Ειδικά Θέματα Παράλληλου Προγραμματισμού

Εργασία #1 – OpenMP (2022-23)

Γενικές Πληροφορίες

Το πρώτο σετ ασκήσεων ήταν σε προγραμματισμό με OpenMP, και συγκεκριμένα σε C. Όσον αφορά το μηχάνημα που χρησιμοποιήθηκε για τις μετρήσεις, ήταν με επεξεργαστή Intel i5-10600KF (6 πυρήνων), σε gcc 8.1.0 (μέσω του Code::Blocks IDE σε Windows).

Άσκηση 1

Το Πρόβλημα

Στην πρώτη άσκηση μας δόθηκε η σειριακή υλοποίηση ενός προγράμματος, το οποίο συναρτήσει του ακεραίου N που θα του περαστεί ως παράμετρος, επιστρέφει το πλήθος πρώτων αριθμών στο διάστημα [0, N], καθώς και τον μεγαλύτερο πρώτο αριθμό στο εν λόγω διάστημα. Στη συνέχεια, έπρεπε χωρίς να τροποποιήσουμε καθόλου τον σειριακό κώδικα και προσθέτοντας μόνο οδηγίες προεπεξεργαστή σχετικά με OpenMP (δηλαδή #pragma omp …), να βελτιστοποιήσουμε τον αλγόριθμο από θέμα χρόνου, και να πειραματιστούμε χρονομετρώντας την απόδοση του σειριακού αλγορίθμου και του αλγορίθμου που έκανε χρήση OpenMP με 1, 2, 3 και 4 νήματα αντίστοιχα.

Μέθοδος Παραλληλοποίησης

Η αρχική σειριακή μέθοδος που μας δόθηκε ακολουθεί στην επόμενη σελίδα:

```
void serial_primes(long int n) {
	long int i, num, divisor, quotient, remainder;

	if (n < 2) return;
	count = 1;				/* 2 is the first prime */
	lastprime = 2;

	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 */
		{
			count++;
			lastprime = num;
		}
	}
}
```

Για την παραλληλοποίηση του άνω κώδικα, έκανα τις ακόλουθες τροποποιήσεις (τονισμένες με πορτοκαλί χρώμα):

```
void openmp_primes(long int n) {
	long int i, num, divisor, quotient, remainder;

	if (n < 2) return;
	count = 1;				/* 2 is the first prime */
	lastprime = 2;

    omp_set_dynamic(0);

    #pragma omp parallel for num_threads(x) schedule(guided) private(i, num,
    divisor, quotient, remainder) reduction(+: count) reduction(max: lastprime)

    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 */
        {
            count++;
            lastprime = num;
        }
    }
}
```

Ακολουθούν αναλυτικές αιτιολογήσεις για τις προσθήκες που έκανα στον κώδικα:

- **omp_set_dynamic(0):**

  Ο λόγος που πρόσθεσα την άνω εντολή στον κώδικα ήταν προκειμένου να εξαναγκάσω την χρήση του ακριβή αριθμού νημάτων που θέτω στην παράλληλη περιοχή.

- **#pragma omp parallel for num_threads(x) schedule(guided) private(i, num, divisor, quotient, remainder) reduction(+: count) reduction(max: lastprime):**

  Καταρχάς, επέλεξα parallel for καθώς παρατήρησα ότι το for loop επιδέχεται διαχωρισμό σε όσα x νήματα επιθυμούμε χωρίς καμία επιπλοκή, υπό κάποιες σημαντικές προϋποθέσεις που θα αναλύσω στη συνέχεια.

  Έπειτα, επέλεξα schedule(guided), καθώς αποδείχθηκε το πιο γρήγορο schedule έπειτα από πειράματα με τα υπόλοιπα schedules (static, dynamic και runtime). Το runtime ήταν ελάχιστα χειρότερο (υποθέτω λόγω κάποιου μικρού overhead που παρουσιάζεται στην πρόσβαση της μεταβλητής OMP_SCHEDULE), ακολουθούμενο από το dynamic με πολύ μικρή διαφορά (αν και σταθερή), και με τελευταίο το static, φανερά χειρότερο από τα 3 άλλα schedules. Οι δοκιμές έγιναν και με διάφορες custom chunk τιμές, όμως δεν παρατήρησα κάποια βελτίωση στον χρόνο, και αν μη τι άλλο παρατήρησα μια μικρή πτώση του χρόνου.

