Ιόνιο Πανεπιστήμιο – Τμήμα Πληροφορικής
Παράλληλος Προγραμματισμός
2023-24

Το πρότυπο παράλληλης επεξεργασίας OpenMP

(Εισαγωγή)

https://mixstef.github.io/courses/parprog/

Μ.Στεφανιδάκης

Το πρότυπο OpenMP

- Καθορίζει ένα API παράλληλης επεξεργασίας
  - Στην αρχική μορφή του ήταν προσανατολισμένο στη διαχείριση threads με το μοντέλο fork – join
  - Αργότερα προστέθηκαν δυνατότητες χρήσης tasks, εντολών SIMD και προγραμματισμού σε GPU
- Επιτρέπει τον παράλληλο προγραμματισμό σε υψηλότερο επίπεδο
  - Απ’ ό,τι τα POSIX threads
- Υλοποιείται από τον μεταγλωττιστή
  - Για τις γλώσσες C, C++ και Fortran (HPC programming)
  - Υποστήριξη από όλους τους «εμπορικούς» μεταγλωττιστές

Προγραμματισμός με το OpenMP

- Χρήση C pragmas
  - Οδηγίες (directives) για τον μεταγλωττιστή πέρα από τη γλώσσα C
  - Συχνά χρησιμοποιούνται για την υλοποίηση extensions του μεταγλωττιστή
  - Ο κάθε μεταγλωττιστής αποφασίσει πώς θα τις χειριστεί
    - Στην περίπτωση του OpenMP τροποποιούν τον κώδικα που γράφουμε

```
#pragma omp parallel
```

- Συμπληρωματικές συναρτήσεις (βιβλιοθήκη) για διαχείριση του OpenMP
  - Λήψη πληροφορίας για το περιβάλλον εκτέλεσης ή εκτέλεση ρυθμίσεων

Προγραμματισμός με το OpenMP

- Include headers
  - `#include <omp.h>`
  - Ορισμοί του OpenMP API
- Compiler flags
  - Π.χ. `gcc -O2 -Wall -fopenmp hello-omp.c -o hello-omp`
  - Ειδοποιεί τον μεταγλωττιστή για τα pragmas του OpenMP που θα συναντήσει και για να συνδέσει το εκτελέσιμο πρόγραμμά μας με αναφορές στην κοινή βιβλιοθήκη του συστήματος που υλοποιεί τις λειτουργίες του OpenMP
    - Αν δεν προστεθεί θα αγνοηθούν τα #pragma omp ή/και θα δημιουργηθεί σφάλμα σύνδεσης με τη βιβλιοθήκη

Παραλληλισμός με threads

- Δημιουργία μιας ομάδας threads που μπορούν να εκτελούνται παράλληλα
  - Για την εκτέλεση ενός (του ίδιου!) τμήματος κώδικα που ονομάζεται παράλληλη περιοχή (parallel region)
  - Το thread που δημιουργεί την παράλληλη περιοχή (master ή primary thread) συνεχίζει και αυτό με την εκτέλεση της παράλληλης περιοχής
  - Στο τέλος της παράλληλης περιοχής εισάγεται ένα έμμεσο (implicit) barrier συγχρονισμού των threads
- Αντιστοιχία με το μοντέλο fork – join που ήδη γνωρίζουμε
  - Προσοχή: η παράλληλη περιοχή δεν υλοποιεί καταμερισμό εργασίας (work sharing)

Παράλληλη περιοχή threads

- #pragma omp parallel

```
#pragma omp parallel
{
     printf("Hello world!\n");

} // NOTE: implicit barrier sync here
```

- Το printf() θα εκτελεστεί από έναν συγκεκριμένο αριθμό threads
  - Εδώ ο αριθμός δεν προσδιορίζεται οπότε το OpenMP θα διαλέξει έναν «κατάλληλο» αριθμό (π.χ. τον αριθμό των hardware threads του συστήματος εκτέλεσης)

Τι συμβαίνει;

```
.LC0:
        .string  "Hello world!"
main._omp_fn.0:
        .cfi_startproc
        endbr64
        lea      rdi, .LC0[rip]
        jmp      puts@PLT
        .cfi_endproc
main:
        .cfi_startproc
        endbr64
        sub      rsp, 8
        .cfi_def_cfa_offset 16
        xor      ecx, ecx
        xor      edx, edx
        xor      esi, esi
        lea      rdi, main._omp_fn.0[rip]
        call     GOMP_parallel@PLT
        xor      eax, eax
        add      rsp, 8
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
```

η παράλληλη περιοχή έγινε η συνάρτηση `main._omp_fn.0`

η διεύθυνση της `main._omp_fn.0` *περνάει ως παράμετρος στη συνάρτηση* `GOMP_parallel` *του OpenMP*

