Ιωάννης Χουλιάρας
Email: cs02631@uoi.gr
Course: Παράλληλα Συστήματα και Προγραμματισμός
Professor: Βασίλειος Δημακόπουλος

#1 εργαστηριακή άσκηση
Αριθμός Μητρώου: 2631
Εαρινό εξάμηνο 2020
27 Απριλίου 2020

Με την πρώτη εργαστηριακή άσκηση στο μάθημα "Παράλληλα Συστήματα και Προγραμματισμός" θα κάνουμε μία εισαγωγή στον παράλληλο προγραμματισμό με το μοντέλο κοινόχρηστου χώρου διευθύνσεων μέσω του προτύπου OpenMP. Ζητείται η παραλληλοποίηση στο σειριακό πρόγραμμα του πολλαπλασιασμού πινάκων με δύο διαφορετικούς τρόπους, καθώς και η εύρεση πρώτων αριθμών.

Οι μετρήσεις πραγματοποιήθηκαν στο παρακάτω σύστημα:

ΟΝΟΜΑ ΥΠΟΛΟΓΙΣΤΗ	opti3060ws07
ΕΠΕΞΕΡΓΑΣΤΗΣ	Intel i3-8300 @ 3.70GHz
ΠΛΗΘΟΣ ΠΥΡΗΝΩΝ	4
ΜΕΤΑΦΡΑΣΤΗΣ	gcc v7.5.0

Πρόβλημα 1
Στην άσκηση αυτή ζητείται η παραλληλοποίηση με έξι διαφορετικούς τρόπους. Παραλληλοποιώντας έναν από τους τρεις βρόχους κάθε φορά και κάθε βρόχος με διαφορετικό δρομολογητή (schedule), static και dynamic.

Μέθοδος Παραλληλοποίησης

Χρησιμοποιήθηκε το σειριακό πρόγραμμα από την προσωπική ιστοσελίδα του καθηγητή του μαθήματος. Αρχικά στο αρχείο firstFor.c παραλληλοποιείται το πρώτο for με την εντολή **pragma omp parallel for** και περνάμε τις μεταβλητές j, k , sum ως private καθώς δεν έχουν αρχικοποιηθεί στην αρχή της main. Με αυτό τον τρόπο για κάθε νήμα, οι μεταβλητές στην αρχή έχουν άκυρες τιμές (σκουπίδια). Εφόσον κάθε νήμα θα δίνει διαφορετικές τιμές στις μεταβλητές, τα υπόλοιπα δεν θα έχουν πρόσβαση στην τιμή τους.

```
#include <omp.h>


// set number of threads to 4 :
omp_set_num_threads(4);
start = omp_get_wtime();
#pragma omp parallel for private(j, k, sum) schedule(static)
for (i = 0; i < N; i++)
{
  for (j = 0; j < N; j++)
  {
    for (k = sum = 0; k < N; k++)
      sum += A[i][k] * B[k][j];
    C[i][j] = sum;
  };
}
end = omp_get_wtime();
```

Στο δεύτερο αρχείο secondFor.c έχουμε για ακόμη μία φορά τις μεταβλητές j, k, sum ως private αλλά αυτή την φορά το **pragma omp parallel for** είναι στο δεύτερο for του αλγορίθμου. Σε καμία από τις δύο υλοποιήσεις δεν χρειάστηκε αμοιβαίος αποκλεισμός.

```
#include <omp.h>

omp_set_num_threads(4);
start = omp_get_wtime();
for (i = 0; i < N; i++)
{
#pragma omp parallel for private(j, k, sum) schedule(static)
  for (j = 0; j < N; j++)
  {
    for (k = sum = 0; k < N; k++)
      sum += A[i][k] * B[k][j];
    C[i][j] = sum;
  };
}
end = omp_get_wtime();
```

Στο τρίτο αρχείο thirdFor.c το **pragma omp parallel for** έχει τοποθετηθεί στο τρίτο for του αλγορίθμου. Η μεταβλητή k έχει οριστεί ως private και οι υπόλοιπες από default σε shared γιατί κάθε νήμα χρειάζεται την πληροφορία των θέσεων στους πίνακες. Επίσης χρησιμοποιείται το reduction όπου προσθέτει τους πολλαπλασιασμούς των θέσεων και μετά τοποθετεί την τιμή καθώς χρειάζεται αμοιβαίος αποκλεισμός.

```
#include <omp.h>

omp_set_num_threads(4);
start = omp_get_wtime();
for (i = 0; i < N; i++)
for (j = 0; j < N; j++)
{
#pragma omp parallel for private(k) schedule(static) reduction(+:C[i][j])
  for (k = 0; k < N; k++)
    C[i][j] += A[i][k] * B[k][j];
    };
end = omp_get_wtime();
```