  Ύστερα, επέλεξα private για τις μεταβλητές i, num, divisor, quotient και remainder, καθώς γίνεται εύκολα αντιληπτό από την πορεία του προγράμματος ότι κάθε νήμα χρειάζεται τα δικά του αντίγραφα των μεταβλητών για το παράλληλο κομμάτι.

  Στη συνέχεια, απαιτήθηκε αρκετός πειραματισμός προκειμένου να καταλήξω στη χρήση υποβιβάσεων. Αρχικά, είχα θέσει το count και το lastprime ως shared, θέτοντας το εσωτερικό της if ως κρίσιμη περιοχή. Όμως, με αυτόν τον τρόπο, αν και υπολογιζόταν σωστά το count κάθε φορά, παρατήρησα αποκλίσεις στο lastprime από τη φυσική τιμή. Η αιτία μου είναι πλέον προφανής: στο if δεν υπήρχε έλεγχος σχετικά με το lastprime (πχ να υπήρχε και ένα && lastprime < num στο τέλος των συνθηκών του if), οπότε αν ένα νήμα τελείωνε την παράλληλη περιοχή του *μετά* από το νήμα που είχε αναλάβει τις τελευταίες επαναλήψεις του loop, τότε ενημέρωνε την lastprime με λανθασμένες τιμές. Έτσι, κατέληξα στη χρήση υποβίβασης. Αυτό μου έλυσε όλα τα προβλήματα, καθώς πλέον το lastprime συνυπολογίζεται ως το συνολικό μέγιστο μεταξύ όλων των lastprime των νημάτων (το οποίο είναι πλέον αναγκαστικά private, καθώς βρίσκεται σε υποβίβαση).
  Επίσης, η υποβίβαση του count προφανώς δεν είναι αναγκαία και θα μπορούσε το count τεθεί ως κοινόχρηστη μεταβλητή η οποία θα επηρεαζόταν μόνο στο εσωτερικό της if (με επιβεβαιωμένο συγχρονισμό μέσω #pragma omp critical), όμως παρατήρησα μια ξεκάθαρη βελτίωση χρόνου της τάξης του 10-15% έχοντας το count σε υποβίβαση (το οποίο υποθέτω ότι έχει να κάνει με επιταχύνσεις λόγω έλλειψης συναγωνισμού για την κρίσιμη περιοχή), οπότε το διατήρησα έτσι.

Πειραματικά Αποτελέσματα - Μετρήσεις

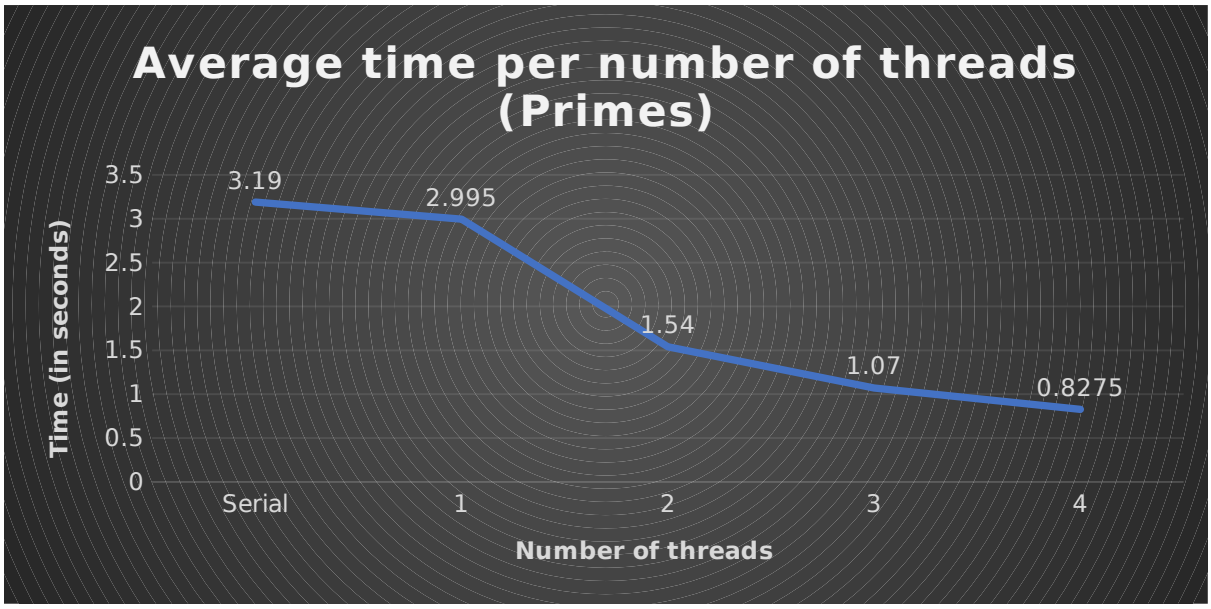
Οι μετρήσεις έγιναν όλες στον υπολογιστή που αναφέρθηκε στην εισαγωγή της αναφοράς, και η χρονομέτρηση έγινε μέσω της συνάρτησης omp_get_wtime(), που προσφέρει το OpenMP. Μέτρησα