Αναγνωριστικός αριθμός thread

- Κάθε thread έχει έναν μοναδικό αριθμό
  - Λήψη με τη συνάρτηση omp_get_thread_num()
  - Χρησιμοποιείται για να διαφοροποιηθεί η δουλειά κάθε thread μέσα στην ίδια παράλληλη περιοχή
    - Ο τρόπος αυτός εργασίας ακολουθεί το μοντέλο “single program multiple data” (SPMD)
- Πόσα threads εκτελούν την παράλληλη περιοχή;
  - Λήψη με τη συνάρτηση omp_get_num_threads()

Ελέγχοντας τον αριθμό των threads

- Ο αριθμός των threads σε μια παράλληλη περιοχή μπορεί να προσδιοριστεί με διάφορους τρόπους
  - Μεταβλητή περιβάλλοντος OMP_NUM_THREADS
    - Θα χρησιμοποιηθεί αν δεν ορίζεται αλλιώς μέσα στον κώδικα
  - Συνάρτηση omp_set_num_threads()
    - Από το σημείο κλήσης και μετά
  - Ως πρόσθετο στοιχείο (clause) στο #pragma omp parallel
    - Όσο διαρκεί η κρίσιμη περιοχή

```
#pragma omp parallel num_threads(16)
```

Εισαγωγή στην εμβέλεια μεταβλητών

- Data Scope
  - Πολύ σημαντική έννοια στον προγραμματισμό με το OpenMP
  - Τι συμβαίνει με τις μεταβλητές του σειριακού προγράμματος; τι βλέπει κάθε thread;
- Δύο βασικές κατηγορίες μεταβλητών
  - Private: κάθε thread βλέπει διαφορετική «κόπια» της μεταβλητής
  - Shared: όλα τα threads βλέπουν την ίδια μεταβλητή
    - Πρέπει να υπάρχει έλεγχος πρόσβασης
- Όταν δεν δηλώνεται ρητά το είδος μιας μεταβλητής, το OpenMP έχει default κανόνες για αυτό

Παράλληλη περιοχή και εμβέλεια μεταβλητών

- Μέσα στην παράλληλη περιοχή
  - Οποιαδήποτε μεταβλητή δηλώνεται μέσα στην παράλληλη περιοχή θεωρείται private
    - Το ίδιο ισχύει για μεταβλητές συναρτήσεων που καλούνται μέσα από την παράλληλη περιοχη
  - Global μεταβλητές είναι shared
  - Μεταβλητές που δεν έχει προσδιοριστεί το είδος τους χρησιμοποιούν το default (εξ' ορισμού shared)
- Ρητός προσδιορισμός είδους εμβέλειας
  - Το #pragma omp parallel (όπως και άλλα pragmas) παίρνουν προαιρετικά clauses προσδιορισμού εμβέλειας για λίστες μεταβλητών
    - private() και shared()

```
#pragma omp parallel private(id)
```

Διαμοιρασμός εργασίας ενός loop

- Loop worksharing construct
  - Η εργασία ενός loop διαμοιράζεται στα threads μιας παράλληλης περιοχής
    - Δεν παρέχει παραλληλισμό threads, πρέπει να έχει προηγηθεί ένα #pragma omp parallel
  - Εφαρμόζεται σε ένα for loop της C
- Περιορισμοί στο είδος του for loop
  - Η μεταβλητή επανάληψης θα πρέπει να είναι ακέραια
  - Θα πρέπει να είναι γνωστά και σταθερά η αρχή και το τέλος της επανάληψης
  - Η σύγκριση θα πρέπει να είναι <, <=, >, >= με σταθερή τιμή
  - Η αύξηση/μείωση της μεταβλητής επανάληψης θα πρέπει να γίνεται κατά ένα σταθερό βήμα

Διαμοιρασμός εργασίας ενός loop

- #pragma omp for

```
#pragma omp parallel
{

  #pragma omp for
  for (int i=0;i<N;i++) {

    printf("Thread %d working on element %d\n",
            omp_get_thread_num(),i);

  } // implicit barrier here - use nowait clause to avoid


} // implicit barrier here
```

- Στο τέλος του #pragma omp for υπάρχει ένα έμμεσο barrier – αν δεν το θέλουμε χρησιμοποιούμε το προαιρετικό clause nowait

Διαμοιρασμός εργασίας ενός loop

- #pragma omp parallel for

```
#pragma omp parallel for
for (int i=0;i<N;i++) {

   printf("Thread %d working on element %d\n",
             omp_get_thread_num(),i);

}
```

- Συνήθης συνδυασμός του #pragma omp parallel με το #pragma omp for
  - Μόνο αν θέλουμε να κάνουμε και κάτι άλλο εκτός του for μέσα στην παράλληλη περιοχή χρησιμοποιούμε ξεχωριστά pragmas
- Η μεταβλητή επανάληψης του loop είναι πάντα private

