Parallel Programming with OpenMP

Computer Science Department, University of Crete

Parallel Programming

Based on presentations by D.Nikolopoulos, Tim Mattson

Τι είναι το OpenMP

- “Open Multi-Processing”
- Πρότυπο (standard) API για την ανάπτυξη παράλληλων εφαρμογών
- Βασίζεται σε κοινή μνήμη
- C, C++, Fortran
- Αποτελείται από:
  - Compiler directives
  - Run time routines
  - Environment variables
- Specification που ορίζεται από το OpenMP Architecture Review Board (http://www.openmp.org)

Τι δεν είναι

- Όχι για προγραμματισμό με κατανεμημένες μνήμες
- Δεν είναι υλοποιημένο παντού με τον ίδιο τρόπο
- Δεν εγγυάται την πιο αποδοτική χρήση της κοινής μνήμης
- Δεν ελέγχει για εξαρτήσεις, data races, συγχρονισμό που λείπει, κλπ.
- Δεν μπορεί να χειριστεί παράλληλο I/O

Παράδειγμα

- For με ανεξάρτητες επαναλήψεις

```
for (i = 0; i < n; i++)
  c[i] = a[i] + b[i];
```

- For με παράλληλες επαναλήψεις σε OpenMP

```
#pragma omp parallel for shared(n, a, b, c) private(i)
for (i = 0; i < n; i++)
  c[i] = a[i] + b[i];
```

Συντακτικό

- Οι περισσότερες δομές του OpenMP είναι compiler directives

```
#pragma omp construct [clause [clause] ...]
```

- Οι δηλώσεις συναρτήσεων και τύπων βρίσκονται στο αρχείο

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

- Οι περισσότερες δομές του OpenMP αντιστοιχούν σε κάποιο structured code block
  - Block κώδικα που έχει ένα σημείο εισόδου στην πρώτη και ένα σημείο εξόδου μετά την τελευταία εντολή
  - Επιτρέπεται η έξοδος και με `exit()`

Hello world

```
#include <stdio.h>

int main()
{


    int ID = 0;
    printf("   hello(%d) ",   ID);
    printf("   world(%d) \n", ID);

}
```

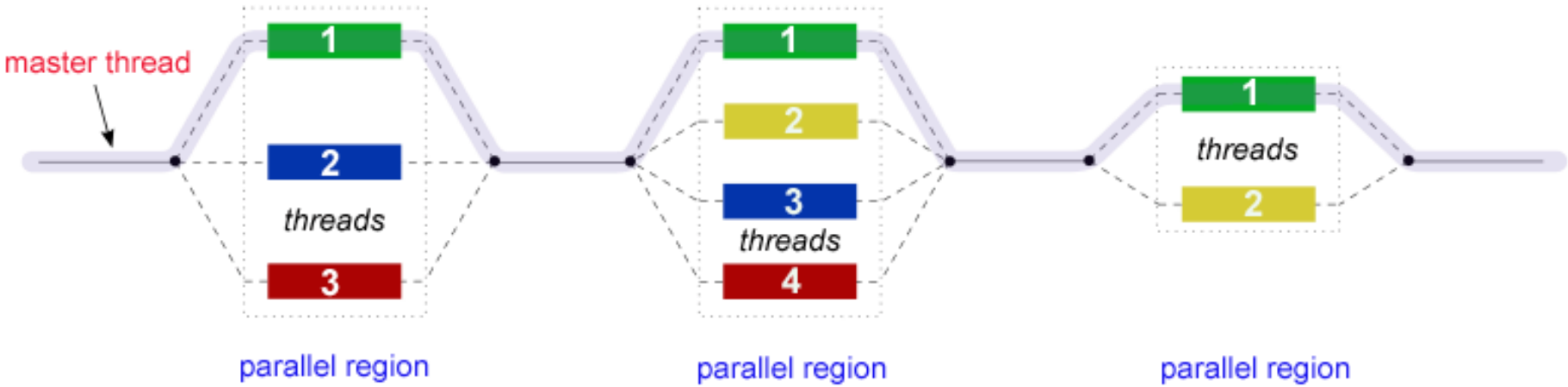
Hello world

```
#include <stdio.h>
#include <omp.h>
int main()
{
#pragma omp parallel
  {
    int ID = omp_get_thread_num();
    printf(”   hello(%d) ”,   ID);
    printf(”   world(%d) \n”, ID);
  }
}
```