Πειραματικά αποτελέσματα - μετρήσεις

Οι μετρήσεις πραγματοποιήθηκαν στο σύστημα που αναφέρεται στην εισαγωγή και χρονομετρήθηκαν με την χρήση της omp get wtime(). Χρησιμοποιήθηκαν από 1 έως 4 νήματα και schedule (static), schedule(dynamic) για κάθε μία μέτρηση όπου πραγματοποιήθηκαν για τον πολλαπλασιασμό των πινάκων με μέγεθος 1024.

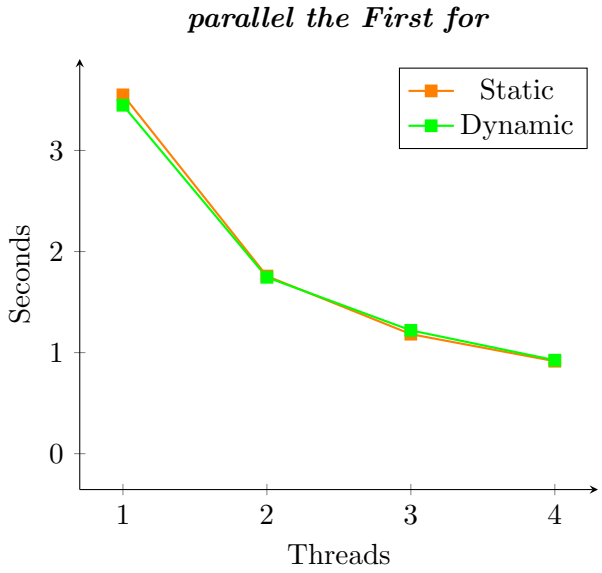
Κάθε πείραμα εκτελέστηκε 4 φορές και υπολογίστηκαν οι μέσοι όροι. Προσοχή δόθηκε οι χρόνοι να μην συμπεριλαμβάνουν την ανάγνωση των αρχείων. Τα αποτελέσματα δίνονται στους παρακάτω πίνακες (οι χρόνοι είναι σε sec):

Παραλληλοποίηση πρώτου for με schedule(static)

ΝΗΜΑΤΑ	1Η ΕΚΤΕΛΕΣΗ	2Η ΕΚΤΕΛΕΣΗ	3Η ΕΚΤΕΛΕΣΗ	4Η ΕΚΤΕΛΕΣΗ	ΜΕΣΟΣ ΟΡΟΣ
1	3.45	3.45	3.81	3.49	3.55
2	1.76	1.76	1.75	1.76	1.7575
3	1.18	1.19	1.18	1.18	1.1825
4	0.91	0.88	0.98	0.89	0.915

Παραλληλοποίηση πρώτου for με schedule(dynamic)

ΝΗΜΑΤΑ	1Η ΕΚΤΕΛΕΣΗ	2Η ΕΚΤΕΛΕΣΗ	3Η ΕΚΤΕΛΕΣΗ	4Η ΕΚΤΕΛΕΣΗ	ΜΕΣΟΣ ΟΡΟΣ
1	3.45	3.45	3.45	3.44	3.4475
2	1.76	1.73	1.73	1.76	1.745
3	1.18	1.19	1.17	1.34	1.22
4	0.89	0.88	0.89	1.04	0.925



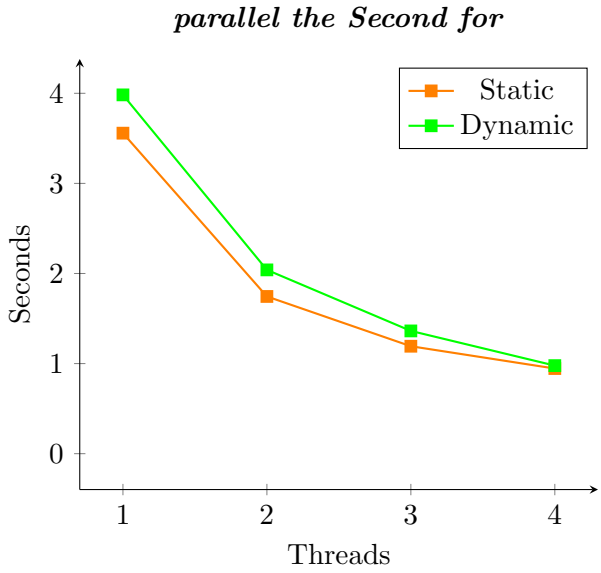
Παρατηρούμε ότι οι χρόνοι είναι σχεδόν ίδιοι. Το static με 3 νήματα έκανε ελαφρώς πιο γρήγορο χρόνο αλλά όχι ιδιαίτερα σημαντικό.