Μέθοδοι διαμοιρασμού

- Static
  - Η default μέθοδος, χωρίζει σε ισόποσα μέρη την εργασία
    - Κατάλληλη όταν κάθε επανάληψη έχει τον ίδιο φόρτο
- Dynamic
  - Ανάθεση μιας επανάληψης (ή ενός chunk επαναλήψεων) ανά thread και μόλις ολοκληρωθεί, νέα ανάθεση
    - Κατάλληλο όταν το φορτίο διαφέρει σημαντικά ανά επανάληψη
- Guided
  - Όπως το dynamic, η ανάθεση όμως γίνεται σε chunks επαναλήψεων που υπολογίζονται δυναμικά διαιρώντας τις επαναλήψεις που απομένουν δια του αριθμού των threads
    - Μεταξύ του static και του dynamic

```
#pragma omp parallel for schedule(guided)
```

Αρχική τιμή private μεταβλητών

- Σε μια παράλληλη περιοχή οι μεταβλητές μπορούν να είναι shared ή private
  - Οι shared μεταβλητές έχουν πάντα την ίδια και μοναδική θέση
  - Οι private μεταβλητές έχουν μια «κόπια» ανά thread της παράλλλης περιοχής
- Αρχικοποίηση private μεταβλητών
  - Δεν γίνεται
    - Συνήθως έχουν τιμή 0 αν δεν τις αρχικοποιήσουμε εμείς (εξαρτάται από το λειτουργικό σύστημα)
  - Ακόμα και αν έχει τιμή έξω από την παράλληλη περιοχή
    - Η υπάρχουσα τιμή δεν αντιγράφεται στις «κόπιες»

Firstprivate

- Αντιγραφή της υπάρχουσας τιμής έξω από την παράλληλη περιοχή σε κάθε κόπια της μεταβλητής

```
int main() {

  int x = 22;

  #pragma omp parallel private(x)
     {
    printf("Thread %d: my private x=%d\n",omp_get_thread_num(),x);
  }

  #pragma omp parallel firstprivate(x)
     {
    printf("Thread %d: my firstprivate x=%d\n",omp_get_thread_num(),x);
  }

  return 0;
}
```

θα τυπώσει (μάλλον) 0: η κόπια της x δεν είναι αρχικοποιημένη

θα τυπώσει 22: η αρχική τιμή αντιγράφεται στην κόπια της x

Συγχρονισμός threads

- Το OpenMP παρέχει διάφορα εργαλεία συγχρονισμού των threads
- Με τη βοήθεια των pragmas
  - critical construct: υλοποίηση κρίσιμων περιοχών για αμοιβαίο αποκλεισμό κατά την ενημέρωση κοινών πόρων

```
/ all threads - update shared variable
#pragma omp critical
{
  count++;
}
```

    - Υπάρχει και το atomic construct για γρήγορες ενημερώσεις με εγγυημένη ατομικότητα εκτέλεσης
  - barrier construct: συγχρονισμός όλων των threads μιας παράλληλης περιοχής

```
#pragma omp barrier
```

master vs single construct

- Το master construct
  - Απλό εργαλείο για την εκτέλεση ενός κομματιού κώδικα από το master thread
  - Δεν παρέχει κανέναν συγχρονισμό με τα άλλα threads
    - Όποιο thread δεν είναι master παρακάμπτει το κομμάτι κώδικα αυτό
- Το single construct
  - Είναι ένα worksharing construct όπως το for
  - Αναθέτει την εκτέλεση του κομματιού κώδικα σε ένα οποιοδήποτε thread
  - Στο τέλος προστίθεται ένα έμμεσο barrier – όλα τα υπόλοιπα threads θα περιμένουν να ολοκληρωθεί η εργασία στο single
    - Δυνατότητα αφαίρεσης έμμεσου barrier με το nowait

OpenMP reduction

- Λειτουργίες reduction μπορούν να εκτελεστούν με όσα ξέρουμε μέχρι τώρα
  - parallel for για τον κατανεμημένο υπολογισμό επιμέρους αποτελεσμάτων
    - Θα χρειαστούμε private ή firstprivate μεταβλητές
  - critical για ενημέρωση του συνολικού αποτελέσματος από τα επιμέρους αποτελέσματα
- Το OpenMP προσφέρει όμως μια πολύ πιο «αυτοματοποιημένη» λύση
  - Σε ορισμένα constructs (όπως το parallel ή το for) μπορεί να χρησιμοποιηθεί το clause reduction(τελεστής:λίστα μεταβλητών)

reduction clause

- reduction(τελεστής:λίστα μεταβλητών)
  - Ο τελεστής είναι ένας από τους +, -, *, &, |, ^, &&, ||, min, max
    - Μπορούν επίσης να οριστούν custom τελεστές με το #pragma omp declare reduction
  - Για τις μεταβλητές που ορίζονται στη λίστα θα δημιουργηθούν «κόπιες» ανά thread
  - Οι «κόπιες» θα αρχικοποιηθούν στην κατάλληλη τιμή ανάλογα με τον τελεστή
    - Π.χ. στο 0 για το + και στο 1 για το *
  - Στο τέλος της εκτέλεσης του construct που ορίστηκε το reduction οι μεταβλητές συνδυάζονται σε τελικό αποτέλεσμα
    - Συνδυασμός ανάλογα με τελεστή