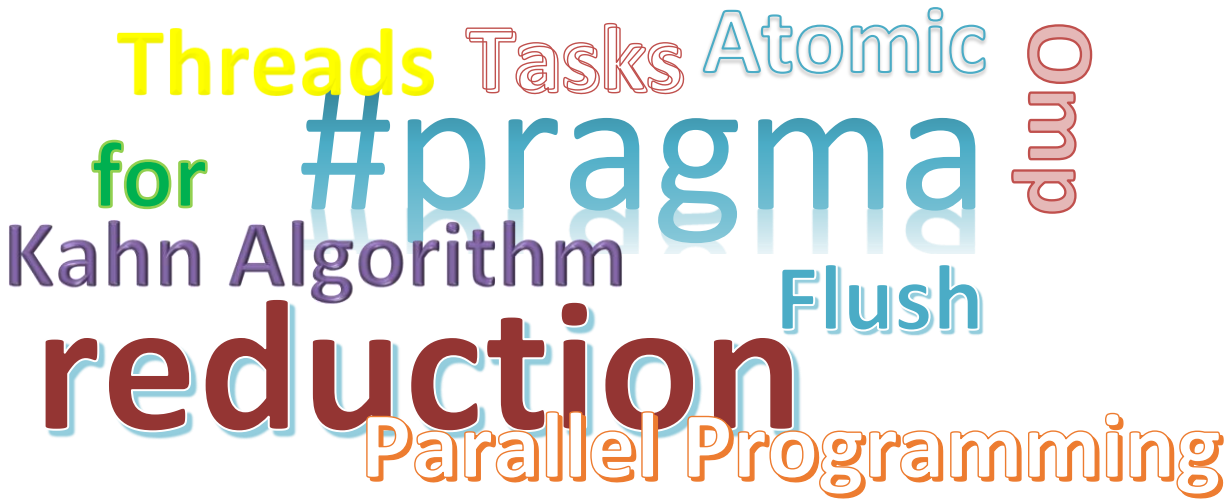
ΠΑΡΑΛΛΗΛΗ ΕΠΕΞΕΡΓΑΣΙΑ

ΕΞΑΜΗΝΙΑΙΑ ΟΜΑΔΙΚΗ ΕΡΓΑΣΙΑ



ΜΕΛΗ ΟΜΑΔΑΣ 41 :

ΑΛΕΞΙΟΥ ΣΤΑΥΡΟΣ ΑΜ 1059680

ΑΦΕΝΤΑΚΗ ΦΛΩΡΕΝΤΙΑ ΑΜ 1059576

ΚΟΤΣΙΜΠΟΥ ΦΩΤΕΙΝΗ ΑΜ 1059567

ΣΙΓΟΥΡΟΥ ΑΛΚΗΣΤΙΣ ΑΙΚΑΤΕΡΙΝΗ ΑΜ 1059661

Υλοποίηση της main

Στα παρακάτω πλαίσια σας παρουσιάζουμε την υλοποίηση της main. Ο κώδικας μας χωρίζεται σε 4 βασικά βήματα:

1) Διάβασμα txt αρχείου με γράφημα σε μορφή mtx και δημιουργία του πίνακα γειτνίασης.
2) Εκτύπωση του πίνακα γειτνίασης.
3) Δημιουργία των λιστών S και L και ενημέρωση της S.
4) Εκτέλεση του Kahn αλγόριθμου.

```
int main()
{
        int delNode; //ο κόμβος που διαγράφουμε από τον πίνακα γειτνίασης
         int i,j,z;

        FILE * fp;

     ////////////// STEP 1 => Διαβάζουμε το αρχείο και το τοποθετούμε στον  πίνακα γειτνίασης ///////

         fp = fopen ("dag323.txt", "r");
        if(fp == NULL)
         {
                printf ("Error opening file\n");
                 exit(1);
        }

         fscanf(fp, "%d", &V); //Θέτει την V ίση με το πλήθος των κόμβων

         adjMatrix = (int **)malloc((V+1) * sizeof(int *));

        for ( i=0; i<=(V+1); i++)
        adjMatrix[i] = (int *)malloc(V * sizeof(int)); //Δυναμική δέσμευση του πίνακα

         init();
        fscanf(fp, "%d %d", &j, &z); //Προσπερνάει τις επόμενες δύο μεταβλητές για να φτάσει στις ακμές

        while(fscanf(fp, "%d %d", &i, &j) != EOF) //Διαβάζει τις γραμμές του αρχείου txt για να πάρει τις ακμές
        {

                addEdge(i,j); // Προσθέτει την ακμή που διάβασε στον πίνακα γειτνίασης

        }
        fclose(fp);


       ////////////// STEP 2 => Εκτυπώνω τον πίνακα γειτνίασής που δημιουργήσαμε ////////////////


        printf("------------------------------------------------------------------ \n\n ");
        printf("\n O dothen pinakas geitniasis einai: \n\n ");
        printAdjMatrix();
        printID();
        printf("------------------------------------------------------------------\n\n");
```

```
////////////// STEP 3 =>Δημιουργούμε τις λίστες L, S και ενημερώνουμε την S ////////////////

  struct timespec start, finish ;
  clock_gettime (CLOCK_REALTIME, &start); //Αρχή χρονομέτρου


                                  //lista L==> Λίστα τοπολογικής διάταξης
  list * headL = NULL;
  headL = (list *) malloc(sizeof(list));
  if (headL == NULL)
  {
            return 1;
  }
                                        //lista S==> Λίστα με indegree=0
  list * headS = NULL;
  headS = (list *) malloc(sizeof(list));
  if (headS == NULL)
  {
           return 1;
  }



  headS->next = NULL;    //Αρχικοποίηση των head
  headL->next = NULL;

  updateS2(adjMatrix,headS,headL);

       /////////////// STEP 4 => Εκτελώ τον αλγόριθμο Kahn ////////////////////////
                        ////// STEP 4.1 => Loop για ενημέρωση S, L ////////

  while(headS->next!=NULL)
  {

            // printList(headS);
            // printf("=S\n\n");
            push(headL,delNode=remove_last(headS));

            // printList(headL);
            // printf("=L\n\n");
            //printf("o komvos poy diagrafw einai %d\n\n\n",delNode);
            updateMatrix(adjMatrix,delNode);
            inDegree(adjMatrix);

            //printID();
            updateS2(adjMatrix,headS,headL);
            // printf("-----------------------------------------------------------------\n\n");
   }

////// STEP 4.2  =>Ελέγχω για ακμές στο γράφημα και εμφανίζω πόρισμα /////////

  if(untitled()!=0)
   {
            printf("To grafima periexei akomi akmes ara kai kiklo \n Den uparxei topologiki diataxi
            gia to sugkekrimeno grafima.\n");
            return(1);
  }
```

```
                    ////// STEP 4.3  =>Τύπωμα της τυπολογικής διάταξης  /////////

        printf("-----------------------------------------------------------------\n\n");
        printf("H topologiki diataxi einai : ");
        printf("\t");
        printList(headL);
        printf("\n\n\n\n");
        printf("Euxaristw :) \n\n\n");


        clock_gettime(CLOCK_REALTIME, &finish);
        long seconds = finish.tv_sec - start.tv_sec; //υπολογισμός του συνολικού χρόνου από την
        δημιουργία
        long ns = finish.tv_nsec - start.tv_nsec; // των λιστών έως την εκτύπωση της τοπολογικής διάταξης
        printf("Total seconds: %lf\n", (double)seconds + (double)ns/(double)1000000000);

        if (start.tv_nsec > finish.tv_nsec)
        {       // clock underflow
                --seconds;
                ns += 1000000000;
        }


//Δημιουργία ενός txt αρχείου που θα κρατάει πρακτικά για τους χρόνους εκτέλεσης μας ,αναλόγως τα
                                                threads
        char sentence[1000];
        // Δημιουργούμε pointer για να διατρέξω το αρχείο
        FILE *fptr;

        // Ανοιξε το αρχείο σε μορφή write
        if((fptr=fopen("time323.txt","a")) == NULL)
        {
                printf("Error!");
                exit(1);
        }

        else
        {
                fprintf (fptr, "Me %d threads o xronos einai %lf  seconds\n",
                NUM_THREADS,(double)seconds + (double)ns/(double)1000000000);
                fclose(fptr);

        }

                //Απελευθερώνουμε την μήνη που είχαμε δεσμεύσει για τον πίνακα
        for(i=0; i<(V+1);i++)
        {
                free(adjMatrix[i]);
        }

        free(adjMatrix);

        return(0);
}
```