τη σειριακή υλοποίηση, και μέτρησα αντίστοιχους χρόνους στην παράλληλη περιοχή, για 1, 2, 3 και 4 νήματα αντίστοιχα. Επίσης, για μεγαλύτερη ακρίβεια αποτελεσμάτων, έγιναν 4 μετρήσεις για κάθε είδος νηματοποίησης και διατήρησα τον μέσο όρο. Ακολουθούν γραφικά τα στατιστικά:

Ο πίνακας με τις αναλυτικές μετρήσεις (ο χρόνος είναι σε δευτερόλεπτα):

Number of Threads	First Run	Second Run	Third Run	Fourth Run	Average
Serial	3.19	3.19	3.19	3.19	3.19
1	2.99	3	3	2.99	2.995
2	1.53	1.51	1.58	1.54	1.54
3	1.06	1.08	1.06	1.08	1.07
4	0.82	0.83	0.84	0.82	0.8275

Και από τον παραπάνω πίνακα, προκύπτει το εξής line chart:



Σχόλια

Καταρχάς, μου έκανε πάρα πολύ μεγάλη εντύπωση ότι υπήρξε διαφορά μεταξύ της σειριακής υλοποίησης και της δοκιμής με 1 νήμα. Θεωρώ ότι αυτό μάλλον έχει να κάνει είτε α) με κάποιο πιθανό παρασκηνιακό optimization με βάση την εντολή omp_set_dynamic(0) (καθώς απροσδόκητα παρατήρησα χειρότερους χρόνους για 1 νήμα χωρίς την εν λόγω εντολή), είτε β) λόγω ανακρίβειας της εντολής omp_get_wtime().

Κατά τα άλλα, τα αποτελέσματα είναι πλήρως αναμενόμενα. Για k νήματα, παρατηρείται βελτίωση πολύ κοντά στο 1/k σε σχέση με τη σειριακή εκτέλεση, το οποίο εμμέσως (σε συνδυασμό με τα ορθά αποτελέσματα καθώς και την απουσία σημείων που θα προσέθεταν overhead, όπως πχ critical section) επιβεβαιώνει και την ορθότητα του αλγορίθμου.

Άσκηση 2

Το Πρόβλημα

Στην δεύτερη άσκηση μας δόθηκε η σειριακή υλοποίηση ενός προγράμματος, το οποίο εφαρμόζει Gaussian blur για τη θόλωση μιας εικόνας. Η συνάρτηση λαμβάνει ως ορίσματα μια εικόνα imgin, μια εικόνα imgout (στην οποία αποθηκεύει την αλλοιωμένη εικόνα), και μια ακτίνα θόλωσης radius, η οποία όσο μεγαλύτερη είναι, τόσο μεγαλύτερο το θόλωμα της εικόνας. Στη συνέχεια, έπρεπε πάλι χωρίς να τροποποιήσουμε καθόλου τον σειριακό κώδικα και προσθέτοντας μόνο οδηγίες προεπεξεργαστή σχετικά με OpenMP, να βελτιστοποιήσουμε τον αλγόριθμο από θέμα χρόνου, χρησιμοποιώντας α) OpenMP loops και β) OpenMP tasks. Συγκεκριμένα, είχαμε την οδηγία κάθε υπολογισμός γραμμής να αποτελεί ένα task. Τέλος, ζητούταν η σύγκριση της απόδοσης του σειριακού αλγορίθμου, του αλγορίθμου για OpenMP loops και του αλγορίθμου για OpenMP tasks με 1, 2, 3 και 4 νήματα αντίστοιχα.

Μέθοδος Παραλληλοποίησης

Η αρχική σειριακή μέθοδος που μας δόθηκε ακολουθεί στην επόμενη σελίδα:

```
void gaussian_blur_serial(int radius, img_t *imgin, img_t *imgout)
{
	int i, j;
	int width = imgin->header.width, height = imgin->header.height;
	double row, col;
	double weightSum = 0.0, redSum = 0.0, greenSum = 0.0, blueSum = 0.0;

	for (i = 0; i < height; i++)
	{
		for (j = 0; j < width ; j++)
		{
			for (row = i-radius; row <= i + radius; row++)
			{
				for (col = j-radius; col <= j + radius; col++)
				{
					int x = clamp(col, 0, width-1);
					int y = clamp(row, 0, height-1);
					int tempPos = y * width + x;
					double square = (col-j)*(col-j)+(row-i)*(row-i);
					double sigma = radius*radius;
					double weight = exp(-square / (2*sigma)) /
(3.14*2*sigma);

					redSum += imgin->red[tempPos] * weight;
					greenSum += imgin->green[tempPos] * weight;
					blueSum += imgin->blue[tempPos] * weight;
					weightSum += weight;
				}
			}
			imgout->red[i*width+j] = round(redSum/weightSum);
			imgout->green[i*width+j] = round(greenSum/weightSum);
			imgout->blue[i*width+j] = round(blueSum/weightSum);

			redSum = 0;
			greenSum = 0;
			blueSum = 0;
			weightSum = 0;
		}
	}
}
```

Οι δύο αλγόριθμοί μου ακολουθούν στις επόμενες σελίδες.

α) Για την παραλληλοποίηση του σειριακού κώδικα με OpenMP loops, έκανα την ακόλουθη τροποποίηση (τονισμένη με πορτοκαλί χρώμα):

```
/* Parallel Gaussian Blur with OpenMP loop parallelization */
void gaussian_blur_omp_loops(int radius, img_t *imgin, img_t *imgout)
{
    omp_set_dynamic(0);

    int i, j;
      int width = imgin->header.width, height = imgin->header.height;
      double row, col;
      double weightSum = 0.0, redSum = 0.0, greenSum = 0.0, blueSum = 0.0;

      #pragma omp parallel for num_threads(x) schedule(runtime) private(row,
      col, i, j) firstprivate(weightSum, redSum, greenSum, blueSum)

      for (i = 0; i < height; i++)
      {
            for (j = 0; j < width ; j++)
            {
                  for (row = i-radius; row <= i + radius; row++)
                  {
                        for (col = j-radius; col <= j + radius; col++)
                        {
                              int x = clamp(col, 0, width-1);
                              int y = clamp(row, 0, height-1);
                              int tempPos = y * width + x;
                              double square = (col-j)*(col-j)+(row-i)*(row-i);
                              double sigma = radius*radius;
                              double weight = exp(-square / (2*sigma)) /
(3.14*2*sigma);

                              redSum += imgin->red[tempPos] * weight;
                              greenSum += imgin->green[tempPos] * weight;
                              blueSum += imgin->blue[tempPos] * weight;
                              weightSum += weight;
                        }
                  }
                  imgout->red[i*width+j] = round(redSum/weightSum);
                  imgout->green[i*width+j] = round(greenSum/weightSum);
                  imgout->blue[i*width+j] = round(blueSum/weightSum);

                  redSum = 0;
                  greenSum = 0;
                  blueSum = 0;
                  weightSum = 0;
            }
      }
}
```

β) Για την παραλληλοποίηση του σειριακού κώδικα με OpenMP tasks, έκανα την ακόλουθη τροποποίηση (τονισμένη με πορτοκαλί χρώμα):

```
void gaussian_blur_omp_tasks(int radius, img_t *imgin, img_t *imgout)
{
      int i, j;
      int width = imgin->header.width, height = imgin->header.height;
      double row, col;
      double weightSum = 0.0, redSum = 0.0, greenSum = 0.0, blueSum = 0.0;

      omp_set_dynamic(0);

    #pragma omp parallel num_threads(x)
    {
        #pragma omp single
        {
            for (i = 0; i < height; i++)
            {
                # pragma omp task firstprivate(i) private(j, row, col, weightSum,
redSum, greenSum, blueSum)

                for (j = 0; j < width ; j++)
                {
                    for (row = i-radius; row <= i + radius; row++)
                    {
                        for (col = j-radius; col <= j + radius; col++)
                        {
                            int x = clamp(col, 0, width-1);
                            int y = clamp(row, 0, height-1);
                            int tempPos = y * width + x;
                            double square = (col-j)*(col-j)+(row-i)*(row-i);
                            double sigma = radius*radius;
                            double weight = exp(-square / (2*sigma)) /
(3.14*2*sigma);

                            redSum += imgin->red[tempPos] * weight;
                            greenSum += imgin->green[tempPos] * weight;
                            blueSum += imgin->blue[tempPos] * weight;
                            weightSum += weight;
                        }
                    }
                    imgout->red[i*width+j] = round(redSum/weightSum);
                    imgout->green[i*width+j] = round(greenSum/weightSum);
                    imgout->blue[i*width+j] = round(blueSum/weightSum);

                    redSum = 0;
                    greenSum = 0;
                    blueSum = 0;
                    weightSum = 0;
                }
            }
        }
    }
}
```

Ακολουθούν αναλυτικές αιτιολογήσεις για τις προσθήκες που έκανα στον κώδικα στην επόμενη σελίδα:

α) gaussian_blur_omp_loops:

- **omp_set_dynamic(0):**

  Πάλι, ο λόγος που πρόσθεσα την άνω εντολή στον κώδικα ήταν προκειμένου να εξαναγκάσω την χρήση του ακριβή αριθμού νημάτων που θέτω στην παράλληλη περιοχή.

- **#pragma omp parallel for num_threads(x) schedule(runtime) private(row, col, i, j) firstprivate(weightSum, redSum, greenSum, blueSum):**

  Καταρχάς, επέλεξα πάλι parallel for, καθώς παρατήρησα ότι και αυτή τη φορά το for loop επιδέχεται διαχωρισμό σε όσα x νήματα επιθυμούμε χωρίς επιπλοκές.

  Έπειτα, όσον αφορά τα schedules, όλα τα schedules που δοκίμασα (πάλι, δοκίμασα και τις 4 διαφορετικές επιλογών από schedules, και στα 3 schedules που επιδέχονται custom chunk sizes δοκίμασα και διάφορα διαφορετικά μεγέθη) είχαν πολύ κοντινές επιδόσεις, με αποτέλεσμα κανένα να μην βγαίνει καθαρός «νικητής» όπως στην πρώτη άσκηση. Έτσι, επέλεξα schedule(runtime), καθώς έκρινα ότι μιας και δεν υπήρχαν διαφορές, θα ήταν προτιμότερο ένα schedule που θα διαμορφωνόταν στη δημιουργία του προγράμματος, ανάλογα με τις συνθήκες που επικρατούν τότε.

  Ύστερα, επέλεξα private για τις μεταβλητές row, col, i και j, καθώς πάλι γίνεται εύκολα αντιληπτό από την πορεία του προγράμματος ότι κάθε νήμα χρειάζεται τα δικά του αντίγραφα των μεταβλητών για το παράλληλο κομμάτι. Επίσης, επέλεξα firstprivate για τις μεταβλητές weightSum, redSum, greenSum, blueSum, καθώς κάθε νήμα θα πρέπει να ξεκινάει με αρχικές τιμές των προαναφερθέντων μεταβλητών τις μεταβλητές που αναφέρονται πριν την for, δηλαδή 0 για όλες τους.

  Εδώ δεν συνάντησα κάποια δυσκολία, και έφτασα στην λύση την οποία θεώρησα βέλτιστη χωρίς πολλούς πειραματισμούς.

β) gaussian_blur_omp_tasks:

- **omp_set_dynamic(0):**

  Και πάλι, ο λόγος που πρόσθεσα την άνω εντολή στον κώδικα ήταν προκειμένου να εξαναγκάσω την χρήση του ακριβή αριθμού νημάτων που θέτω στην παράλληλη περιοχή.