Παραλληλοποίηση δεύτερου for με schedule(static)

ΝΗΜΑΤΑ	1Η ΕΚΤΕΛΕΣΗ	2Η ΕΚΤΕΛΕΣΗ	3Η ΕΚΤΕΛΕΣΗ	4Η ΕΚΤΕΛΕΣΗ	ΜΕΣΟΣ ΟΡΟΣ
1	3.83	3.49	3.46	3.45	3.5575
2	1.75	1.74	1.74	1.75	1.745
3	1.22	1.18	1.18	1.19	1.1925
4	1.01	0.97	0.89	0.91	0.945

Παραλληλοποίηση δεύτερου for με schedule(dynamic)

ΝΗΜΑΤΑ	1Η ΕΚΤΕΛΕΣΗ	2Η ΕΚΤΕΛΕΣΗ	3Η ΕΚΤΕΛΕΣΗ	4Η ΕΚΤΕΛΕΣΗ	ΜΕΣΟΣ ΟΡΟΣ
1	3.94	4.06	3.90	4.03	3.9825
2	2.20	2.33	1.83	1.80	2.04
3	1.39	1.47	1.40	1.19	1.3625
4	0.90	1.06	0.90	1.05	0.9775



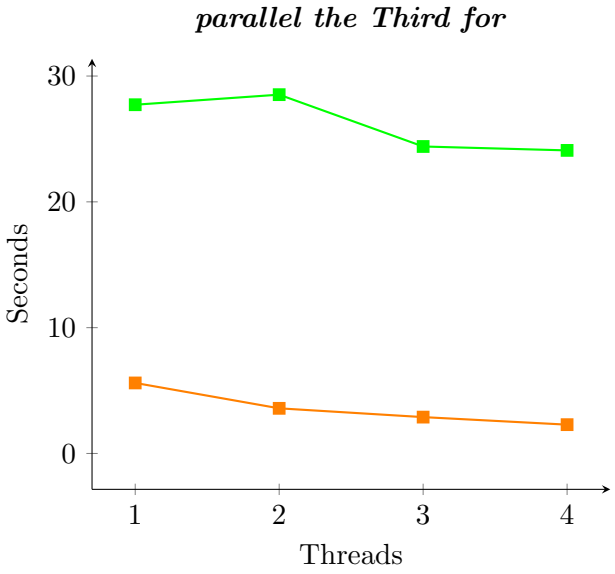
Στο δεύτερο for-loop παρατηρούμε ότι με static παίρνουμε καλύτερα αποτελέσματα από ότι με dynamic όμως με 4 νήματα οι χρόνοι είναι σχεδόν ίσοι.

Παραλληλοποίηση τρίτου for με schedule(static)

ΝΗΜΑΤΑ	1Η ΕΚΤΕΛΕΣΗ	2Η ΕΚΤΕΛΕΣΗ	3Η ΕΚΤΕΛΕΣΗ	4Η ΕΚΤΕΛΕΣΗ	ΜΕΣΟΣ ΟΡΟΣ
1	5.78	5.82	5.41	5.41	5.605
2	3.58	3.58	3.59	3.60	3.5875
3	3.03	2.83	2.70	3	2.89
4	2.29	2.29	2.30	2.28	2.29

Παραλληλοποίηση τρίτου for με schedule(dynamic)

ΝΗΜΑΤΑ	1Η ΕΚΤΕΛΕΣΗ	2Η ΕΚΤΕΛΕΣΗ	3Η ΕΚΤΕΛΕΣΗ	4Η ΕΚΤΕΛΕΣΗ	ΜΕΣΟΣ ΟΡΟΣ
1	27.95	27.63	27.70	27.60	27.72
2	28.74	28.69	28.96	26.96	28.52
3	25.84	24.55	23.47	23.75	24.40
4	24.54	23.64	23.75	24.46	24.09



Στο τρίτο for-loop έχουμε την μεγαλύτερη διαφορά από τις άλλες 2 περιπτώσεις. Στο static οι χρόνοι είναι σχετικά μικροί ενώ στο dynamic εμφανίζεται μία μεγάλη αύξηση που οφείλεται στον τρόπο της δρομολόγησης και του αμοιβαίου αποκλεισμού που απαιτείται για την σωστή υλοποίηση του αλγορίθμου.

Σχόλια

Παρατηρώντας τα αποτελέσματα βλέπουμε ότι αυξάνοντας τον αριθμό των νημάτων, ο χρόνος εκτέλεσης μειώνεται σημαντικά από το σειριακό πρόγραμμα που χρονομετρήθηκε με μέσο όρο 3.16 seconds. Οι καλύτεροι χρόνοι σημειώθηκαν στην παραλληλοποίηση του πρώτου βρόχου μιας και παραλληλοποιείται όλος ο αλγόριθμος σε αντίθεση με τους άλλους δύο. Ο τρίτος τρόπος σημειώνει μία πολλή σημαντική αύξηση που οφείλεται στην πρόσθεση μιας κρίσιμης περιοχής τόσο στην πολιτική static όσο και στην dynamic που παρατηρείται μία αρκετά μεγάλη αύξηση χρόνου.

Πρόβλημα 2

Στην άσκηση ζητείται η παραλληλοποίηση του προγράμματος για την εύρεση πρώτων αριθμών. Θα πρέπει να δημιουργηθεί στο σειριακό πρόγραμμα του καθηγητή μία ακόμη μέθοδος όπου χωρίς να αλλαχθεί ο αλγόριθμος, να προστεθεί κώδικας για την ορθή παραλληλοποίηση.

Μέθοδος Παραλληλοποίησης

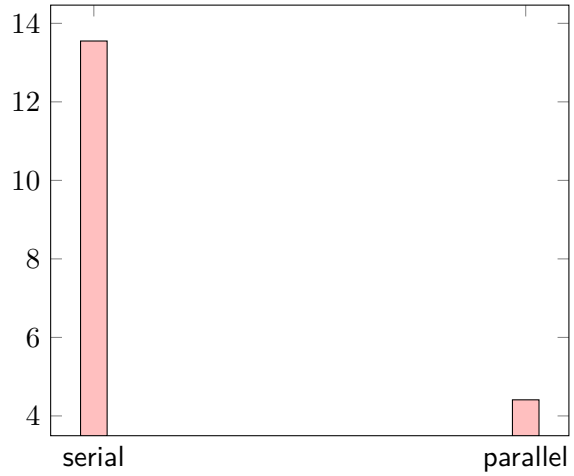
Στο αρχείο primeNumbers.c που δημιουργήθηκε από το σειριακό πρόγραμμα που μας δόθηκε, προστέθηκε μία ακόμη μέθοδος στην οποία αντιγράφουμε την υλοποίηση της σειριακής έκδοσης και προσθέτουμε **pragma omp parallel for** με τις μεταβλητές (i, quotient, remainder, num, divisor) ως privates καθώς δεν έχουν αρχικοποιηθεί ακόμα και περιέχουν σκουπίδια. Η μεταβλητή count θα περαστεί ως shared καθώς όλα τα νήματα θα πρέπει να έχουν πρόσβαση στην τιμή της. Απαιτείται αμοιβαίος αποκλεισμός φυσικά για την αύξηση της τιμής κάθε φορά. Τέλος η μεταβλητή lastPrime θα περαστεί ως lastprivate ώστε το τελευταίο νήμα να ενημερώσει την τελική τιμή της. Επίσης πριν την τοποθέτηση της παράλληλης περιοχής, θέτουμε το omp set dynamic(0) και τον αριθμό νημάτων που θέλουμε που στην περίπτωση μας είναι 4.

```
#include <omp.h>

// set 4 threads here but first set dynamic(0)
omp_set_dynamic(0);
omp_set_num_threads(4);

start = omp_get_wtime();

// parallel here using pragma omp parallel for..
// set the values : (i, quotient, remainder, num and divisor) as (privates)
// set the value : (lastPrime) as (lastprivate)
// set the values : (count) as (shared) #default
#pragma omp parallel for private(quotient, remainder, num, divisor) ↩
    lastprivate(lastprime) default(shared)
for (i = 0; i < (n-1)/2; ++i) {    /* For every odd number */
  num = 2*i + 3;

  divisor = 1;
  do
  {
    divisor += 2;                  /* Divide by the next odd */
    quotient  = num / divisor;
    remainder = num % divisor;
  } while (remainder && divisor <= quotient);  /* Don't go past sqrt */

  if (remainder || divisor == num) /* num is prime */
  {
#pragma omp critical
    count++;
    lastprime = num;
  }
}
end = omp_get_wtime();
```