Παραλληλοποιημένες συναρτήσεις

Στην συνέχεια παραθέτουμε τις συναρτήσεις που έχουμε παραλληλοποιήσει, με σχετική επεξήγηση για την κάθε μία.

```
//προσθήκη στο τέλος της λίστας

void push(list * head, int name)
{
  #pragma omp parallel shared(head,name)
  {
     #pragma omp single nowait
     {
        list * current = head;
        while (current->next != NULL)
 //διατρέχεται η λίστα , ξεκινώντας από το head μέχρι να ανακαλύψει το τελευταίο στοιχείο της .
        {
           current = current->next;
        }

    //Προσθέτουμε νέα μεταβλητή
      current->next = (list *) malloc(sizeof(list));
      #pragma omp task
      current->next->name = name;
      #pragma omp task
      current->next->next = NULL;
    }
  }
}
```

Επεξήγηση παραλληλοποίησης

Χρησιμοποιούμε το #pragma omp parallel για την έναρξη παράλληλης περιοχής και την δημιουργία νημάτων και θέτουμε ως παράμετρο την εντολή shared(head, name), ώστε όλα μας τα νήματα να μπορούν να κάνουν ταυτόχρονη επεξεργασία στις δύο μεταβλητές. Στη συνέχεια χρησιμοποιούμε την εντολή #pragma omp single nowait, ώστε μόνο ένα νήμα να έχει πρόσβαση στην αλλαγή της θέσης του pointer στην λίστα και με το nowait, επιτρέπουμε στα υπόλοιπα νήματα να προσπεράσουν το προκαθορισμένο barrier και να συνεχίσουν με τις υπόλοιπες συναρτήσεις. Το νήμα που εισέρχεται στην παράλληλη περιοχή δημιουργεί δύο tasks. Το κάθε task, ανατίθεται σε ένα νήμα και «δένεται» με αυτό. Το ένα νήμα είναι υπεύθυνο για την μετάβαση του pointer στο επόμενο στοιχείο και το δεύτερο για την δημιουργία της νέας θέσης.

```
int remove_last(list * head)
 {
   int retval = 0;
   //Αν υπάρχει μόνο ένα στοιχείο στην λίστα  ,αφαίρεσε το
   if (head->next == NULL)
   {
       retval = head->name;
      return -1;
   }

   //Πήγαινε στον προτελευταίο κόμβο
   #pragma omp parallel shared(head)
   {
      #pragma omp single nowait
      {
         list * current = head;
        while (current->next->next != NULL)
        {
          #pragma omp taskwait
           current = current->next;
        }
   //Το current δείχνει στον προτελευταίο στοιχείο της λίστας ,
αφαίρεσε το
       current->next */
      retval = current->next->name;
      free(current->next);
      #pragma omp task
      current->next = NULL;
      }
   }
        return retval;

}
```