- **#pragma omp parallel num_threads(x) & #pragma omp single:**

  Σε αυτό το σημείο ακολούθησα την κλασική μεθοδολογία ανάθεσης tasks (με μια σειριακή διάσχιση του for από το πρώτο νήμα που βρει την #pragma omp single εντολή), όπως ακριβώς παρατηρείται και στο σύγγραμμα του κ. Δημακόπουλου, «Παράλληλα Συστήματα και Προγραμματισμός», στην ενότητα 4.10.1 (διάσχιση λίστας), καθώς και στις διαφάνειες του μαθήματος. Μιας και η οδηγία ήταν να παραλληλοποιήσουμε τους υπολογισμούς κάθε γραμμής (και το πρώτο for διασχίζει κάθε γραμμή), έθεσα τόσα tasks όσο και οι γραμμές του προγράμματος, και κάθε task είχε το αντίστοιχο πρώτο εμφωλευμένο for, όπως ακριβώς διαμορφωνόταν με βάση το κάθε αντίστοιχο i.

- **# pragma omp task firstprivate(i) private(j, row, col, weightSum, redSum, greenSum, blueSum):**

  Εδώ επέλεξα την i ως firstprivate, καθώς είναι αναγκαίο κάθε νήμα να έχει τη δική του μεταβλητή i, με την ακριβή τιμή που είχε τη στιγμή που δημιουργήθηκε το task. Επίσης, κρίνω προφανές ότι κάθε task χρειάζεται private μεταβλητές για όλες τις λοιπές μεταβλητές του task, προκειμένου να αποφευχθούν συγχύσεις μεταξύ νημάτων.

Πειραματικά Αποτελέσματα - Μετρήσεις

Και αυτές οι μετρήσεις έγιναν όλες στον υπολογιστή που αναφέρθηκε στην εισαγωγή της αναφοράς, και η χρονομέτρηση έγινε μέσω της συνάρτησης omp_get_wtime(), που προσφέρει το OpenMP. Μέτρησα τη σειριακή υλοποίηση, και μέτρησα αντίστοιχους χρόνους στην παράλληλη περιοχή, για 1, 2, 3 και 4 νήματα αντίστοιχα. Επίσης, για μεγαλύτερη ακρίβεια αποτελεσμάτων, έγιναν 4 μετρήσεις για κάθε είδος νηματοποίησης και διατήρησα τον μέσο όρο. Ακολουθούν γραφικά τα στατιστικά:

Ο πίνακας με τις αναλυτικές μετρήσεις (ο χρόνος είναι σε δευτερόλεπτα):

