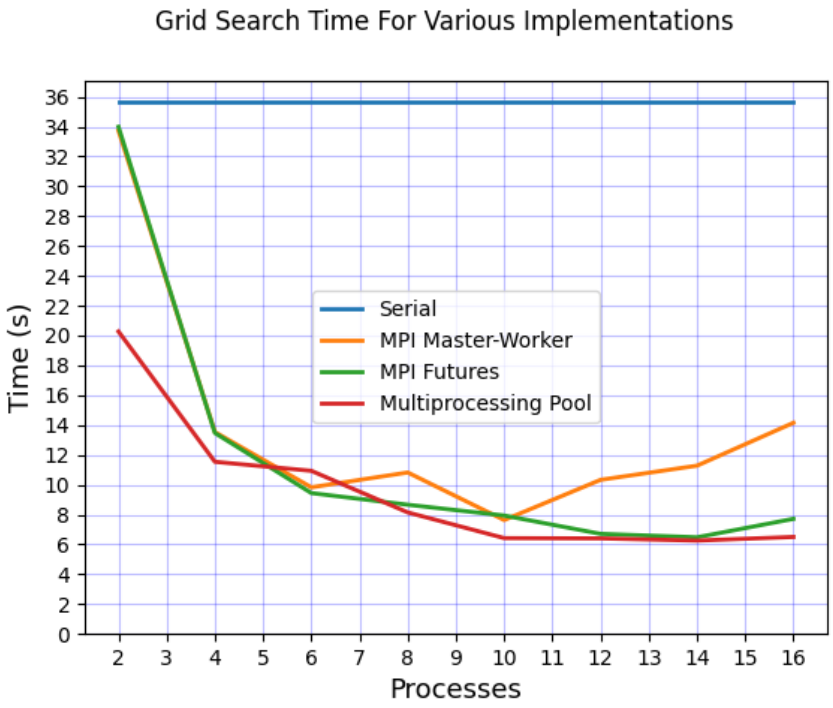
Οι Workers περιμένουν με MPI_Recv να τους σταλεί κάποια εργασία (μέθοδος «work») μέσα από τον communicator, την εκτελούν και επιστρέφουν τα αποτελέσματα στην Master διεργασία. Παράλληλα, η κύρια διεργασία έχει καλέσει τη μέθοδο «get_results», με την οποία μπλοκάρει στο MPI_Recv μέχρι ένας οποιοσδήποτε Worker να τελειώσει:

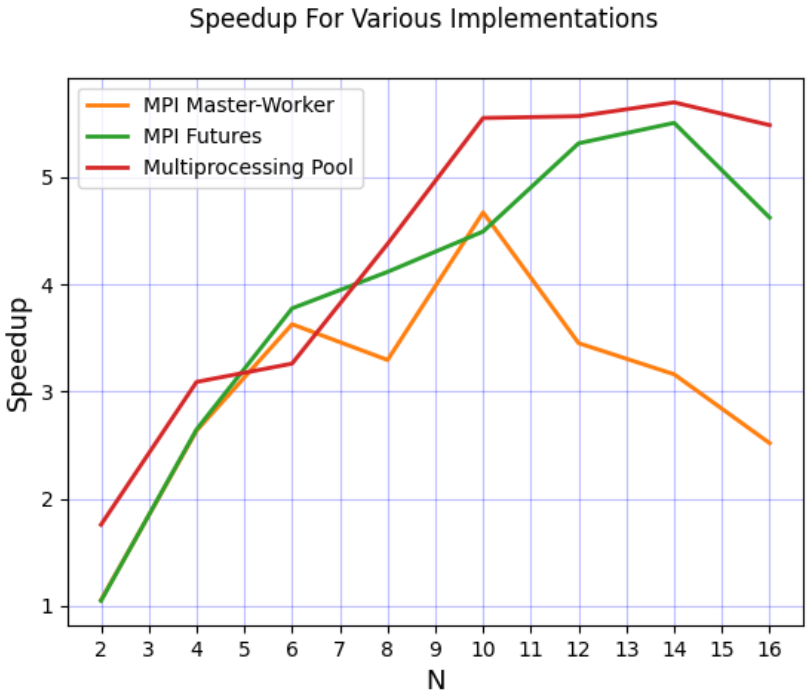
```
res = self.comm.recv(source=MPI.ANY_SOURCE, tag=MPI.ANY_TAG,
status=status)
```

Μόλις ένας Worker τελειώσει, ο Master λαμβάνει τα αποτελέσματα και με την συνάρτηση «send_work_to» στέλνει στον ίδιο Worker την επόμενη διαθέσιμη εργασία. Αυτή η διαδικασία εκτελείται μέχρι να τελειώσουν όλοι οι συνδυασμοί της παραμετρικής αναζήτησης. Τέλος εκτυπώνουμε τα αποτελέσματα της παραμετρικής αναζήτησης.