Επεξήγηση παραλληλοποίησης

Αντίστοιχα με την push, χρησιμοποιούμε το #pragma omp parallel για την έναρξη παράλληλης περιοχής και την δημιουργία νημάτων και θέτουμε ως παράμετρο την εντολή shared(head), ώστε όλα μας τα νήματα να μπορούν να κάνουν ταυτόχρονη επεξεργασία στην μεταβλητή. Στη συνέχεια χρησιμοποιούμε την εντολή #pragma omp single nowait ώστε μόνο ένα νήμα να έχει πρόσβαση στην αλλαγή της θέσης του pointer στην λίστα και με το nowait, επιτρέπουμε στα υπόλοιπα νήματα να προσπεράσουν το προκαθορισμένο barrier και να συνεχίσουν με τις υπόλοιπες συναρτήσεις. Μέσα στο **while** loop δημιουργούμε ένα taskwait από το ένα νήμα που διατρέχει την συνάρτηση, ώστε να αλλάζει την θέση του pointer, χρησιμοποιούμε το wait για να διασφαλίσουμε την χρονισμένη μεταβολή του pointer για να μην αλλάζει παράλληλα μέσα και έξω από το **while**. Στο τέλος έχουμε άλλο ένα task για να καταργεί την θέση από τον κόμβο που διαγράψαμε.

```
//Υπολογισμός του  inDegree του κάθε κόμβου στον πίνακα γειτνίασης
 void inDegree(int **arr)
{
   int i,j,tempD;
  #pragma omp parallel
  {
    #pragma omp for private(i) reduction(+:tempD) schedule(dynamic)
    for (j=0; j<V; j++)
    //Αφού κάθε γραμμή του πίνακα αναπαριστά
    {
   //όλους τους περιορισμούς ενός κόμβου x
      tempD=0;
      for (i=0; i<V; i++)
      {
       //Τότε και κάθε στήλη θα αναπαριστά όλες τις πηγές (sources X)  με προορισμό έναν κόμβο y
         If (arr[i][j] == 1)
           tempD++;
       }
      arr[V][j] = tempD;
   }
   #pragma omp flush
  }
}
```

Επεξήγηση παραλληλοποίησης

Χρησιμοποιούμε το #pragma omp parallel για την έναρξη παράλληλης περιοχής και την δημιουργία νημάτων

Στην συνέχεια έχουμε την εντολή for για να παραλληλοποιηθεί το επακόλουθο **for** loop και θέτουμε ως παραμέτρους τις εντολές private και schedule με ορίσματα τις μεταβλητές (i) και (dynamic) αντίστοιχα. Αναθέτουμε με δυναμικό τρόπο σε κάθε νήμα κάποια j και με τις private μεταβλητές διασφαλίζουμε την ατομική χρήση του εμφωλευμένου **for** loop καθώς κάθε μεταβλητή αντιπροσωπεύει ξεχωριστό stack για το κάθε νήμα. Επίσης γίνεται χρήση του reduction(+:tempD), με την οποία αποθηκεύει σε κάθε νήμα την τιμή της μεταβλητής. Η χρησιμότητά της εντολής είναι η δυνατότητα να κάνουμε πράξεις εκτός του εμφωλευμένου **for** loop χωρίς να χάνονται οι τιμές. Τέλος το #pragma omp flush επιβεβαιώνει πως όλες οι εντολές ανάγνωσης και εγγραφής που υπάρχουν πριν το flush, θα ολοκληρωθούν πριν το flush και καμία εντολή ανάγνωσης και εγγραφής που υπάρχει μετά το flush, δεν θα ξεκινήσει πριν τελειώσει το flush.

```
//Ενημέρωση της λίστας S , η λίστα S περιέχει όλους τους κόμβους με inDegree=0

void updateS2(int **arr,list * head,list * L)
{
   int j,bool1,bool2;
//Οι κόμβοι της S δεν περιέχονται ούτε στον πίνακα γειτνίασης ούτε στην L
  #pragma omp parallel for private(bool1,bool2,j) shared(arr,head,L,adjMatrix,V)
     for(j=0;j<V;j++)
     {
      //bool1,bool2 => μεταβλητές ελέγχου για τις λίστες L,S αντίστοιχα
     //ενημέρωση της λίστας S θα γίνει μόνο αν ο κόμβος δεν περιέχεται ούτε στην S ,ούτε στην L
       bool1=existInList(L,j);
       bool2=existInList(head,j);

       if(bool1==0&&bool2==0)
        {
          if(arr[V][j]==0 )
          {
           #pragma omp task
            printf("---Eisagw komvo %d stin S---\n\n",j);
           #pragma omp task
            push(head,j);
           #pragma omp taskwait
          }
       }
    }
}
```