Compilation

```
> gcc -fopenmp foo.c
> export OMP_NUM_THREADS=4
> ./a.out
  hello(0)    world(0)
  hello(1)    world(1)
  hello(3)    world(3)
  hello(2)    world(2)
> ./a.out
  hello(0)    world(0)
  hello(3)    world(3)
  hello(2)    hello(1)    world(1)
  world(2)
```

Fork-Join μοντέλο παραλληλισμού



Όροι

- OpenMP Team: Master + Workers
- “Παράλληλο region”: block κώδικα που εκτελείται από όλα τα threads ταυτόχρονα
  - Το ID του master thread είναι πάντα 0
  - Ο αριθμός thread μπορεί να αλλάξει μόνο μπαίνοντας σε ένα parallel region
  - Τα parallel regions μπορεί να είναι φωλιασμένα, αλλά δεν είναι απαραίτητο ότι θα εκτελεστούν έτσι
  - Μπορεί με χρήση ενός “if” να γίνει παραλληλισμός υπό προϋποθέσεις ή κώδικας που εκτελείται σειριακά
- “Work sharing”: μοιράζεται η εκτέλεση του κώδικα μεταξύ όλων των μελών του OpenMP Team

Στοιχεία του OpenMP

- Directives
  - Parallel regions
    - `if`: Παραλληλισμός υπό συνθήκη
    - `num_threads`: Αριθμός threads που θα εκτελέσουν το region
  - Work sharing
  - Synchronization
  - Data-sharing attributes
    - `private`: Νέα μεταβλητή για κάθε thread
    - `firstprivate`: με αρχικοποίηση
    - `lastprivate`: μετά το region κρατά την τιμή που γράφτηκε στο τελευταίο iteration
    - `shared`: Κοινή μνήμη, ο συγχρονισμός είναι ευθύνη του προγραμματιστή
    - `reduction`:
    - `threadprivate`: Νέα μεταβλητή για κάθε thread, δυναμική δέσμευση, persistent
    - `copyin`: Αρχικοποίηση `threadprivate`

Στοιχεία του OpenMP

- Environment Variables
  - Number of threads
  - Scheduling algorithm
  - Dynamic thread adjustment
  - Nested parallelism
- Runtime Environment
  - Number of threads
  - Thread ID
  - Dynamic thread adjustment
  - Nested parallelism
  - Timers
  - API for locks, barriers

OpenMP παραλληλισμός

- Multi-threading, shared memory
  - Τα threads επικοινωνούν γράφοντας κοινή μνήμη
- Race conditions
  - Όπως και στα pthreads
- Συγχρονισμός
  - Barrier, locks, κλπ
  - Κοστίζει πολύ
- Επειδή ο συγχρονισμός είναι ακριβός:
  - Directives που ελέγχουν την πρόσβαση στη μνήμη
  - Αποφυγή της ανάγκης για συγχρονισμό

Προγραμματιστικό Μοντέλο

- Η εκτέλεση μοιράζεται σε όλα τα threads
- Πρέπει να είναι “μέσα” σε parallel region
- Πρέπει να εκτελεστεί από όλα τα threads στο team ή κανένα
- Όχι barrier κατά την έναρξη, μόνο κατά τη λήξη
  - Εκτός αν υπάρχει `nowait`
- Οι δομές παραλληλισμού δεν ξεκινούν νέα threads
- Για συντομία, δομές που ξεκινούν μαζί με το parallel region

```
#pragma omp parallel
{
#pragma omp for
  for (...)
}
```

```
#pragma omp parallel for
for (...)
```