Μετρήσεις

Για τις μετρήσεις των τριών διαφορετικών υλοποιήσεων έχουμε ένα Python Script (plot.py) το οποίο τα εκτελεί για τα ίδια δεδομένα, αλλά με αυξανόμενο αριθμό διεργασιών ανά δύο, από 2 έως 16, και φτιάχνει δύο διαγράμματα. Στο πρώτο φαίνεται οι χρόνοι εκτέλεσης διεργασιών και στο δεύτερο το speedup για κάθε αριθμό επεξεργαστικών στοιχείων.





Από τα παραπάνω διαγράμματα φαίνεται ότι η καλύτερη υλοποίηση είναι αυτή με το Multiprocessing Pool αφού βλέπουμε να έχει την καλύτερη επιτάχυνση από τα υπόλοιπα.

Οι υλοποιήσεις MPI με Master-Worker και Futures βλέπουμε ότι για δύο διεργασίες έχουν ελάχιστη επιτάχυνση σε σχέση με την σειριακή αφού μόνο μια διεργασία εκτελεί τους υπολογισμούς για την παραμετρική αναζήτηση. Το μηχάνημα που εκτελούμε τα script έχει επεξεργαστή με 8 πυρήνες και 16 threads (hyperthreading) με αποτέλεσμα να βλέπουμε χρονοβελτίωση μέχρι τις 10 ταυτόχρονες διεργασίες και μετά να μένει σταθερή ή να μειώνεται. Αυτό μπορεί να συμβαίνει λόγο του overhead στην διαχείριση των threads όπως context switching και synchronization. Ειδικά για την υλοποίηση Master-Worker, η οποία είναι και η πιο αργή, αναμένουμε να υπάρχει και overhead λόγω των Send και Receive (συγχρονισμός Master-Worker) που εκτελούνται μεταξύ των διεργασιών.

Ερώτημα 3: OpenMP tasking

Με (dynamic, 2), οι επαναλήψεις χωρίζονται σε ομάδες συνεχών επαναλήψεων μεγέθους 2 και τοποθετούνται σε μια ουρά. Αρχικά ανατίθεται σε κάθε νήμα μια απ' αυτές τις ομάδες επαναλήψεων. Μόλις ένα νήμα ολοκληρώσει τις επαναλήψεις του, ανακτά μια άλλη ομάδα από την ουρά. Η διαδικασία συνεχίζεται μέχρι να εξαντληθούν όλες οι ομάδες της ουράς

Επειδή σε κάθε επανάληψη εκτελούμε την ίδια συνάρτηση work, αλλά με διαφορετική παράμετρο i, το πρόγραμμα αυτό μπορεί να επωφεληθεί από το μοντέλο tasking του OpenMP. Μέσα στον επαναληπτικό βρόχο, ένα μόνο νήμα δημιουργεί όλα τα tasks που πρέπει να εκτελεστούν. Τα tasks αυτά τοποθετούνται σε μια ουρά και εκτελούνται από κάποιο νήμα όταν αυτό είναι ελεύθερο. Για να θεωρείται η υλοποίηση με task ισοδύναμη με πριν, θα πρέπει κάθε task να περιλαμβάνει δύο συνεχόμενες επαναλήψεις. Ιδιαίτερη προσοχή απαιτεί η περίπτωση όπου το N είναι περιττό, καθώς τότε το τελευταίο task θα πρέπει να περιλαμβάνει μόνο μια επανάληψη. Παρακάτω φαίνεται η ισοδύναμη υλοποίηση με tasks:

```
#pragma omp parallel
#pragma omp single
{
    for (int i = 0; i < N; i+=2)
    {
        #pragma omp task firstprivate(i) shared(A)
        {
            A[i] = work(i);

            if (i < N-1)
                A[i+1] = work(i+1);
        }
    }
}
```

Περιβάλλον Υλοποίησης

Παραθέτουμε πληροφορίες για το περιβάλλον όπου υλοποιήθηκαν οι λύσεις της εργαστηριακής άσκησης:

Χαρακτηριστικό	**Τιμή**
Operating System	Microsoft Windows 11 Home (x64), Version 24H2, Build 26100.3037 WSL (Windows Subsystem for Linux): Ubuntu 24.04 LTS
CPU model	AMD Ryzen 7 8845HS @ 3.8 GHz
Physical CPU cores	8
Logical CPU cores	16
RAM	32 GB DDR5
GPU	NVIDIA GeForce RTX 4060 Laptop GPU
VRAM	8188 MB
L1 data cache	8 x 32 KB (8-way, 64-byte line)
L2 cache	8 x 1024 KB (8-way, 64-byte line)
L3 cache	16 MB (16-way, 64-byte line)
Instruction Sets	MMX (+), SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4A, x86-64, AES, AVX, AVX2, AVX512 (DQ, BW, VL, CD, IFMA, VBMI, VBMI2, VNNI, BITALG, VPOPCNTDQ, BF16), FMA3, SHA