ΕΡΓΑΛΕΙΑ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

ΕΙΣΑΓΩΓΗ ΣΤΟΝ ΠΑΡΑΛΛΗΛΟ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟ ΜΕ OpenMP

Νίκος Τρυφωνίδης

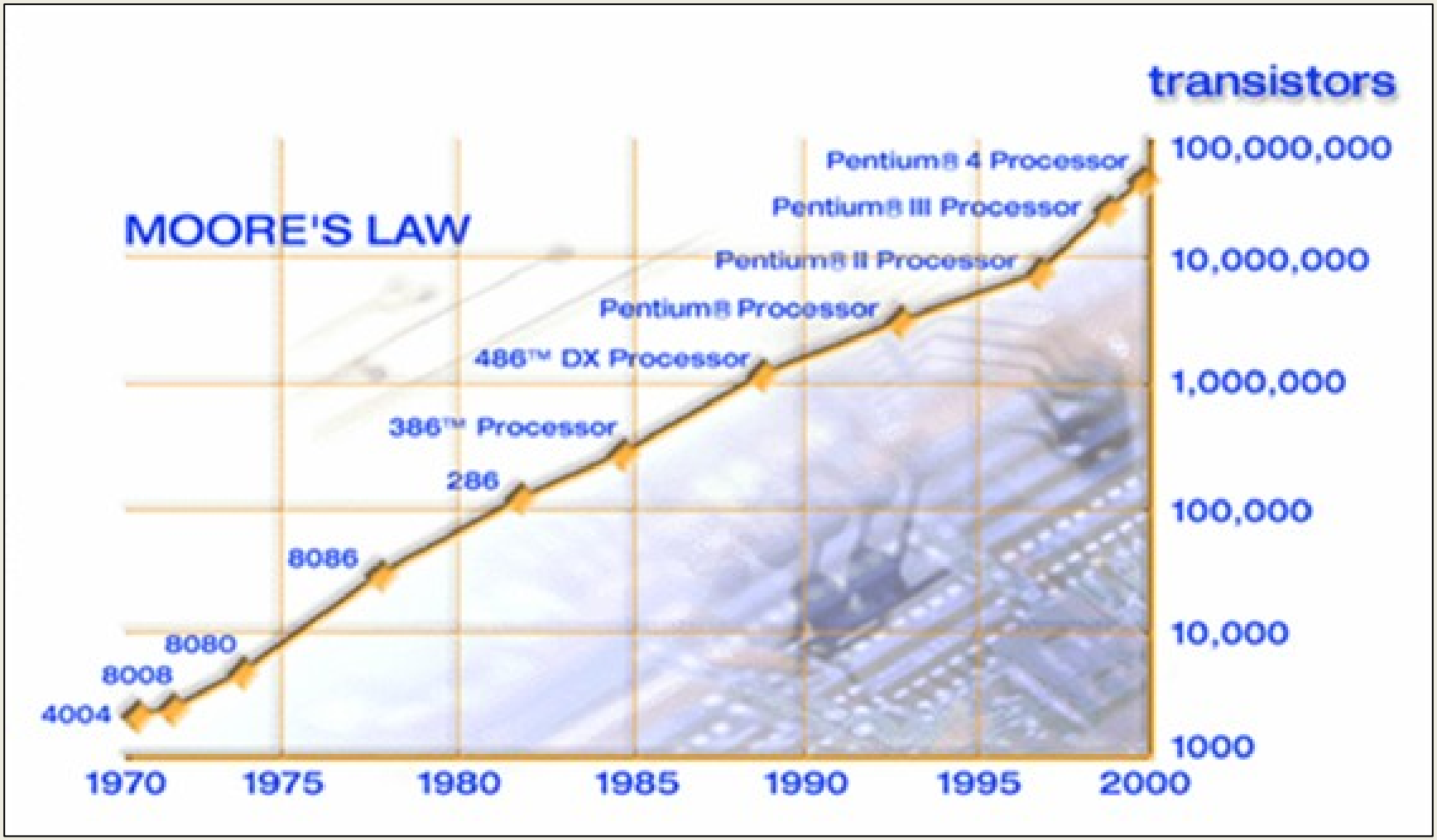
Μέρος 1ο:

Η ΑΝΑΓΚΗ ΓΙΑ ΠΑΡΑΛΛΗΛΟ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟ

Γιατί Παράλληλος Προγραμματισμός;

- Οι επιστημονικές υπολογιστικές εφαρμογές είναι πλέον βασικό κομμάτι στις θετικές επιστήμες (και όχι μόνο).
- Οι εφαρμογές αυτές είναι συνήθως ιδιαίτερα απαιτητικές σε υπολογιστικούς πόρους.
- Υπάρχει ανάγκη για όσο το δυνατό περισσότερους υπολογιστικούς πόρους (ταχύτητα, μνήμη).

Νόμος του Moore



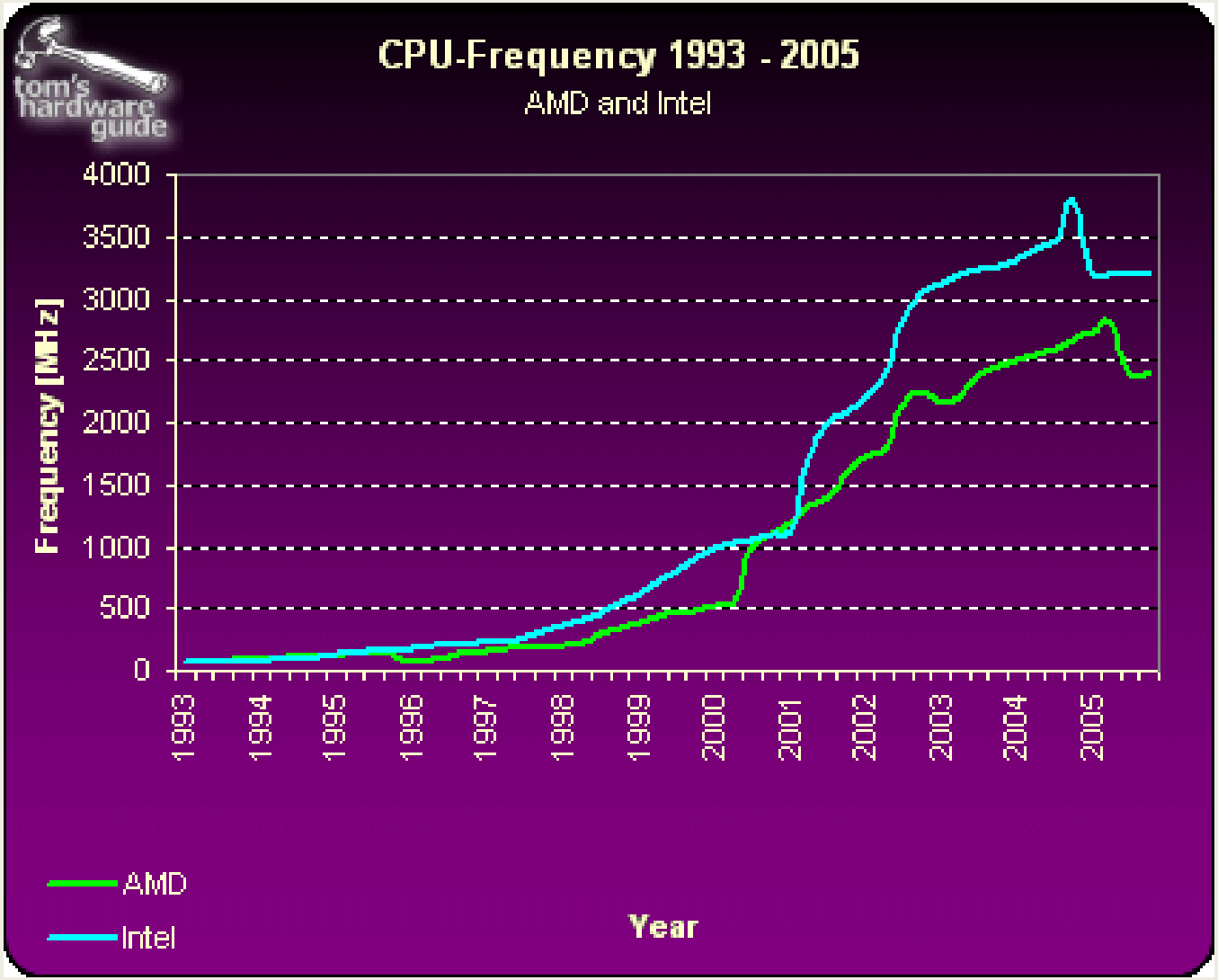
Νόμος του Moore

- Ο αριθμός των transistors σε ένα ολοκληρωμένο κύκλωμα (μπορεί να) διπλασιάζεται (περίπου) κάθε δύο χρόνια.