Παράδειγμα

```
#pragma omp parallel default (none) \
  shared (n,a,b,c,d) private(i)
  {
#pragma omp for nowait
    for (i=0; i<n-1; i++)
      b[i] = (a[i] + a[i+1]) / 2;
#pragma omp for nowait
    for (i=0; i<n; i++)
      d[i] = 1.0/c[i];
  } /*-- End of parallel region --*/
```

Reduction

- Η δομή `reduction` ορίζει πώς πολλά αντίγραφα μιας τοπικής μεταβλητής συνδυάζονται σε μία μεταβλητή που θα υπάρχει στο master thread στο τέλος
- Συντακτικό: `reduction (operator: variable list)`
- Οι μεταβλητές στη λίστα μεταβλητών αυτόματα δηλώνονται ως private στα threads
- Ο `operator` μπορεί να είναι μια από τις πράξεις:
  - +, *, -, &, |, ^, &&, ||
- `ordered`: Οι πράξεις θα γίνουν με τη σειριακή σειρά

```
#pragma omp parallel reduction (+: sum) num_threads(8) {
  /* compute local sums here */
}
/* sum here contains sum of all instances of sums */
```

Παράδειγμα OpenMP

```
/**********************************************
  An OpenMP version of a threaded program to compute PI.
 **********************************************/
#pragma omp parallel default(private) shared (npoints) \
  reduction(+: sum) num_threads(8)
  {
    num_threads = omp_get_num_threads();
    sample_points_per_thread = npoints / num_threads;
    sum = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
      rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
      rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
      if  (((rand_no_x -0.5) * (rand_no_x -0.5) +
         (rand_no_y -0.5) * (rand_no_y -0.5)) < 0.25)
      {
        sum ++;
      }
    }
  }
```

Συγχρονισμός

- Το OpenMP δίνει τα abstractions για να χωρίσει ο προγραμματιστής τη μνήμη σε thread-local κομμάτια
  - single/master: υπολογισμός ανά thread
  - Patterns επικοινωνίας (reduction)
- Μερικές φορές αυτό δε γίνεται
  - Επικοινωνία μεταξύ threads
- Δομές συγχρονισμού
  - critical: μόνο ένα thread κάθε φορά
  - atomic: μόνο για εγγραφή στη μνήμη
  - barrier: ραντεβού των threads του parallel section
  - ordered: περιορισμός στη σειρά
  - locks: low-level συγχρονισμός (όπως στα pthreads)
  - flush: μηχανισμός memory consistency

Παράδειγμα sections

- Κομμάτια κώδικα που θα εκτελεστούν παράλληλα, από 1 thread το καθένα

```
#pragma omp parallel default (none) \
  shared (n,a,b,c,d) private(i)
  {
#pragma omp sections nowait
    {
#pragma omp section
      for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1]) / 2;
#pragma omp section
      for (i=0; i<n; i++)
        d[i] = 1.0/c[i];
    } /*-- End of sections --*/
  } /*-- End of parallel region --*/
```

Single και Master

- `single`: Μόνο ένα thread θα εκτελέσει τον κώδικα

```
#pragma omp single [clause[[,] clause] ...]
{
  <code block>
}
```

- `master`: Μόνο το master thread θα εκτελέσει τον κώδικα

```
#pragma omp master [clause[[,] clause] ...]
{
  <code block>
}
```

- Χρησιμεύουν για κώδικα που πρέπει να τρέξει σε έναν επεξεργαστή, π.χ., I/O
- Δεν γίνεται barrier ούτε στην είσοδο ούτε στην έξοδο από αυτά τα blocks

Συγχρονισμός με Barriers

- `omp barrier`: Όλα τα thread του parallel section
- Εννοείται barrier στο τέλος του for

```
#pragma omp for
  for (i=0; i<N; i++)
    a[i] = c[i] + b[i];
/* --- Implicit barrier --- */
#pragma omp for
  for (i=0; i<N; i++)
    d[i] = a[i] + b[i];
/* --- Implicit barrier --- */
```