Επεξήγηση παραλληλοποίησης

Χρησιμοποιούμε το #pragma omp parallel για την έναρξη παράλληλης περιοχής και την δημιουργία νημάτων. Στη συνέχεια έχουμε την εντολή for για να παραλληλοποιηθεί το επακόλουθο **for** loop και θέτουμε ως παραμέτρους την εντολή private με ορίσματα τις μεταβλητές (bool1,bool2,j) και την εντολή shared(arr,head,L,adjMatrix,V), ώστε όλα μας τα νήματα να μπορούν να κάνουν ταυτόχρονη επεξεργασία στις τέσσερις μεταβλητές. Στη συνέχεια με την εντολή #pragma omp task δημιουργούνται δύο έργα, το καθένα από τα οποία, ανατίθεται σε ένα νήμα και «δένεται» με αυτό. Το νήμα αυτό στη συνέχεια είναι υπεύθυνο για τη τύπωση printf ("---Eisagw komvo %d stin S---\n\n",j) και την εισαγωγή του κόμβου j στη λίστα S. Στο τέλος, έχουμε την εντολή #pragma omp taskwait, ώστε να διασφαλιστεί ότι σε κάθε επανάληψη του **for** loop θα έχουν ολοκληρωθεί και τα αντίστοιχα task.

```
//Ενημέρωση του πίνακα γειτνίασης arr[][]
void updateMatrix(int ** arr,int x)
{
//Αφαιρείται ο κόμβος X
   int j;
   #pragma omp parallel for shared(arr,x,V) schedule(dynamic)
       for(j=0;j<V;j++)
         arr[x][j]=0;
 // #pragma omp flush
}
```

Επεξήγηση παραλληλοποίησης

Χρησιμοποιούμε το #pragma omp parallel για την έναρξη παράλληλης περιοχής και την δημιουργία νημάτων και θέτουμε ως παράμετρο την εντολή shared(arr,x,V), ώστε όλα μας τα νήματα να μπορούν να κάνουν ταυτόχρονη επεξεργασία στις δύο μεταβλητές και την schedule (dynamic), ώστε να γίνεται δυναμική ανάθεση στα νήματα μέσα στο **for** loop.

```
//Ελέγχει το γράφο για αν είναι κυκλικός
int untitled()
{
   int i,check=0;
  #pragma omp parallel for shared (check,adjMatrix,V) schedule(dynamic)
      for(i=0;i<V;i++)
      {
         if(adjMatrix[V][i]!=0)
         {
             check=1;
         }
      }
   return check;
}
```

Επεξήγηση παραλληλοποίησης

Χρησιμοποιούμε το #pragma omp parallel για την έναρξη παράλληλης περιοχής και την δημιουργία νημάτων και θέτουμε ως παράμετρο την εντολή shared(check,adjMatrix,V), ώστε όλα μας τα νήματα να μπορούν να κάνουν ταυτόχρονη επεξεργασία στις τρεις μεταβλητές και την schedule(dynamic), ώστε να γίνεται δυναμική ανάθεση στα νήματα μέσα στο **for** loop.

```
void printList(list * head)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            list * current = head->next;

            while (current != NULL)
            {
                #pragma omp task
                printf("%d\t", current->name);
                current = current->next;
            }
        }
    }

}
```

Επεξήγηση παραλληλοποίησης

Χρησιμοποιούμε το #pragma omp parallel για την έναρξη παράλληλης περιοχής και την δημιουργία νημάτων. Στη συνέχεια χρησιμοποιούμε την εντολή #pragma omp single, ώστε μόνο ένα νήμα να έχει πρόσβαση στην αλλαγή της θέσης του pointer στην λίστα και ύστερα με την εντολή #pragma omp task δημιουργείται ένα έργο, που ανατίθεται σε ένα νήμα και «δένεται» με αυτό. Το νήμα αυτό στη συνέχεια είναι υπεύθυνο για τη τύπωση printf ("%d\t", current->name) και την μετάβαση του pointer στο επόμενο στοιχείο.