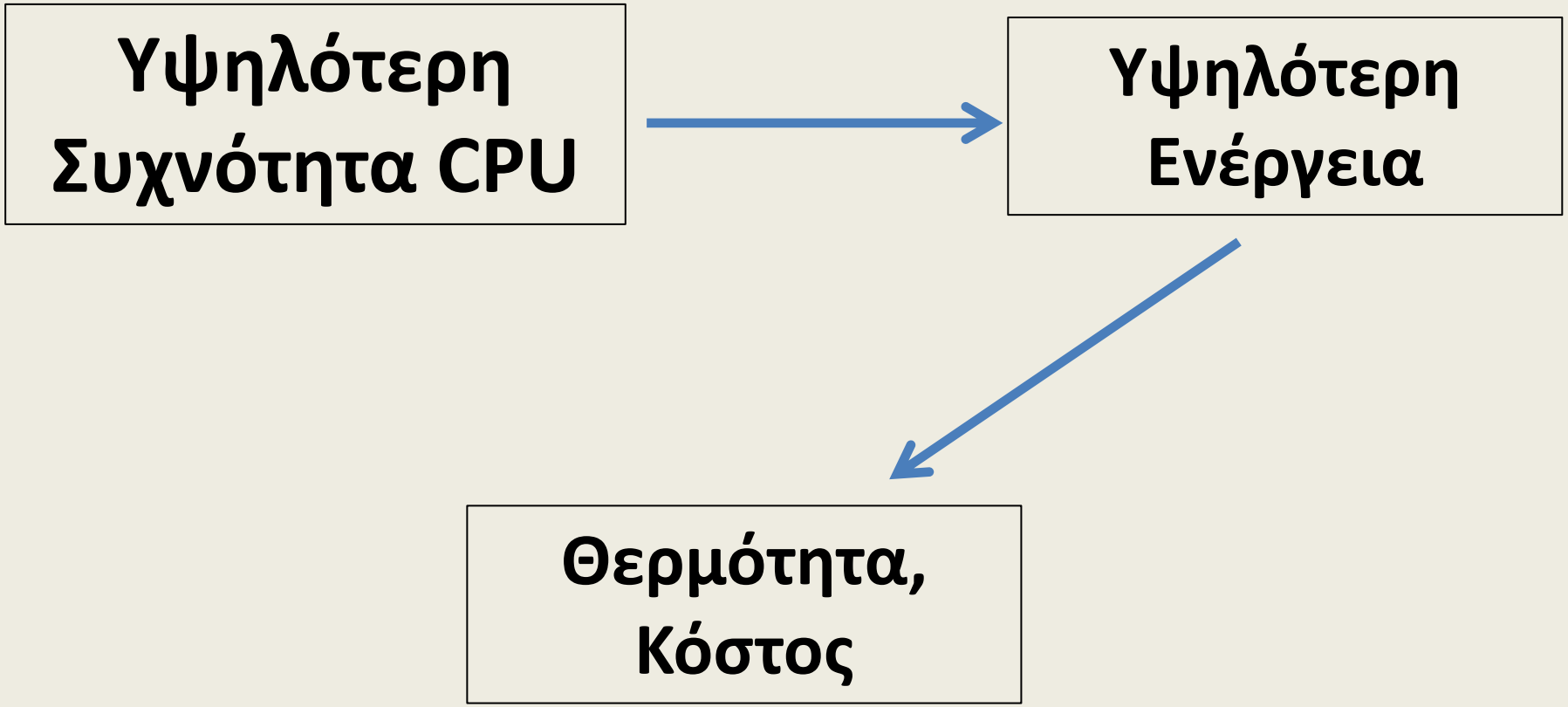
- Μέχρι κάποιο σημείο:

Περισσότερα transistors → **Υψηλότερη Συχνότητα CPU**

Συχνότητα CPU – Χρόνος



Το πρόβλημα;


Υψηλότερη Συχνότητα CPU
Υψηλότερη Ενέργεια
Θερμότητα, Κόστος

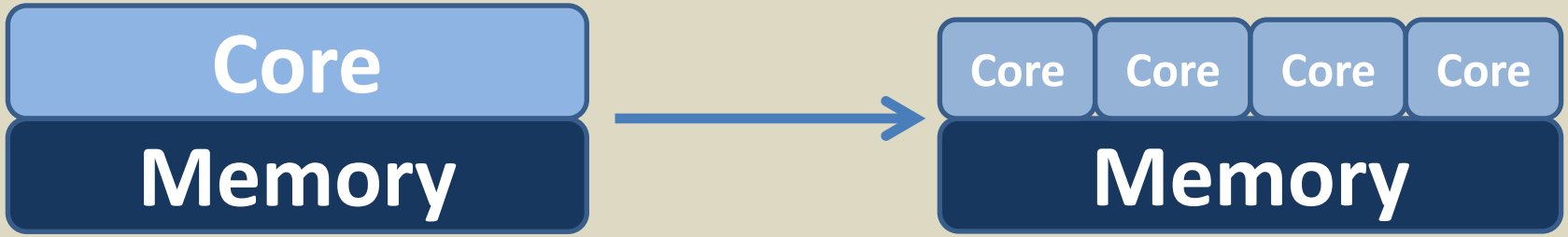
Η Λύση (;)

- Πώς μπορούν να χρησιμοποιηθούν τα διαρκώς αυξανόμενα transistors ενός CPU;
- Αντί για ***έναν*** επεξεργαστή/πυρήνα υψηλής συχνότητας, μπορούμε να έχουμε ***πολλούς*** πυρήνες χαμηλότερης συχνότητας.

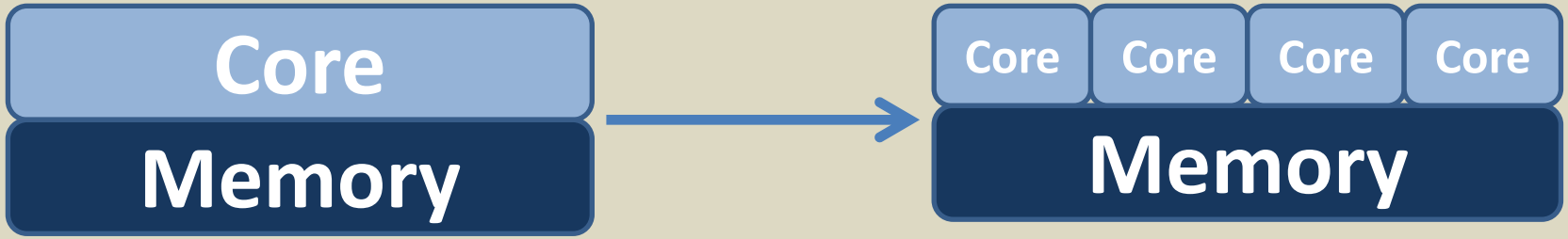
Η Λύση (;)

- Πώς μπορούν να χρησιμοποιηθούν τα διαρκώς αυξανόμενα transistors ενός CPU;

- Αντί για ***έναν*** επεξεργαστή/πυρήνα υψηλής συχνότητας, μπορούμε να έχουμε ***πολλούς*** πυρήνες χαμηλότερης συχνότητας.