- Δεν χρειάζεται αν τα threads εκτελούν τα ίδια iterations στα 2 for

Αποφυγή barrier με nowait

- Για να μην περιμένουν threads όταν δεν χρειάζεται, `nowait`
- Όταν υπάρχει, τα threads που εκτελούν τη συγκεκριμένη δομή παραλληλισμού δεν συγχρονίζονται στο τέλος
- Μπορεί να γίνει συγχρονισμός σε άλλα σημεία με `barrier`

```
#pragma omp for nowait
  for (i=0; i<N; i++)
    a[i] = c[i] + b[i];
#pragma omp for nowait
  for (i=0; i<N; i++)
    d[i] = a[i] + b[i];
/* more work */
#pragma omp barrier
```

Παράδειγμα barrier

```
#pragma omp parallel if (n>limit) default (none) \
  shared (n,a,b,c,x,y,z) private (f,i,scale)
  {
    f = 1.0 /* --- Executed by all threads ---*/
#pragma omp for nowait
      for (i=0; i<n; i++) /* ---Parallel loop, work distributed---*/
        z[i] = x[i] + y[i];
#pragma omp for nowait
      for (i=0; i<n; i++) /* ---Parallel loop, work distributed---*/
        a[i] = b[i] + c[i];
#pragma omp barrier /*---Synchronization---*/
      /* more work */
    scale = sum (a,0,n) + sum (z,0,n) + f; /* ---Executed by all threads ---*/
  }
/*---End of parallel region---*/
```

Συγχρονισμός με critical

- Μόνο ένα thread μπορεί να εκτελέσει ένα `critical` region ανά πάσα στιγμή

```
float res;
#pragma omp parallel
{
  float B;
  int i, id, nthrds;
  id = omp_get_thread_num();
  nthrds = omp_get_num_threads();
  for(i=id;i<niters;i+nthrds){
    B = big_job(i); /* big jobs run in parallel */
#pragma omp critical
      consume (B, res);
      /* Only one thread at a time performs complex reduction */
  }
}
```

Συγχρονισμός με atomic

- Όπως το `critical`, μόνο ένα thread
- Μόνο για memory writes
- Πιο γρήγορο

```
#pragma omp parallel
{
  double tmp, B;
  B = doit();
  tmp = big_ugly(B);
#pragma omp atomic
    X += tmp; /* one thread at a time */
}
```

Συγχρονισμός με locks

- Low-level: ο προγραμματιστής έχει έλεγχο

  omp_init_lock(), omp_set_lock(),
  omp_unset_lock(), omp_test_lock(),
  omp_destroy_lock()

- Φωλιασμένα locks: το thread που έχει κλειδώσει μπορεί να ξανακλειδώσει

  omp_init_nest_lock(), omp_set_nest_lock(),
  omp_unset_nest_lock(), omp_test_nest_lock(),
  omp_destroy_nest_lock()

Κλήσεις στο runtime

- Αριθμός threads

  omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()

- Κώδικας σε εκτέλεση εντός parallel region

  omp_in_parallel()

- Μπορεί το runtime να αλλάζει αριθμό threads δυναμικά;

  omp_set_dynamic, omp_get_dynamic();

- Διαθέσιμοι επεξεργαστές

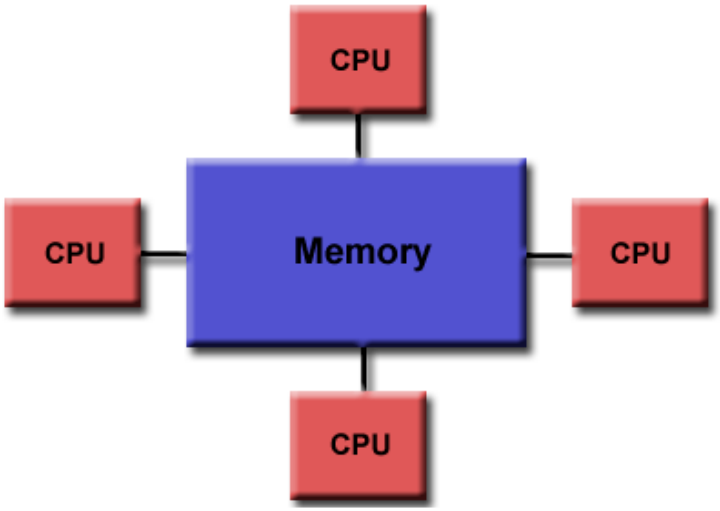
  omp_num_procs()

