Εργασία 2 (Java Threads).

Πολλαπλασιασμός Πινάκων (ΠΠ)

Το πρόβλημα του ΠΠ είναι πιθανότατα το περισσότερο μελετημένο πρόβλημα παράλληλου υπολογισμού. Αποτελεί τη βάση πολλών υπολογισμών (εικόνες, σήματα, επιστημονικοί υπολογισμοί). Έχει πολύ απλή δομή αλλά είναι υπολογιστικά απαιτητικό (θεωρούμε μόνο τετραγωνικούς πίνακες μεγέθους NxN. Το κόστος του ΠΠ είναι N^3).

Το επισυναπτόμενο αρχείο mm.pdf δίνει τις γενικές κατευθύνσεις για την άσκηση.

Αν γράψετε στο Google "multithreaded matrix multiplication in java" θα βρείτε πολλές λύσεις και πολύ κώδικα για να μελετήσετε.

Στο αρχείο zip που επισυνάπτεται δίνονται κώδικες και reports από ένα διαγωνισμό της Intel για ΠΠ. Δεν έχουν πιθανώς άμεση εφαρμογή αλλά γενικότερο ενδιαφλερον.

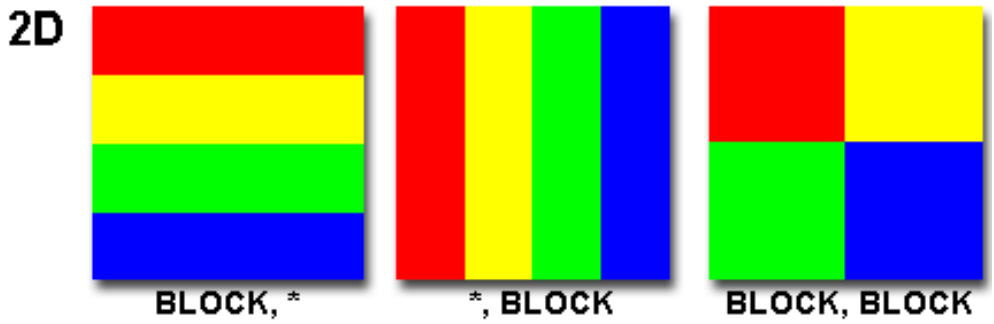
Ο στόχος είναι να επιτύχουμε το καλυτερο δυνατό χρόνο εκτέλεσης δίνοντας έμφαση στη Τοπικότητα των Αναφορών στη Μνήμη (Locality of Refernce). Πειραματιστείτε με

(α) Χρήση τοπικής μεταβλητής για την αποθήκευση των ενδιαμέσων αποτελεσμάτων του εσωτερικο γινομένου C[ij] = C[ij]+...

(α) Τη σειρά εκτέλεσης των loops i,j. k

(γ) Την αποθήκευση του ανάστροφου B αντί του B, ώστε οι A,B να σαρώνονται και οι δύο κατά γραμμή (όχι κατά στήλη ο B).

(δ) Διαφοτερικές αναθέσεις του πίνακα των αποτελεσμάτων όπως φαίνεται παρακάτω. Η block μορφή έχει ιδιαίτερο ενδιαφέρον γιατί επιτρέπει αναδρομική διαίρεση.



Πειράματα:

Αριθμός Νημάτων (Cores): 2, 4, 8, 16
Μέγεθος N: 100, 200, 400, 800

Προσπαθείστε να πετύχετε το καλύτερο χρόνο

Οδηγίες

1. Ξεκινείστε από έναν ακολουθιακό κώδικα σαν αυτό που βρίσκεται στο mm.pdf

```
for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {
            for (k=0; k<N; k++) {
                  c[i][j] += a[i][k]*b[k][j];
            }
      }
}
```

Εφαρμόστε τις αλλαγές (α)-(γ) και μετρήστε χρόνους ώστε πετύχετε το καλύτερο χρόνο. Αν βρείτε κάποια άλλη βελτιστοποίηση εφαρμόστε την.

2. Στον καλύτερο ακολουθιακό αλγόριθμο εφαρμόστε πολυνηματικό κώδικα όπως αυτός που βρίσκεται στο mm.pdf

```
Thread 0
for (i=0; i<N/2; i++) {
      for (j=0; j<N; j++) {
            for (k=0; k<N; k++) {
                  c[i][j] += a[i][k]*b[k][j];
            }
      }
}
Thread 1
for (i=N/2; i<N; i++) {
            for (j=0; j<N; j++) {
                  for (k=0; k<N; k++) {
                        c[i][j] += a[i][k]*b[k][j];
            }
      }
}
```

Δοκιμάστε κατανομή κατά γραμμές ή στήλες και διαφορετικό μέγεθος και αριθμό νημάτων.

3. Η block προσέγγιση είναι προαιρετική! Μπορεί να βασιστείτε στο κώδικα που ακολουθεί . Υπάρχει και αναδρομική λύση!

```
#define min(a,b) (((a)<(b))?(a):(b))

/* This auxiliary subroutine performs a smaller dgemm operation
 *  C := C + A * B
 * where C is M-by-N, A is M-by-K, and B is K-by-N. */
static void do_block (int lda, int M, int N, int K, double* A, double* B,
double* C)
{
  /* For each row i of A */
  for (int i = 0; i < M; ++i)
    /* For each column j of B */
    for (int j = 0; j < N; ++j)
    {
      /* Compute C(i,j) */
      double cij = C[i+j*lda];
      for (int k = 0; k < K; ++k)
        cij += A[i+k*lda] * B[k+j*lda];
      C[i+j*lda] = cij;
    }
```

```
}

/* This routine performs a dgemm operation
 *  C := C + A * B
 * where A, B, and C are lda-by-lda matrices stored in column-major format.
 * On exit, A and B maintain their input values. */
void square_dgemm (int lda, double* A, double* B, double* C)
{
  /* For each block-row of A */
  for (int i = 0; i < lda; i += BLOCK_SIZE)
    /* For each block-column of B */
    for (int j = 0; j < lda; j += BLOCK_SIZE)
      /* Accumulate block dgemms into block of C */
      for (int k = 0; k < lda; k += BLOCK_SIZE)
      {
        /* Correct block dimensions if block "goes off edge of" the matrix */
        int M = min (BLOCK_SIZE, lda-i);
        int N = min (BLOCK_SIZE, lda-j);
        int K = min (BLOCK_SIZE, lda-k);

        /* Perform individual block dgemm */
        do_block(lda, M, N, K, A + i + k*lda, B + k + j*lda, C + i + j*lda);
      }
}
```