Multi-Core CPUs



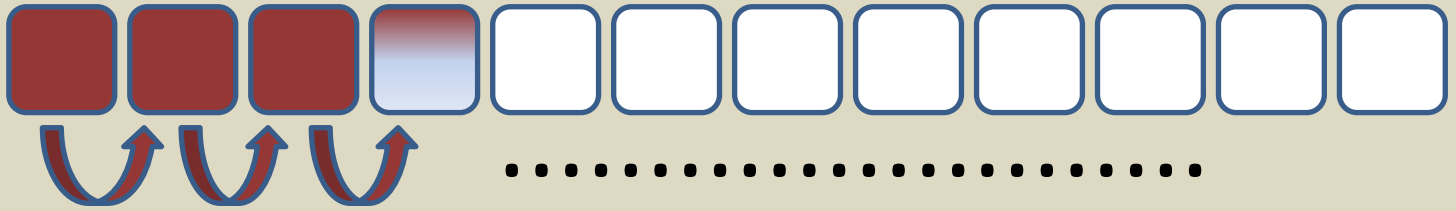
- Με αυτόν τον τρόπο, *δεν κερδίζουμε υψηλότερη ταχύτητα υπολογισμών*, αλλά έχουμε τη **δυνατότητα εκτέλεσης πολλών υπολογισμών ταυτόχρονα**.
- Σήμερα κυκλοφορούν αποκλειστικά πολυπύρηνοι επεξεργαστές.

Χρήση των Multi-Core CPUs

- Συνηθισμένη χρήση υπολογιστή: το λειτουργικό σύστημα κατανέμει τις διάφορες διεργασίες του στους διαφορετικούς πυρήνες.
- Πώς μπορούμε να χρησιμοποιήσουμε τους πολλούς πυρήνες ενός επεξεργαστή για υπολογιστικές εφαρμογές;

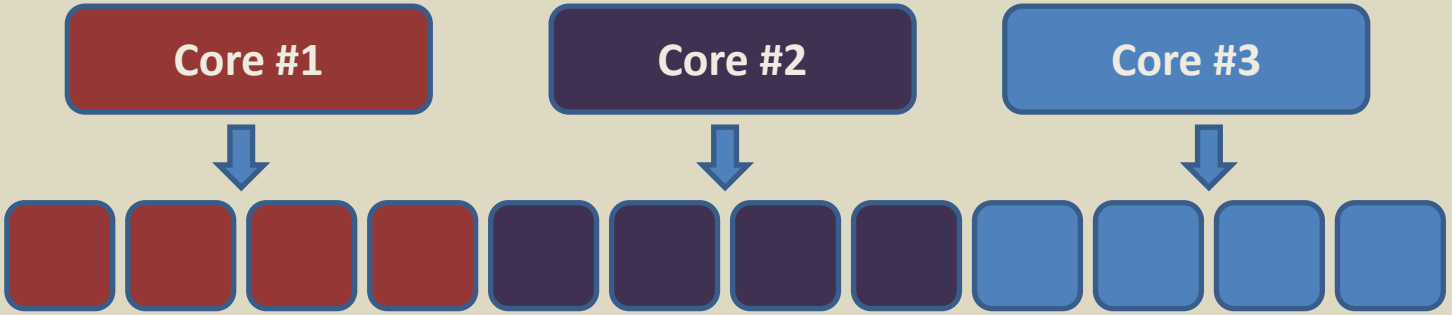
Χρήση Shared Memory

- Ας υποθέσουμε ότι έχουμε να εκτελέσουμε κάποιον υπολογισμό στα στοιχεία ενός πίνακα
- Σειριακά (με έναν πυρήνα), ο επεξεργαστής πηγαίνει από το πρώτο στοιχείο μέχρι το τελευταίο, εκτελώντας τους υπολογισμούς:



Χρήση Shared Memory

- Με περισσότερους πυρήνες, μπορούμε να μοιράσουμε τη δουλειά και οι πυρήνες να εκτελέσουν τους υπολογισμούς ***παράλληλα***.
- Για παράδειγμα, με 3 πυρήνες:



Παράλληλος Προγραμματισμός

- Η ανάπτυξη προγραμμάτων για παράλληλη εκτέλεση αποτελεί πλέον ξεχωριστό κλάδο της υπολογιστικής επιστήμης.
- Τεράστια χρηματικά ποσά ξοδεύονται για δημιουργία και συντήρηση υπερυπολογιστών, που περιέχουν εκατοντάδες χιλιάδες επεξεργαστές.
- Λίστα με τους 500 ισχυρότερους υπερυπολογιστές στον κόσμο: www.top500.org

Πώς προγραμματίζονται οι υπερυπολογιστές;

- Βασικό θέμα στον προγραμματισμό πολλαπλών πυρήνων είναι ο διαμοιρασμός της εργασίας και ο συγχρονισμός μεταξύ των πυρήνων.
- Υπάρχουν τρία βασικά μοντέλα παράλληλου προγραμματισμού:
  1. Message-Passing Programming
  2. Shared-Memory Programming
  3. Accelerators (GPUs)

Shared-Memory Programming

- Το Shared-Memory Programming είναι ίσως το απλούστερο μοντέλο παράλληλου προγραμματισμού.
- Μπορεί να χρησιμοποιηθεί εύκολα χωρίς επιπλέον απαιτούμενο υλικό.
- Το OpenMP, με το οποίο θα ασχοληθούμε, είναι ίσως το πιο γνωστό και δημοφιλές περιβάλλον για shared-memory programming.

Μέρος 2ο:

ΕΙΣΑΓΩΓΗ ΣΤΟ OPENMP: THREADS, PARALLEL REGIONS

Τί Είναι Το OpenMP;

- Το **OpenMP** (Open Multiprocessing) είναι ένα περιβάλλον προγραμματισμού (**API**) που αποτελείται από εντολές, συναρτήσεις και μεταβλητές.

- Υποστηρίζεται από τις δύο δημοφιλέστερες γλώσσες προγραμματισμού για υπολογιστικές εφαρμογές (**C/C++** και **FORTRAN**).

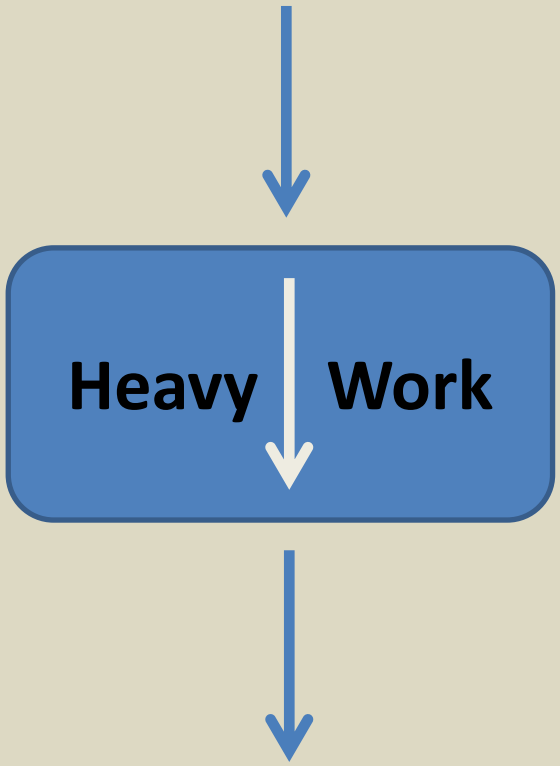
Τί Είναι Το OpenMP;

- Το OpenMP χρησιμοποιείται ουσιαστικά ως **προσθήκη** σε ένα πρόγραμμα.