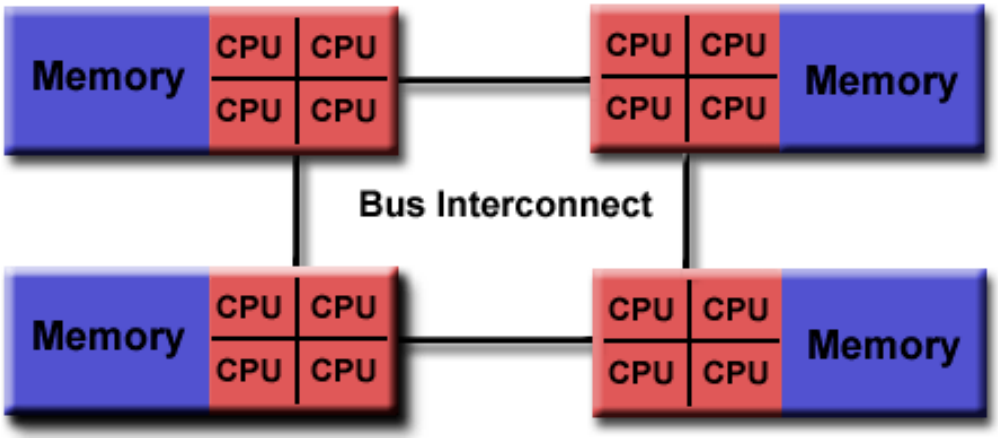
Scheduling

- Έλεγχος της κατανομής εργασίας (worksharing)
  - `schedule(static[, chunk])`:
    Compile-time κατανομή των επαναλήψεων σε threads
  - `schedule(dynamic[, chunk])`:
    Οι επαναλήψεις χωρίζονται σε κομμάτια (chunks) και κάθε thread που τελειώνει ένα chunk παίρνει το επόμενο διαθέσιμο
  - `schedule(guided[, chunk])`:
    Όπως το dynamic αλλά τα chunks ξεκινούν μεγαλύτερα και στη συνέχεια μικραίνουν ως το καθορισμένο όριο
  - `schedule(runtime)`:
    Ο αλγόριθμος κατανομής εργασίας θα ελέγχεται από το unix environment τη στιγμή που ξεκινάει το πρόγραμμα
    - `export OMP_SCHEDULE=STATIC`

Ιεραρχίες Μνήμης

- Shared memory: μοντέλο κοινής μνήμης
- Όλα τα threads μοιράζονται την κοινή μνήμη
- Πολλά είδη κοινής μνήμης
  - Uniform Memory Access
  - Non-Uniform Memory Access
  - Cache hierarchy: L1, L2, L3, shared-L3, ...





Μοντέλο Μνήμης

- Μοντέλο μνήμης:
  - Coherence: συνέπεια μεταξύ αντιγράφων της ίδιας διεύθυνσης σε διαφορετικά σημεία της ιεραρχίας μνήμης
  - Consistency: σειρά με την οποία γίνονται όντως οι εγγραφές στη μνήμη και σειρά με την οποία γίνονται αντιληπτές οι εγγραφές στη μνήμη από κάθε thread