Παράδειγμα εκτέλεσης του κώδικα

Σας παρουσιάζουμε την εκτέλεση του κώδικα μας για ένα μικρό γράφημα με 5 κόμβους, ώστε να γίνει αντιληπτή η ορθότητα του αλγορίθμου μας. Σημειώνουμε ότι οι χρόνοι που παρουσιάζονται στην συνέχεια είναι για γράφημα με 323 επίπεδα και περίπου 1500 κόμβους. Επίσης να τονιστεί ότι στην εκτέλεση του μεγάλου γραφήματος έχουμε αφαιρέσει από τον κώδικα την συνεχή εκτύπωση της S και της L, και τυπώνεται μόνο η τελική τυπολογική διάταξη του γραφήματος , για εξοικονόμηση χρόνου. Τα βήματα αυτά βρίσκονται με μορφή σχόλιων, σε περίπτωση που θέλετε να τα ελέγξετε.

```
O dothen pinakas geitniasis einai:

0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
0 0 0 0 0
0 1 1 1 1

Komvoi:         0       1       2       3       4
InDegree:       0       1       1       1       1


------------------------------------------------------------------

---Eisagw komvo 0 stin S---

0       =S

0       =L

o komvos poy diagrafw einai 0

Komvoi:         0       1       2       3       4
InDegree:       0       0       1       1       1


---Eisagw komvo 1 stin S---

------------------------------------------------------------------

1       =S

0       1       =L

o komvos poy diagrafw einai 1

Komvoi:         0       1       2       3       4
InDegree:       0       0       0       1       1


---Eisagw komvo 2 stin S---

------------------------------------------------------------------

2       =S

0       1       2       =L

o komvos poy diagrafw einai 2

Komvoi:         0       1       2       3       4
InDegree:       0       0       0       0       1
```

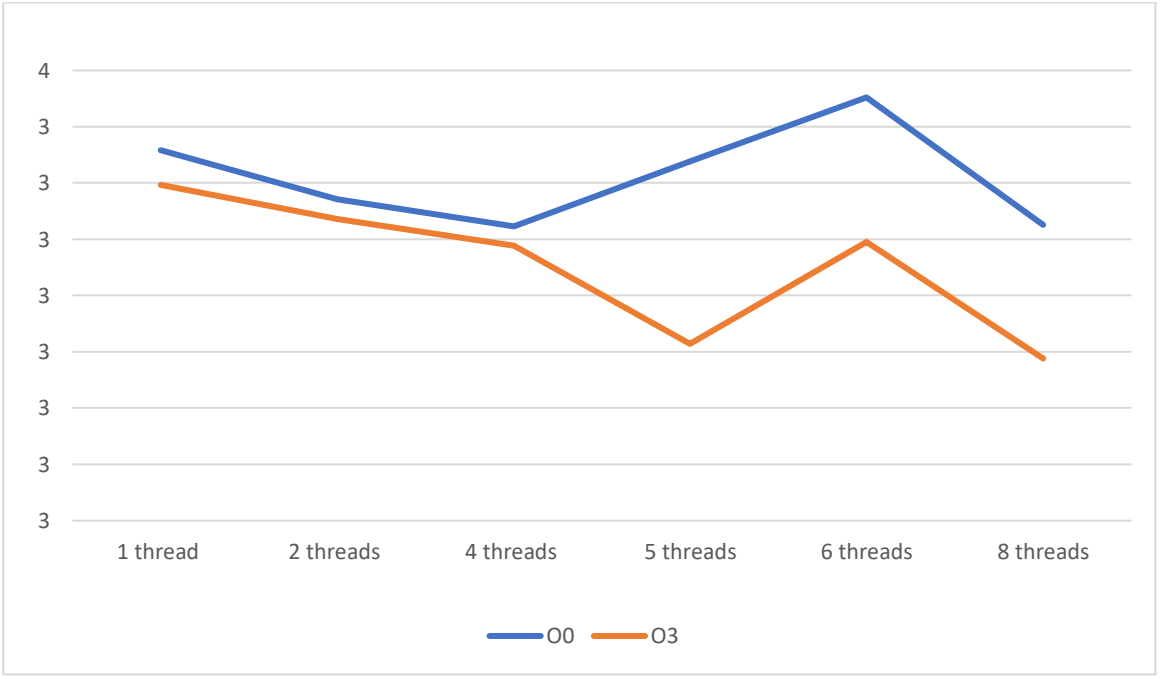
```
---Eisagw komvo 3 stin S---

------------------------------------------------------------------------

3       =S

0       1       2       3       =L

o komvos poy diagrafw einai 3

Komvoi:         0       1       2       3       4
InDegree:       0       0       0       0       0


---Eisagw komvo 4 stin S---

------------------------------------------------------------------------

4       =S

0       1       2       3       4       =L

o komvos poy diagrafw einai 4

Komvoi:         0       1       2       3       4
InDegree:       0       0       0       0       0


------------------------------------------------------------------------

H topologiki diataxi einai :    0       1       2       3       4



Euxaristw :)
```

Αποτελέσματα Παραλληλοποίησης

Οι χρόνοι που σας παρουσιάζουμε είναι οι MO, από τις τιμές που έχουμε καταγράψει στο time323.txt (1455 κόμβους), από 5 διαφορετικά run του κώδικα μας, για κάθε περίπτωση.

		Χρόνοι(sec)					
Σειριακός :		10.9982802					
Παράλληλος:	**Νήματα**	**1**	**2**	**4**	**5**	**6**	**8**
	-O0	3.4290692	3.3855872	3.3614450	3.4192760	3.4760278	3.3625840
	-O3	3.3982676	3.3677310	3.3640946	3.2572110	3.3477072	3.2440204

Γράφημα



Παρατηρείται μια ελάχιστη αύξηση στον runtime όταν χρησιμοποιούμε 5 και 6 νήματα , κάτι το οποίο όμως, βελτιστοποιείται με την χρήση 8 νημάτων.

Ποσοστό παραλληλοποίησης :

$$\frac{\mathrm{T}parallel}{T\mathrm{serial}}100\% = \frac{3{,}2440204}{10{,}9982802}\ 100\% = 29{,}3134\%$$

Η βέλτιστη παραλληλοποίηση που επιτυγχάνουμε είναι 29,3 % σε σχέση με τον σειριακό μας χρόνο.

ΠΑΡΑΡΤΗΜΑ

Υποσημείωση 1:

Οι τιμές υπολογίστηκαν σε laptop ,το οποίο διέθετε 4πύρηνο επεξεργαστή (intel i5 10th generation) και 8 νήματα.

Υποσημείωση 2:

Κατά την διάρκεια εκτέλεσης του προγράμματος , παρατηρήθηκε πως σε μερικά runs είχαμε segmetation fault ,ενώ σε άλλα όχι. Στο segmentation fault κύριο λόγο αποτελεί, η δέσμευση της μνήμης. Πιθανότατα οφείλεται στην δυναμική δέσμευση μνήμης και την αρχικοποίηση της. Επιπλέον τα νήματα μας, χρησιμοποιούν πίνακα με πολύ μεγάλο μέγεθος, ενώ αρχικοποιούνται με default μέγεθος stack (πιθανή επίλυση να θέσουμε μεγαλύτερο μέγεθος stack για το κάθε νήμα). Και τέλος, κατά το στάδιο της καταμέτρησης του χρόνου παρατηρήσαμε πως όσο αυξάνουμε τα threads και χρησιμοποιούμε O3 έναντι του O0 κατά το compile το segmentation fault εξαφανίζεται.

Υποσημείωση 3:

Μετά από αναζήτηση στο διαδίκτυο βρήκαμε ότι το φαινόμενο με την μείωση των segmetation fault ,που οφειλόταν στην χρήση του -O3 ,έναντι του -O0 , έχει παρατηρηθεί ξανά στο παρελθόν και από άλλον χρήστη του Openmp και η Intel το έχει λάβει υπόψιν της ως ζήτημα προς επίλυση.