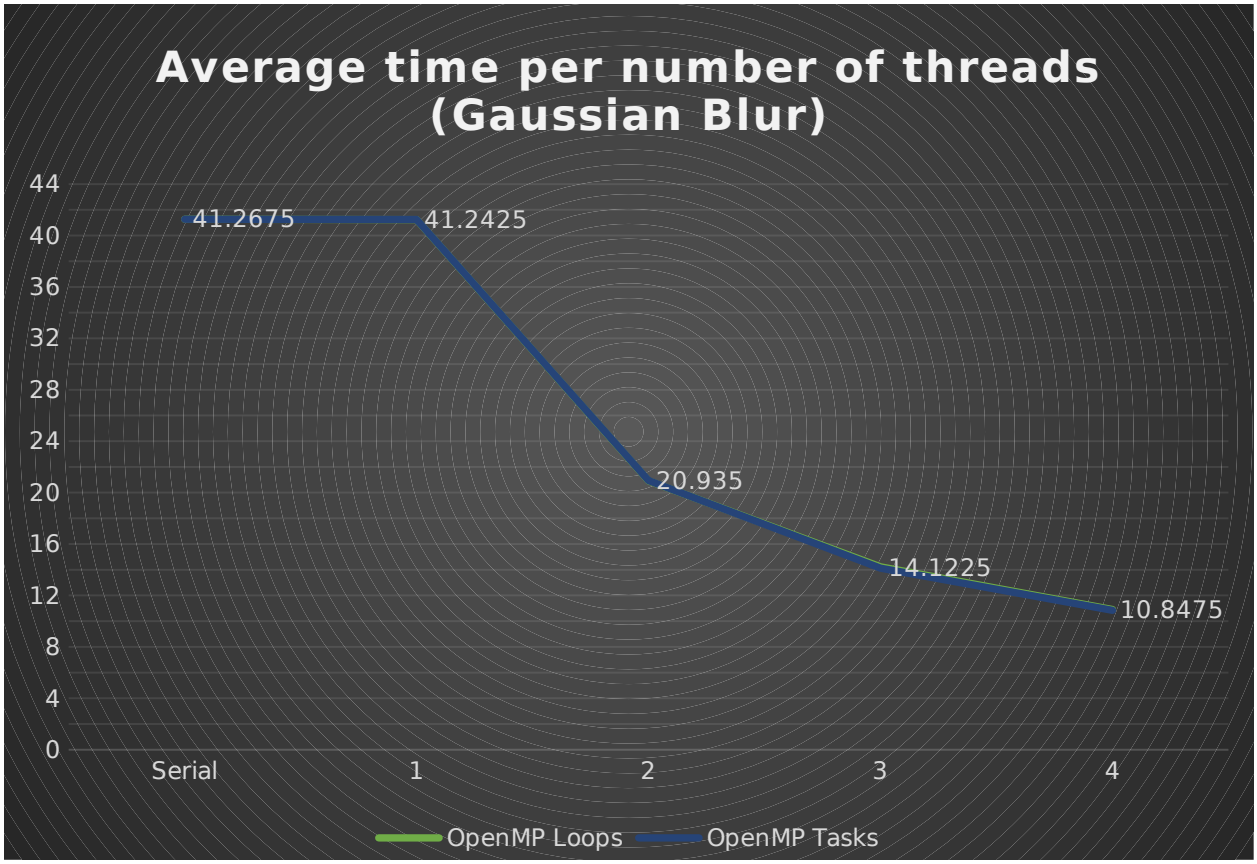
OpenMP Loops:

Number of Threads	First Run	Second Run	Third Run	Fourth Run	Average
Serial	41.26	41.24	41.3	41.27	41.2675
1	41.25	41.21	41.26	41.21	41.2325
2	20.94	20.84	20.9	20.99	20.9175
3	14.14	14.11	14.51	14.17	14.2325
4	10.89	10.88	10.77	11.06	10.9

OpenMP Tasks:

Number of Threads	First Run	Second Run	Third Run	Fourth Run	Average
Serial	41.26	41.24	41.3	41.27	41.2675
1	41.22	41.22	41.26	41.27	41.2425
2	20.85	20.94	20.97	20.98	20.935
3	14.12	14.13	14.11	14.13	14.1225
4	10.75	10.77	10.82	11.05	10.8475

Και από τους 2 παραπάνω πίνακες, κρατώντας τους μέσους όρους, προκύπτει το εξής line chart:



Σχόλια

Στο συγκεκριμένο πρόβλημα, μου έκανε πολύ μεγάλη εντύπωση πόσο μικρή ήταν η διαφορά μεταξύ tasks και loops. Θεωρώ ότι αυτό εξηγείται λόγω του ότι, εκ φύσεως, είναι ένα πρόβλημα που δεν είναι τόσο καλά «δομημένο» για tasks, το οποίο εξηγεί τις μηδαμινές διαφορές (τόσο μηδαμινές που στο line chart η διαφορά μετά βίας φαινόταν, και έπρεπε να αυξήσω κατά πολύ τον κάθετο άξονα προκειμένου να γίνει έστω και λίγο εμφανής). Επίσης, παρατηρείται ότι όσο αυξάνονται τα νήματα, τα tasks είναι ελάχιστα καλύτερα, αν και η διαφορά είναι τόσο μικρή που ενδέχεται ακόμα και να εξαλειφόταν εάν πραγματοποιούσαμε πολλές παραπάνω δοκιμές.

Κατά τα άλλα, τα αποτελέσματα όσον αφορά το speedup του προγράμματος είναι πλήρως αναμενόμενα. Για k νήματα, παρατηρείται πάλι βελτίωση πολύ κοντά στο 1/k σε σχέση με τη σειριακή εκτέλεση, το οποίο είναι προφανώς το βέλτιστο σενάριο όταν παραλληλοποιούμε με k νήματα.