Μοντέλο Μνήμης - Reordering

- Source code order: Η σειρά στη γλώσσα προγραμματισμού
- Code order: Η σειρά στη γλώσσα μηχανής
  - Compiler reordering: ο μεταγλωττιστής αλλάζει σειρά των accesses
- Commit order
  - Out-of-order execution, pipelining, κλπ εντός του κάθε επεξεργαστή
- Κάθε thread έχει μια όψη της μνήμης
- Μπορεί η ανά πάσα στιγμή μνήμη που βλέπει ένα thread να διαφέρει από την πραγματική μνήμη
- Consistency model: Κανόνες για τη σειρά των read/write/synchronization εντολών
- Διαφορετικό για κάθε γλώσσα (αν υπάρχει)
- Διαφορετικό για κάθε ISA
- Διαφορετικό για κάθε υλοποίηση επεξεργαστή

Είδη Consistency

- Sequential Consistency:
  - Οι αναγνώσεις (R), εγγραφές (W), και συγχρονισμοί (S) στη μνήμη είναι sequentially consistent όταν
    - Μένουν στη σειρά που είναι δηλωμένα στο πρόγραμμα (program order) για κάθε επεξεργαστή
    - Φαίνονται από κάθε άλλο επεξεργαστή να γίνονται με αυτή τη σειρά
- Program order = code order = commit order
- Relaxed Consistency:
  - Διάφορα είδη
  - Προκύπτουν αφαιρώντας περιορισμούς στη σειρά των R, W, S από το sequential
  - Μπορεί να είναι ισοδύναμα με το sequential *υπό συνθήκες*, όπως π.χ., race freedom

Consistency μνήμης στο OpenMP

- Weak consistency
  - Οι πράξεις συγχρονισμού (S) δεν μπορούν να αλλάξουν σειρά με πράξεις R ή W
    - Εγγυάται
    - $S \rightarrow W, S \rightarrow R, R \rightarrow S, W \rightarrow S, S \rightarrow S$
- Η λειτουργία S για το OpenMP είναι το flush
  - low-level
  - Εννοείται σε κάθε άλλη λειτουργία συγχρονισμού

Flush

- Ορίζει ένα σημείο στη σειρά εκτέλεσης του κάθε thread
  - Εγγυάται ότι το thread θα δεί μια συνεπή (consistent) όψη της μνήμης για τις μεταβλητές που ανήκουν στο “flush set”
  - Το “flush set” είναι:
    - Οι μεταβλητές που δίνονται στο `flush(list)`
    - Όλες οι μεταβλητές που είναι ορατές από το thread, όταν δεν ορίζεται λίστα μεταβλητών
- Η λειτουργία flush εγγυάται ότι:
  - Όλες οι εντολές ανάγνωσης και εγγραφής σε θέσεις μνήμης του flush set που υπάρχουν πριν το flush, θα ολοκληρωθούν πριν το flush
  - Καμία εντολή ανάγνωσης και εγγραφής σε θέσεις μνήμης του flush set που υπάρχει μετά το flush, δεν θα ξεκινήσει πριν τελειώσει το flush
  - Εντολές flush που έχουν κοινές θέσεις μνήμης δεν μπορούν να αλλάξουν σειρά
- Αντίστοιχο με fence σε άλλα μοντέλα κοινής μνήμης

Συγχρονισμός με flush

- Σε κάθε δομή συγχρονισμού εννοείται ένα flush
  - Είσοδος και έξοδος parallel region
  - Κάθε είδους barrier
  - Είσοδος και έξοδος critical region
  - Κάθε φορά που κλειδώνει ή ξεκλειδώνει ένα lock
- ΔΕΝ υπάρχει flush στις περιπτώσεις:
  - Είσοδος και έξοδος σε master region
  - Είσοδος και έξοδος σε worksharing region

flush και reorder

- Ο compiler μπορεί να αλλάξει τη σειρά εντολών
  - Χρήσιμο σε πολλά optimizations
  - Καλύτερη χρήση όλου του hardware (floating point, ALU, κλπ)
  - Κρύβει latencies
- Το flush απαγορεύει να αλλάξει η σειρά
  - Οποιαδήποτε εντολή δεν αντιμετατίθεται με γενικό flush
  - Μπορεί να αλλάξει σειρά με flush(list) αν δεν είναι στο flush set
- Πολύπλοκο για τον προγραμματιστή να σκέφτεται σε flush
- Η flush δεν συγχρονίζει διαφορετικά threads, αλλά ένα thread με τη μνήμη
  - Συγχρονισμός μέσω μνήμης