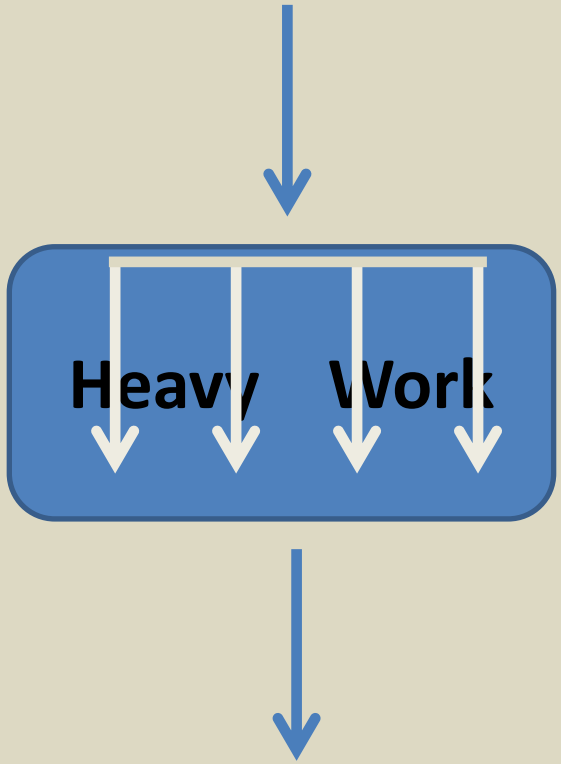
- Με τις κατάλληλες εντολές, τα ιδιαίτερα «βαριά» σημεία του προγράμματος μπορούν να εκτελεστούν παράλληλα από πολλούς πυρήνες, εφόσον αυτοί υπάρχουν.

OpenMP: Βασική Λειτουργία

- **Serial** *(1 Core)*

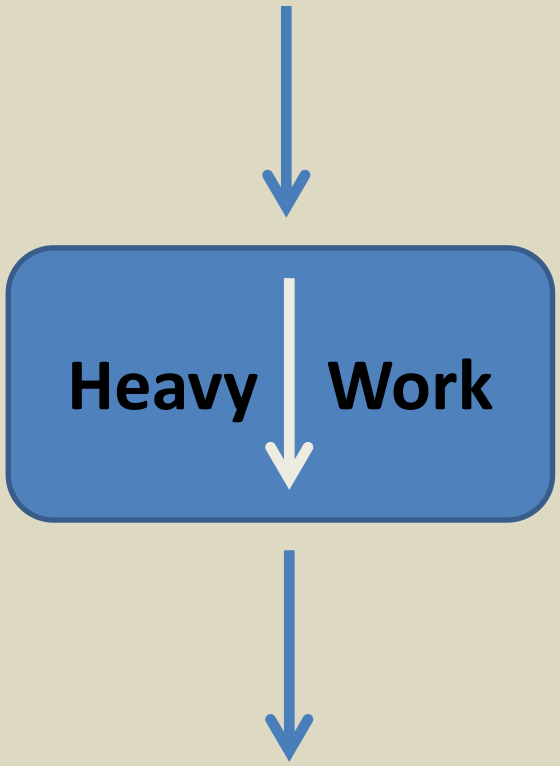


- **Parallel** *(4 Cores)*

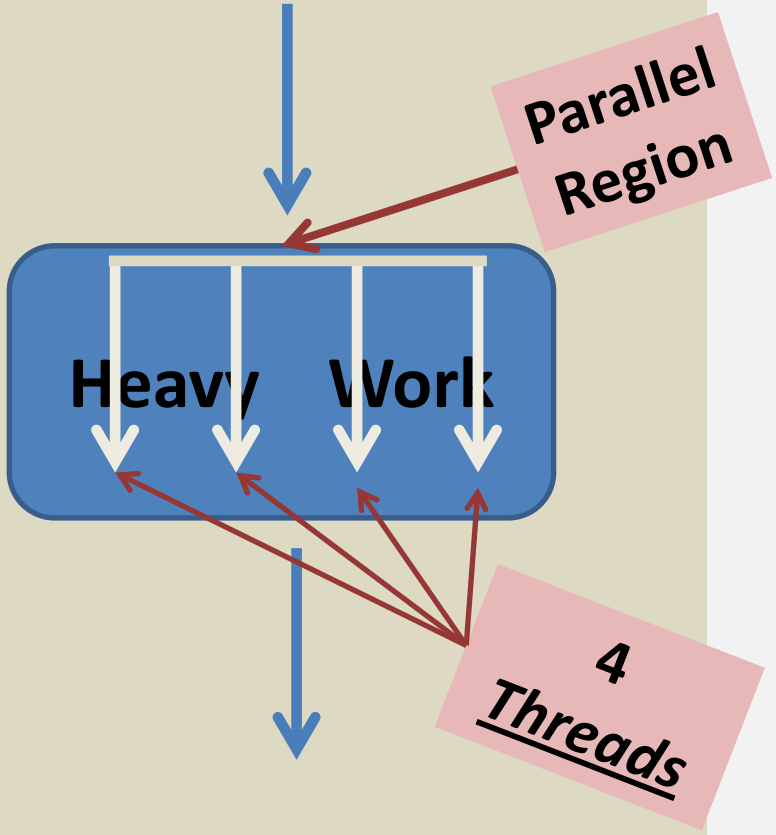


OpenMP: Βασική Λειτουργία

- **Serial** *(1 Core)*



- **Parallel** *(4 Cores)*



OpenMP: Parallel Regions

- Οι περιοχές στον κώδικα που ορίζουμε για να εκτελεστούν παράλληλα, λέγονται ***Parallel Regions***.
- Κατά την εκτέλεση ενός προγράμματος, μόλις ο επεξεργαστής συναντήσει μια parallel region, δημιουργεί μια ομάδα από ***threads***, τα οποία θα εκτελέσουν **παράλληλα** τον κώδικα της parallel region.

OpenMP: Parallel Regions

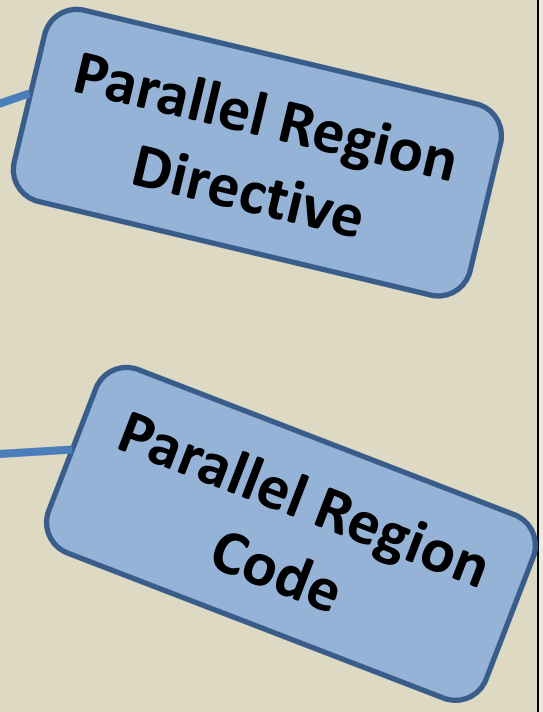
- Οι parallel regions ορίζονται στον κώδικα με ειδικά ***compiler directives***, ως εξής:

```
#pragma omp parallel
{
  (parallel region code)
}
```

OpenMP: Parallel Regions

- Οι parallel regions ορίζονται στον κώδικα με ειδικά ***compiler directives***, ως εξής:

```
#pragma omp parallel
{
  (parallel region code)
}
```