Πειραματικά αποτελέσματα - μετρήσεις

Η χρονομέτρηση έγινε για ακόμη μία φορά με την συνάρτηση omp get wtime() στο σύστημα μας με χρόνους:

Υλοποιηση	1η εκτελεση	2η εκτελεση	3η εκτελεση	4η εκτελεση	Μεσος οροσ
Σειριακή	13.55	13.55	13.56	13.55	13.55
Παράλληλη	4.41	4.41	4.41	4.41	4.41



Σχόλια

Ο αλγόριθμος ξεκινάει με ένα for-loop οπότε για ακόμη μία φορά επιλέγεται το pragma omp parallel for για τον παραλληλισμό. Για την σωστή υλοποίηση, η μεταβλητή counter πρέπει να είναι shared καθώς κάθε νήμα πρέπει να έχει πρόσβαση στην τιμή της. Οι υπόλοιπες μεταβλητές δεν χρειάζεται να είναι shared καθώς χρησιμοποιούνται από τα νήματα τοπικά. Η μεταβλητή lastPrime θα μπορούσε να είναι shared καθώς πρέπει στο τέλος να ενημερωθεί με την σωστή τιμή και γι αυτό πρέπει τα νήματα να έχουν πρόσβαση. Ο μέσος χρόνος με την lastPrime shared είναι 4.53 seconds ενώ ως lastprivate είναι 4.41 seconds. Είναι μια μικρή διαφορά αλλά είναι καλύτερη υλοποίηση καθώς η lastprivate αναθέτει στο τελευταίο νήμα να κάνει την ενημέρωση της μεταβλητής.

Πρόβλημα 3

Στην τρίτη άσκηση μας ζητείται η υλοποίηση του σειριακού προγράμματος για τον πολλαπλασιασμό πινάκων με παραλληλισμό με την βοήθεια των tasks. Από το τερματικό ο χρήστης θα πρέπει να δώσει έναν αριθμό κατάλληλο ώστε να γίνει σωστά η διαμοίραση των εργασιών.

Μέθοδος Παραλληλοποίησης

Στο αρχείο mulmatTasks.c αντιγράφουμε τον κώδικα από τον σειριακό και προσθέτουμε την παράλληλη περιοχή. Ο τρόπος σκέψης είναι ο εξής. Θα ορίσουμε την παράλληλη περιοχή πριν από το πρώτο for-loop με πέρασμα της τιμής S ως firstprivate. Στην παράλληλη περιοχή θα έχουμε ένα pragma omp single ώστε ένα νήμα μόνο να καλέσει μία μέθοδο όπου θα υλοποιείται ο αλγόριθμος. Πριν όμως αναλυθεί η μέθοδος, πρέπει να αλλαχθεί ο κώδικας ώστε να δέχεται τον σωστό αριθμό από τον χρήστη. Παρακάτω βλέπουμε ότι ο κώδικας δέχεται έναν αριθμό, ελέγχει εάν είναι έγκυρος και αν είναι τυπώνει πόσες εργασίες/tasks θα δημιουργηθούν.

```
/* check argc */
if (argc != 2)
exit( 1 + printf("Give me one number for the S\n") );

/* Take S from argv and atoi */
int S = atoi(argv[1]);

/* Check if S is correct */
if (N%S != 0)
exit( 1 + printf("Wrong S number\n") );

tasks = (N/S)*(N/S);
printf("I will make %d Tasks\n", tasks);
```

Αμέσως μετά δημιουργείται η παράλληλη περιοχή και καλείται η μέθοδος. Ο χρόνος πρέπει να αρχίσει να μετράει μόλις δημιουργηθεί η παράλληλη περιοχή και να σταματήσει να χρονομετρεί μετά το τέλος του παραλληλισμού.

```
omp_set_num_threads(4);
start = omp_get_wtime();
#pragma omp parallel firstprivate(S)
{
#pragma omp single
  parallel_work(S); // call my function!
}
end = omp_get_wtime();
```

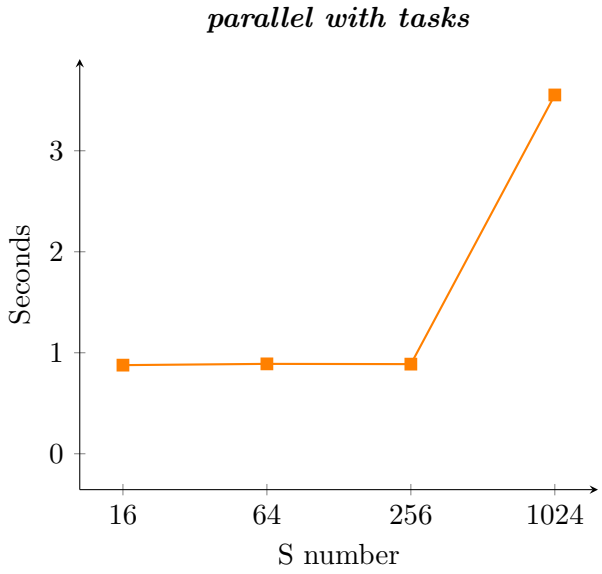
Τέλος στην μέθοδο parallel work έχουμε κάποιες νέες μεταβλητές. Πριν δηλώσουμε τις εργασίες, έχουμε 2 for-loops όπου έχουν βήμα +S και όχι με βήμα 1 που ήταν στην αρχική υλοποίηση. Στο δεύτερο for-loop δημιουργούμε τις εργασίες. Στις εργασίες οι μεταβλητές (i, j) θα δηλωθούν ως shared διότι κάθε εργασία, πρέπει να έχει πρόσβαση στα i, j. Επίσης δηλώνουμε ότι είναι untied ώστε ένα νήμα αν αφήσει μία δουλειά, ένα άλλο να αναλάβει να την τελειώσει. Τα 3 for-loops της σειριακής υλοποίησης τροποποιούνται ως εξής. Δύο νέες μεταβλητές αρχικοποιούνται στις κοινόχρηστες τιμές των i, j, το βήμα τους είναι +1 μέχρι να φτάσει στο i+S και j+S και το τρίτο for-loop να γίνεται η υλοποίηση του αλγορίθμου με τις νέες θέσεις της κάθε εργασίας.

```
void parallel_work(int S)
{
  int i, j, k, sum, p, t;

  for (i = 0; i < N; i = i+S)
    for (j = 0; j < N; j = j+S)
    {
#pragma omp task shared(i, j) untied
      {
        for (p = i; p < i+S; p++)
          for(t = j; t < j+S; t++) {
            for (k = sum = 0; k < N; k++)
              sum += A[p][k]*B[k][t];
            C[p][t] = sum;
          }
      }
    }
}
```

Πειραματικά αποτελέσματα - μετρήσεις

Ο παρακάτω πίνακας παρουσιάζει τις μετρήσεις από το σύστημα μας.

Αριθμος S	1η εκτελεση	2η εκτελεση	3η εκτελεση	4η εκτελεση	Μεσος οροσ
16	0.88	0.87	0.87	0.89	0.8775
64	0.89	0.89	0.89	0.89	0.89
256	0.88	0.89	0.88	0.90	0.8875
1024	3.44	3.87	3.44	3.46	3.5525



Σχόλια

Στην γραφική παράσταση και από τον πίνακα παρατηρούμε ότι όσο έχω έναν αριθμό S που είναι διαφορετικός από 1024, τότε ο χρόνος μειώνεται αρκετά και συγκεκριμένα πιο γρήγορο από οποιαδήποτε άλλη υλοποίηση προσπαθήσαμε στο πρώτο πρόβλημα. Αποδεικνύεται ότι η εισαγωγή των tasks ήταν ένα αρκετά μεγάλο βήμα στα παράλληλα συστήματα και γενικότερα στον προγραμματισμό. Διότι δημιουργούνται

εργασίες που μπορούν να υλοποιηθούν κατευθείαν ή να μείνουν σε μία ουρά ώστε να εκτελεστούν πιο μετά ή ένα νήμα να αφήσει μία δουλειά και ένα άλλο νήμα να συνεχίσει την δουλειά του όπως είδαμε και παραπάνω και αρκετές άλλες δυνατότητες των εργασιών.