Debugging

- Debugger
- Compiler
  - Συχνό είδους λάθος: unintended sharing
  - default(none): παράγει compile-time λάθη για αναφορά σε θέσεις μνήμης που δεν έχουν δηλωθεί

```
#pragma omp parallel for default(none) private(c, eps)
{
  for (i=0; i<NPOINTS; i++) {
    for (j=0; j<NPOINTS; j++) {
      c.r = -2.0+2.5*(double)(i)/(double)(NPOINTS) + eps;
      c.i = 1.125*(double)(j)/(double)(NPOINTS) + eps;
      testpoint();
    }
  }
}
```

- Η δομή parallel for εννοεί το iteration variable ως private
- Αλλά όχι όλων των for loops
- Unspecified j!

Δομές δεδομένων

- Παράλληλη επεξεργασία δομών
  - Η παράλληλη επανάληψη χρειάζεται καλώς ορισμένα όρια
  - Δεν υπάρχει while δομή
- Πώς παραλληλοποιείται μια λίστα;

```
p = head;
while (p) {
  process(p);
  p = p -> next;
}
```

OpenMP και δομές

- Αρχικά φτιάχτηκε για supercomputing εφαρμογές
  - Fortran
  - Πίνακες και regular loops
- Αναδρομή, δείκτες, αναδρομικές δομές
  - Σπάνια στη Fortran για υπολογισμούς Φυσικής, Άλγεβρας, κλπ
- Δύσκολη η προσαρμογή
  - Με το αρχικό OpenMP η διάσχιση λίστας είναι πολύπλοκη
  - Μετατροπή σε doall
- Μετατροπή σε πίνακα
  - Απαρίθμηση στοιχείων
  - Πίνακας δεικτών
  - doall σε πίνακα

Λίστα σε OpenMP

```
/* count elements */
  while (p != NULL) {
    p = p->next;
    count++;
  }

/* populate pointer array */
  p = head;
  for(i=0; i<count; i++) {
    parr[i] = p;
    p = p->next;
  }

/* actual parallelism */
#pragma omp parallel
  {
#pragma omp for schedule(static,1)
    for(i=0; i<count; i++)
      processwork(parr[i]);
  }
```

OpenMP και δομές – Συμπέρασμα

- Γίνεται η παραλληλοποίηση
  - Με άσχημο τρόπο
  - Με κόστος πολλές διασχίσεις των δεδομένων
- Χρειάζεται υποστήριξη για γενικό προγραμματισμό
  - Όχι μόνο for, bounded
  - Language-based
- OpenMP 3.0 Tasks

OpenMP Tasks

- Ανεξάρτητα κομμάτια δουλειάς
  - Κώδικας
  - Δεδομένα
  - Control variables
- Τα tasks εκτελούνται από threads
- Το runtime system του OpenMP αποφασίζει την αντιστοίχιση
  - Μπορεί να εκτελεστούν άμεσα (inline)
  - Μπορεί να καθυστερήσει η εκτέλεση (defer)

Ορισμοί

- `omp task` directive (static)
  - Με code block όπως σε parallel region ή worksharing directive
- Task (dynamic)
  - Η δομή δεδομένων που δημιουργείται και περιγράφει τον υπολογισμό
  - Δημιουργείται την ώρα της εκτέλεσης
  - Όταν ένα thread φτάσει σε `omp task` directive
- Task region (dynamic)
  - Το trace από εντολές που παράγεται όταν ένα thread εκτελεί ένα task