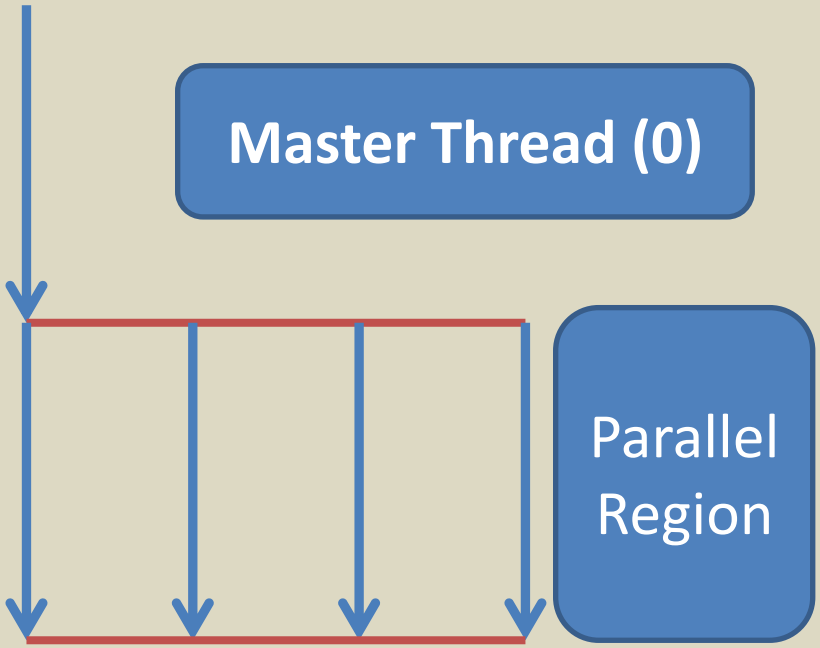
OpenMP: Παράδειγμα (4 Threads)

```
int main () {
 /* (Serial Code) */
 (…)

 /* Parallel Region */
 #pragma omp parallel
 {
   (Parallel Code)
 }
  /* (Serial Code) */
 (…)
}
```

Master Thread (0)

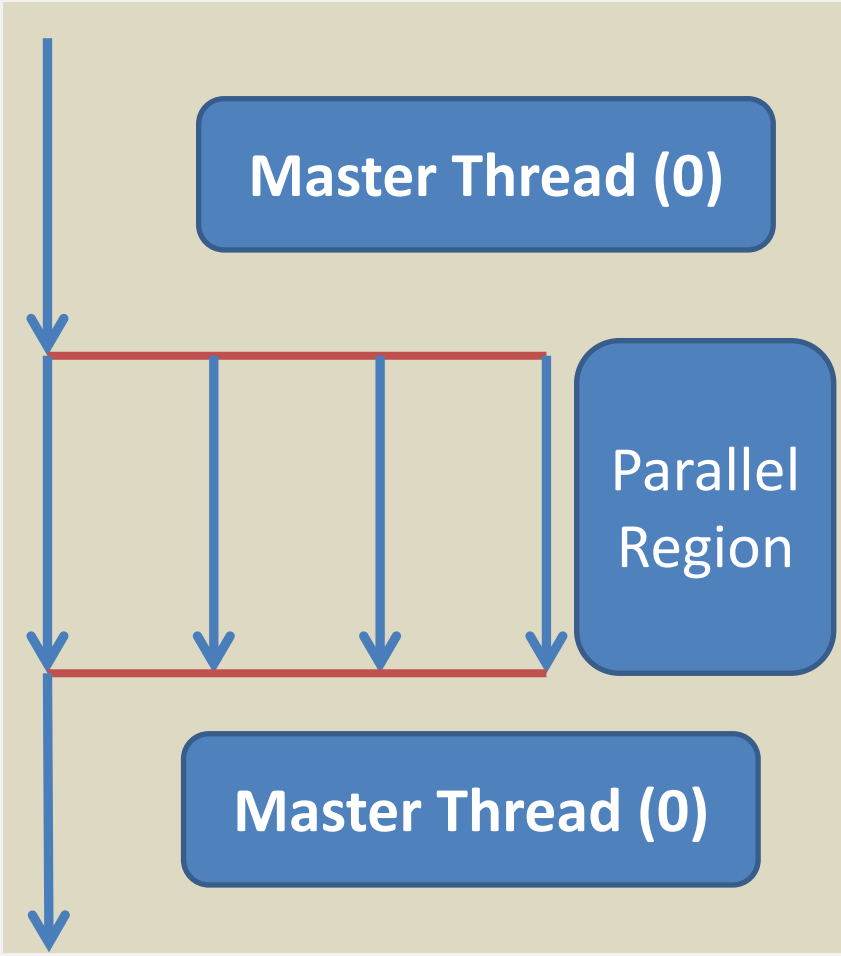
OpenMP: Παράδειγμα (4 Threads)

```
int main () {
 /* (Serial Code) */
 (…)

 /* Parallel Region */
 #pragma omp parallel
 {
   (Parallel Code)
 }
  /* (Serial Code) */
 (…)
}
```



OpenMP: Παράδειγμα (4 Threads)

```
int main () {
 /* (Serial Code) */
 (…)

 /* Parallel Region */
 #pragma omp parallel
 {
   (Parallel Code)
 }
  /* (Serial Code) */
 (…)
}
```



“Hello World” … Παράλληλα!

Serial

```
#include <stdio.h>


int main() {


 printf(“Hello World!\n”);

 return 0;
}
```

Parallel

```
#include <stdio.h>
#include <omp.h>

int main() {
 #pragma omp parallel
 {
  printf(“Hello World!\n”);
 }
 return 0;
}
```

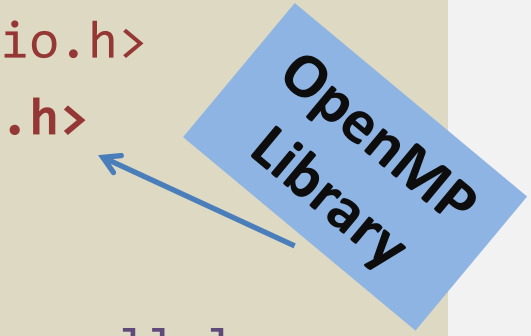
“Hello World” … Παράλληλα!

Serial

```
#include <stdio.h>


int main() {


 printf(“Hello World!\n”);

 return 0;
}
```

Parallel

```
#include <stdio.h>
#include <omp.h>

int main() {
 #pragma omp parallel
 {
  printf(“Hello World!\n”);
 }
 return 0;
}
```



Compilation και Εκτέλεση

- Το compilation για ένα πρόγραμμα που χρησιμοποιεί OpenMP γίνεται με ένα επιπλέον flag στην εντολή. Για παράδειγμα, για το gcc έχουμε:

```
gcc hello.c –o hello –fopenmp
```

Οπότε δημιουργούμε το εκτελέσιμο *hello.*

Compilation και Εκτέλεση

- Πριν εκτελέσουμε ένα πρόγραμμα OpenMP πρέπει να ορίσουμε τον αριθμό των threads που θέλουμε να δημιουργηθούν.

Π.χ. για να τρέξουμε με 4 threads πληκτρολογούμε σε ένα terminal:

```
export OMP_NUM_THREADS=4
```

Compilation και Εκτέλεση

- Τέλος, εκτελούμε το πρόγραμμα ως συνήθως:

```
./hello
```

Compilation και Εκτέλεση

- Τέλος, εκτελούμε το πρόγραμμα ως συνήθως:

```
./hello
```

Και παίρνουμε:

```
Hello World!
Hello World!
Hello World!
Hello World!
```

“Hello World”: Δοκιμή

- Γράψτε ένα πρόγραμμα σαν το παραπάνω και εκτελέστε το με 1, 2, 4 και 8 threads.

Thread ID, Number of Threads

- Κάθε thread, τη στιγμή που δημιουργείται, έχει μια ταυτότητα (ακέραιος, από 0 εώς N-1).
- Το αρχικό (*master*) thread έχει τον αριθμό 0.
- Με βάση την ταυτότητα τους και τον συνολικό αριθμό των threads, μπορούμε να χωρίσουμε την υπολογιστική εργασία στα threads.

Thread ID

- Τα threads μπορούν να μάθουν την ταυτότητα τους καλώντας τη συνάρτηση:

  `omp_get_thread_num()`

  η οποία επιστρέφει τον ακέραιο αριθμό που αντιστοιχεί στην ταυτότητα του thread που την κάλεσε. Π.χ.

```
int myid = omp_get_thread_num();
```

Number of Threads

- Με χρήση μιας παρόμοιας συνάρτησης, τα threads μπορούν να μάθουν πόσα threads υπάρχουν συνολικά:

```
int nThreads = omp_get_num_threads();
```

Thread ID, Number of Threads

- Καλώντας τις δύο παραπάνω συναρτήσεις μέσα από μια parallel region, τα threads πλέον γνωρίζουν την ταυτότητα τους και το συνολικό αριθμό των threads.

Thread ID, Number of Threads

- Καλώντας τις δύο παραπάνω συναρτήσεις μέσα από μια parallel region, τα threads πλέον γνωρίζουν την ταυτότητα τους και το συνολικό αριθμό των threads.
- Δηλώνοντας μεταβλητές (π.χ. τις *myid* και *nThreads)* **εντός της parallel region**, το κάθε thread θα έχει από ένα **προσωπικό** αντίγραφο της κάθε μεταβλητής.

Παραλλαγή του “Hello”

- **Δοκιμή #2:** Προσαρμόστε το προηγούμενο πρόγραμμα (*hello*), ώστε τα threads να εκτυπώνουν, μετά το “Hello World”, την ταυτότητα τους και τον συνολικό αριθμό τους.

- **Δοκιμή #3:** Καλέστε την *printf* μέσω μιας ενδιάμεσης, δικής σας, συνάρτησης.

Συμπεράσματα

- Το πρόγραμμα σε OpenMP τρέχει με όσα threads έχουν οριστεί προηγουμένως με την environment variable **OMP_NUM_THREADS**.

- Η **σειρά** με την οποία τα threads εκτελούν τον κώδικα σε μια parallel region είναι ***εντελώς ακαθόριστη.***

Μέρος 3ο:

DATA SCOPE, ΑΠΛΗ ΠΑΡΑΛΛΗΛΗ ΕΡΓΑΣΙΑ ΣΕ SHARED MEMORY

Data Scope

- Οι μεταβλητές χωρίζονται σε **δύο βασικές κατηγορίες**, όσο αφορά το πώς φαίνονται σε ένα thread:
  1. **Shared**: Όλα τα threads μπορούν να διαβάσουν και να τροποποιήσουν τις shared μεταβλητές.
  2. **Private**: Κάθε thread δημιουργεί ένα προσωπικό αντίγραφο των private μεταβλητών, και μόνο εκείνο μπορεί να τις προσπελάσει.

Δήλωση του Data Scope

```
#pragma omp parallel shared(a) \
                     private(i) \
                     default(none)
```

Δήλωση του Data Scope

```
#pragma omp parallel shared(a) \
                     private(i) \
                     default(none)
```

- **a:** *shared*, προσβάσιμη από όλα τα threads.
- **i:** *private*, κάθε thread έχει το δικό του αντίγραφο (**απροσδιόριστη αρχική τιμή!**)

Δήλωση του Data Scope

```
#pragma omp parallel shared(a) \
                     private(i) \
                     default(none)
```

- **a:** *sha*… όλα τα threads.
- …ο δικό του
- …**αρχική τιμή!**)

Αναγκάζει το χρήστη να ορίσει ακριβώς το scope κάθε μεταβλητής στην parallel region (**προτείνεται**).

Καταμερισμός της Εργασίας

- Με βάση το ***Thread ID*** και το ***συνολικό αριθμό threads*** μπορούμε να μοιράσουμε την υπολογιστική εργασία στα threads.

- Έστω ότι κάθε thread αποθηκεύει το ID του και τον αριθμό των threads στις μεταβλητές **myid** και **nThreads**, αντίστοιχα.

Καταμερισμός της Εργασίας: Παράδειγμα

- Ας υποθέσουμε ότι θέλουμε να προσθέσουμε τον αριθμό 1 σε κάθε στοιχείο ενός πίνακα Ν στοιχείων **A[N]**, χρησιμοποιώντας **4 Threads.**

- Για να χωρίσουμε την εργασία στα 4 threads, θα ορίσουμε τον πίνακα Α ως **shared**, ώστε να είναι προσβάσιμος από όλα τα threads.

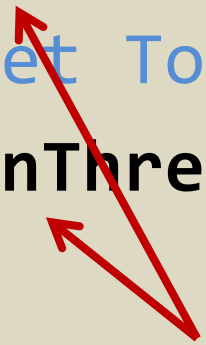
- Θα αναθέσουμε, έπειτα, σε κάθε thread ένα διαφορετικό κομμάτι του πίνακα. Τα threads θα εργαστούν στα κομμάτια που τους αντιστοιχούν **παράλληλα**.

Καταμερισμός: Εντός της Parallel Region

```
/* Get Thread ID */
int myid = omp_get_thread_num();
/* Get Total Number of Threads */
int nThreads = omp_get_num_threads();
```

Καταμερισμός: Εντός της Parallel Region

```
/* Get Thread ID */
int myid = omp_get_thread_num();
/* Get Total Number of Threads */
int nThreads = omp_get_num_threads();
```



Υπενθύμιση: Μεταβλητές που ορίζονται *εντός της parallel region* θεωρούνται **private**.

Καταμερισμός: Εντός της Parallel Region (Συνέχεια)

```
/* Each Thread's Starting Point */
int iStart = myid*N/nThreads;
/* Each Thread's Ending Point */
int iEnd = (myid+1)*N/nThreads;
```

Καταμερισμός: Εντός της Parallel Region (Συνέχεια)

```
/* Each Thread's Starting Point */
int iStart = myid*N/nThreads;
/* Each Thread's Ending Point */
int iEnd = (myid+1)*N/nThreads;
```

Για πίνακα μεγέθους N = 20 και 4 threads:

Thread 0:	**iStart = 0**	**iEnd = 5**
Thread 1:	**iStart = 5**	**iEnd = 10**
Thread 2:	**iStart = 10**	**iEnd = 15**
Thread 3:	**iStart = 15**	**iEnd = 20**

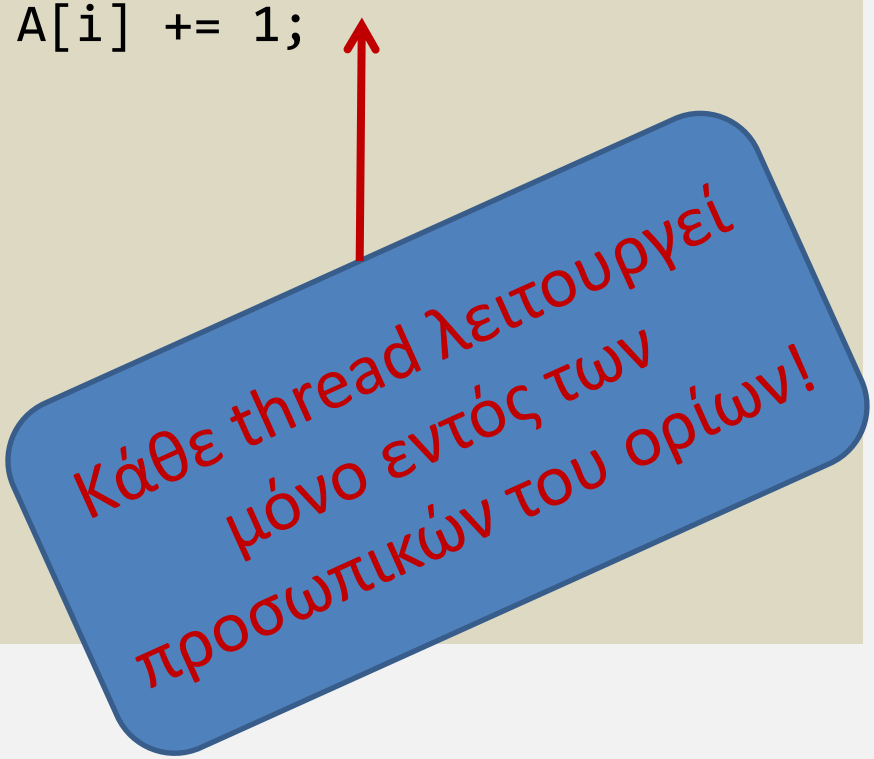
Μετατροπή του Loop σε Παράλληλο

Serial

```
for (i=0; i<N; i++) {
  A[i] += 1;
}
```

Parallel

```
for (i=iStart; i<iEnd; i++) {
  A[i] += 1;
}
```



Ψιλά Γράμματα 1:

- Για να καλύψουμε την περίπτωση που δε διαιρείται ακριβώς ο αριθμός των στοιχείων N με τον αριθμό των threads, θέτουμε το τέλος των iterations του τελευταίου thread ως εξής:

```
if (myid == nThreads - 1) iEnd = N;
```

Ψιλά Γράμματα 2:

- Πώς θα ορίζατε το Data Scope της μεταβλητής του μεγέθους του πίνακα N;

Ψιλά Γράμματα 2:

- Πώς θα ορίζατε το Data Scope της μεταβλητής του μεγέθους του πίνακα N;
- Μπορεί να είναι private, αλλά πρέπει να διατηρηθεί η αρχική του τιμή.

Ψιλά Γράμματα 2:

- Πώς θα ορίζατε το Data Scope της μεταβλητής του μεγέθους του πίνακα N;
- Μπορεί να είναι private, αλλά πρέπει να διατηρηθεί η αρχική του τιμή.
- **Λύση**: *firstprivate.* Κάθε thread έχει προσωπικό αντίγραφο με την αρχική τιμή της μεταβλητής.

Τελικά;

```
#pragma omp parallel \
                shared(A) \
                private(i) \
                firstprivate(N)
{
 /* Get Thread ID */
int myid = omp_get_thread_num();
/* Get Total Number of Threads */
int nThreads=omp_get_num_threads();

/* Each Thread's Starting Point */
int iStart = myid*N/nThreads;
/* Each Thread's Ending Point */
int iEnd = (myid+1)*N/nThreads;
```

```
/* continued… */
 /* Take care of bound */
 if(myid == nThreads-1) iEnd=N;

 /* The Loop */
 for (i=iStart; i<iEnd; i++) {
   A[i] += 1;
 }

} /* End of Parallel Region */
```

Εφαρμογή 1: Vector Addition

- Γράψτε ένα πρόγραμμα που να αρχικοποιεί δύο διανύσματα και να τα προσθέτει. Φροντίστε να επαληθεύεται το αποτέλεσμα.

- Αφού σιγουρευτείτε ότι το σειριακό πρόγραμμα λειτουργεί σωστά, μετατρέψτε το πρόγραμμα σε παράλληλο με χρήση OpenMP.

Μέρος 4ο:

ΑΥΤΟΜΑΤΟΣ ΚΑΤΑΜΕΡΙΣΜΟΣ ΕΡΓΑΣΙΑΣ (WORK-SHARING)

Work-Sharing

- Η παραπάνω διαδικασία χωρισμού της εργασίας σε threads μπορεί να γίνει αυτόματα με χρήση ειδικών directives, που λέγονται ***work-sharing constructs***.

- Ουσιαστικά, αυτό που επιτυγχάνουν τα work-sharing constructs είναι να **παραλληλοποιούν αυτόματα ένα loop** εντός μιας parallel region.

Work-Sharing

Serial

```
for (i=0; i<N; i++) {
    A[i] += 1;
}
```

Parallel – Work-Sharing

```
#pragma omp parallel \
  shared(A)private(i)\
     firstprivate(N)
{
  #pragma omp for
  for (i=0; i<N; i++) {
   A[i] += 1;
  }
}
```

Work-Sharing

Serial

```
for (i=0; i<N; i++) {
    A[i] += 1;
}
```

Ίδιο με το σειριακό loop!

Parallel – Work-Sharing

```
#pragma omp parallel \
  shared(A)private(i)\
     firstprivate(N)
{
  #pragma omp for
  for (i=0; i<N; i++) {
   A[i] += 1;
  }
}
```

Καθόλου επιπλέον κώδικας – μόνο directives!

Work-Sharing Schedules

- Το πώς θα μοιραστεί το loop στα threads εξαρτάται από το schedule που θα δηλώσουμε δίπλα στο work-sharing directive.

- Υπάρχουν 3 διαφορετικά schedules:
  - Static
  - Dynamic
  - Guided

Static Schedule

- Μοιράζει τις επαναλήψεις του loop **εξίσου** στα threads:

```
#pragma omp for schedule(static)
```

4 Threads, 16 Iterations:
4 iterations/Thread

Static Schedule

- Δηλώνοντας έναν αριθμό (*chunksize*):

```
#pragma omp for schedule(static,2)
```

4 Threads, 16 Iterations:
4 iterations/Thread
σε ομάδες των 2 iterations

Dynamic Schedule

- Μοιράζει αρχικά **n** επαναλήψεις στα threads

```
#pragma omp for schedule(dynamic,n)
```

- Μόλις κάποιο thread ολοκληρώσει τις επαναλήψεις του, παίρνει τις επόμενες n επαναλήψεις.

***Σημείωση**: το default dynamic chunksize είναι 1*

Guided Schedule

- Όμοιο με το dynamic, αλλά ξεκινά να μοιράζει μεγάλα κομμάτια επαναλήψεων, τα οποία διαρκώς μειώνει.

```
#pragma omp for schedule(guided,n)
```

n: Ο ελάχιστος αριθμός επαναλήψεων

(default: 1)

Work-Sharing Schedules

- Το **static** είναι η απλούστερη και συνήθως αποτελεσματικότερη μέθοδος, επειδή απαιτεί τον ελάχιστο συγχρονισμό από το σύστημα.

- Το **dynamic** επιβαρύνει το σύστημα, αλλά είναι χρήσιμο σε περιπτώσεις που ορισμένα μέρη ενός loop είναι βαρύτερα από άλλα (load imbalance).

Parallel Region + Work-Sharing

- Έναρξη parallel region και work-sharing με το ίδιο directive:

```
#pragma omp parallel for \
            shared(…) private(…) \
             schedule(…)
```

Ψιλά Γράμματα #3: Synchronization

- Στο τέλος κάθε *parallel* και *work-sharing* region υπάρχει ***implicit barrier.***

- Αυτό σημαίνει ότι **όλα τα threads πρέπει να φτάσουν στο σημείο εκείνο**, πριν συνεχιστεί η εκτέλεση.

Εφαρμογή 2:
Vector Addition + Work-Sharing

- Παραλληλοποιήστε το πρόγραμμα πρόσθεσης διανυσμάτων του προηγούμενου μέρους, κάνοντας χρήση **work-sharing**.
- Συγκρίνετε την απόδοση των δύο παράλληλων προγραμμάτων και δοκιμάστε διαφορετικά schedules.

Μέρος 5ο:

THREAD SAFETY

Thread Safety?

- Μέχρι τώρα, στα παραδείγματα που είδαμε, τα threads δουλεύουν αποκλειστικά στο δικό τους κομμάτι μνήμης.
- Δεν υπήρχε κίνδυνος να μεταβάλλουν μια μεταβλητή ταυτόχρονα.
- Αν κάτι τέτοιο συμβεί, το αποτέλεσμα είναι **απροσδιόριστο** και προφανώς **λάθος**.

Thread Safety

- Τα προγράμματα OpenMP πρέπει να είναι *thread-safe,* δηλαδή η εκτέλεση ενός προγράμματος με πολλά threads δεν πρέπει εισάγει σφάλματα στο πρόγραμμα.

Thread-Unsafe Παράδειγμα

```
sum = 0;
#pragma omp parallel shared(sum, a) private(i)\
                     firstprivate(size)
{
  #pragma omp for schedule(static)
  for(i=0; i<size; i++) {
    sum += a[i];
  }
}
printf("sum = %f\n", sum);
```

Thread-Unsafe Παράδειγμα

```
sum = 0;
#pragma omp parallel shared(s
                            firstpr
{
  #pragma omp for schedule(
  for(i=0; i<size; i++) {
    sum += a[i];
  }
}
printf("sum = %f\n", sum);
```

Τα threads προσπαθούν να γράψουν στη μεταβλητή sum **ταυτόχρονα**!

Αποτέλεσμα: απροσδιόριστο!

Critical Regions

- Θέτοντας μια thread-unsafe περιοχή της parallel region ως ***critical region***, εξασφαλίζουμε ότι **μόνο ένα thread μπορεί να εκτελεί τις εντολές της συγκεκριμένης περιοχής σε μια συγκεκριμένη στιγμή**.

- Τα υπόλοιπα threads περιμένουν τη σειρά τους εκτός της critical region.

Χρήση της Critical Region

```
sum = 0;
#pragma omp parallel shared(sum,
                            firstpriv
{
  #pragma omp for schedule(sta
  for(i=0; i<size; i++) {
    #pragma omp critical
    {
      sum += a[i];
    }
  }
}
printf("sum = %f\n", sum);
```

Τα threads γράφουν στη sum ένα-ένα:
Σωστό αποτέλεσμα!

Critical Regions

- Οι critical regions είναι ένας εύκολος τρόπος να εξασφαλίσουμε τη σωστή συνργασία των threads.
- Όμως, επιβαρύνουν ιδιαίτερα τη λειτουργία ενός προγράμματος (απαιτούν πολύ συγχρονισμό).
- Είναι προτιμότερο, αν γίνεται, να αντικαθίστανται με άλλους τρόπους.

Αντικατάσταση Μιας Critical Region

- Στο προηγούμενο παράδειγμα, προσθέσαμε τα στοιχεία ενός πίνακα σε μια μεταβλητή.

- Για να αποφύγουμε τη χρήση critical region, η συνηθισμένη λύση θα ήταν η χρήση ενός **προσωρινού πίνακα (size = nThreads)**.

Αντικατάσταση Μιας Critical Region

- Κάθε thread θα γράψει στο αντίστοιχο στοιχείο του προσωρινού πίνακα το **προσωπικό του άθροισμα.**

- **Μετά την parallel region**, το master thread θα προσθέσει σειριακά τα επιμέρους αθροίσματα.

Χρήση του Reduction Scope

- Η παραπάνω διαδικασία είναι τόσο συχνή που έχει αυτοματοποιηθεί στο OpenMP, με χρήση του reduction scope.

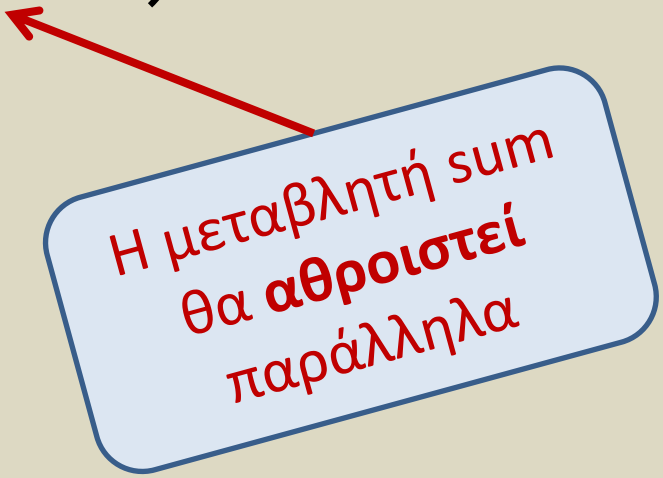
- Το μόνο που έχουμε να κάνουμε είναι να ορίσουμε τη μεταβλητή στην οποία θα αθροίζουμε ως ***reduction.***

Reduction Example

```
sum = 0;
#pragma omp parallel shared(a) private(i)\
                    firstprivate(size) \
                     reduction(+:sum)
{
  #pragma omp for schedule(static)
  for(i=0; i<size; i++) {
      sum += a[i];
  }
}
printf("sum = %f\n", sum);
```

Χρήση του Reduction Scope

```
reduction(+:sum)
```



- Άλλα reductions:
  - `-` (initialized: 0)
  - `*` (initialized: 1)

Εφαρμογή 3: Dot Product

- Γράψτε ένα πρόγραμμα OpenMP που να υπολογίζει παράλληλα το εσωτερικό γινόμενο δύο διανυσμάτων με χρήση work-sharing.

- Αφού δείτε ότι αν δηλώσετε τη μεταβλητή sum ως shared το αποτέλεσμα είναι απρόβλεπτο, κάντε χρήση critical region για να το προστατέψετε. Μετρήστε το χρόνο εκτέλεσης.

Εφαρμογή 3: Dot Product

- Κάντε χρήση του reduction. Βεβαιωθείτε ότι το αποτέλεσμα είναι σωστό και μετρήστε το χρόνο εκτέλεσης.

Εφαρμογή 3: Dot Product

- **Επιπλέον Άσκηση (Δύσκολο):**

  Γράψτε το ίδιο πρόγραμμα εφαρμόζοντας το reduction μη-αυτόματα (με shared temporary array). Συγκρίνετε το χρόνο εκτέλεσης με το αυτόματο.

Μέρος 6ο:

ΕΠΙΠΛΕΟΝ ΕΦΑΡΜΟΓΕΣ

Εφαρμογή 4: Matrix Multiplication

- Γράψτε ένα πρόγραμμα που να πολλαπλασιάζει δύο (τετράγωνους) πίνακες παράλληλα. Συγκρίνετε το χρόνο εκτέλεσης με το σειριακό αντίστοιχο.

Εφαρμογή 5: Μέσος Όρος - Απόκλιση

- Γράψτε ένα πρόγραμμα που να υπολογίζει παράλληλα το μέσο όρο και την απόκλιση των στοιχείων ενός πίνακα, τα οποία προέρχονται από κανονική κατανομή.

Εφαρμογή 6: Εύρεση Μεγίστου

- Γράψτε ένα πρόγραμμα που να αναζητεί και να βρίσκει το μέγιστο από τα στοιχεία ενός πίνακα παράλληλα.

- **Επιπλέον εφαρμογή (δύσκολο):** Κάντε το προηγούμενο πρόγραμμα χωρίς critical region.

Υλικό OpenMP

- http://openmp.org/wp/
- https://computing.llnl.gov/tutorials/openMP/

- Βιβλία:
  - **Using OpenMP** (Barbara Chapman, Gabriele Jost and Ruud van der Pas)
  - **Parallel Programming In OpenMP** (Chandra, Dagum et al)

Ευχαριστώ!

Νίκος Τρυφωνίδης