Εκτέλεση tasks

- Το OpenMP εγγυάται την εκτέλεση των tasks που υπάρχουν σε εκκρεμότητα στα barriers
  - `#pragma omp barrier`
  - `#pragma omp taskwait`

```
/* create threads */
#pragma omp parallel
{
#pragma omp task
    foo();
    /* one foo task per thread */
#pragma omp barrier
    /* now all tasks have completed */
#pragma omp single
  {
#pragma omp task
      bar();
      /* one task created */
  }
  /* implicit barrier will wait for task bar() here */
}
```

Παράδειγμα

- Σειρά Fibonacci
- Divide and conquer

```
int fib(int n) {
  int x, y;
  if(n < 2) return n;
#pragma omp task
    x = fib(n-1);
#pragma omp task
    y = fib(n-2);
#pragma omp taskwait
  return x + y;
}
```

- Πρόβλημα: private task variables
- Τα x και y που γράφονται στα task δεν είναι ίδια με τις τοπικές μεταβλητές x και y!

Παράδειγμα (2)

- Σειρά Fibonacci
- Divide and conquer

```
int fib(int n) {
  int x, y;
  if(n < 2) return n;
#pragma omp task shared(x)
    x = fib(n-1);
#pragma omp task shared(y)
    y = fib(n-2);
#pragma omp taskwait
  return x + y;
}
```

- Πρόβλημα: private task variables
- Τα x και y που γράφονται στα task δεν είναι ίδια με τις τοπικές μεταβλητές x και y!
- Λύση: shared

Παράδειγμα – List

```
List my_list;
Element *e;
#pragma omp parallel
#pragma omp single
{
  for(e = my_list->first; e; e = e->next) {
#pragma omp task
      process(e);
  }
}
```

- Προσοχή: Data race!
- Η μεταβλητή e μπορεί να έχει ταυτόχρονα updates

Παράδειγμα – List

```
List my_list;
Element *e;
#pragma omp parallel
#pragma omp single
{
  for(e = my_list->first; e; e = e->next) {
#pragma omp task firstprivate(e)
      process(e);
  }
}
```

- Λύση: `firstprivate`
- Update της τοπικής μεταβλητής, αντίγραφο ανά task, in sequence

Παράδειγμα – List – Εκτέλεση

```
List my_list;
Element *e;
#pragma omp parallel
#pragma omp single
{
  for(e = my_list->first; e; e = e->next) {
#pragma omp task firstprivate(e)
      process(e);
  }
}
```

- Δημιουργούνται όλα τα threads στο parallel region
- Ένα thread μπαίνει στο single, τα άλλα περιμένουν στο implicit barrier στο τέλος
- Το single thread δημιουργεί tasks
- Όσα threads περιμένουν σε barrier εκτελούν διαθέσιμα tasks όσο περιμένουν
- Το barrier τελειώνει όταν φτάσουν όλοι

Memory Access Patterns

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    sum += A[i][j];
```

```
for (int j = 0; j < n; j++)
  for (int i = 0; i < n; i++)
    sum += A[i][j];
```

- Γρηγορότερη η πρόσβαση σε διαδοχικές διευθύνσεις μνήμης
- Δεδομένα στην cache
- Prefetcher pattern
- Locality

Good Practice

- Μοίρασμα των δεδομένων
  - Δεδομένα ανά thread (embarassingly parallel)
  - Δεδομένα στις μνήμες (locality)
- Αποφύγετε ακριβά global synchronization operations
  - Αποφυγή `ordered`
  - Αποφυγή `critical` (καλύτερα `atomic`)
  - Αποφυγή μεγάλων `critical` regions
- Μείωση των implicit barriers
  - `nowait` όπου δεν χρειάζεται barrier
- Όσο μεγαλύτερο parallel region γίνεται
  - Κόστος δημιουργίας thread
  - Λιγότερο σειριακό κώδικα
  - Compiler optimizations
  - Όχι parallel regions μέσα σε loops
- Load balancing
  - Άνισοι υπολογισμοί προκαλούν καθυστέρηση
  - Κάποια threads περιμένουν
- Avoid False sharing
  - Δεδομένα αρκετά κοντά στην cache

OpenMP

- http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf