Intel Threading Building blocks (TBB)

rebranded “oneAPI TBB” (oneTBB)

Εισαγωγή και λειτουργίες “map”, “reduce”

Intel Threading Building Blocks (TBB)

- Βιβλιοθήκη παραλληλισμού με threads, χρησιμοποιεί τον μηχανισμό των templates της C++
- Χρησιμοποιεί tasks που εκτελούνται από ομάδες threads (task arenas)
- Παρέχει interfaces σε διάφορα επίπεδα
  - Default επιλογές και αρχικοποίηση για τις συνήθεις χρήσεις, μεγαλύτερος έλεγχος για τις ειδικές περιπτώσεις
  - Parallel algorithms, containers, control flow graphs
  - Tasks, Memory allocation & Mutual exclusion

C++ standards

- TBB: χρήση «νέων» χαρακτηριστικών από το πρότυπο C++11
  - Όχι και τόσο νέα πια..
    - C++23 (αναμένεται)
    - C++20 (πειραματική υποστήριξη από GCC)
    - C++17 (default από GCC 11)
    - C++14 (default από GCC 6.1)
    - C++11 (πρώην C++0x, πλήρης υποστήριξη από GCC 4.8.1)

GCC και TBB

- Command line για μεταγλώττιση όταν το TBB είναι εγκατεστημένο κεντρικά

```
g++ -Wall -O2 -std=c++11 file.cpp -o file -ltbb
```

- Στους υπολογιστές του εργαστηρίου χρησιμοποιούμε τη βιβλιοθήκη TBB του συστήματος
  - Υπάρχει στις περισσότερες διανομές linux
    - π.χ. libtbb2 και libtbb-dev packages
    - Όχι πάντα η πιο ενημερωμένη έκδοση

Εγκατάσταση oneTBB από site Intel

- Για εγκατάσταση δείτε εδώ:

  https://www.intel.com/content/www/us/en/developer/articles/tool/oneapi-standalone-components.html#onetbb

  - Μπορείτε να το εγκαταστήσετε ως απλός χρήστης (δεν πειράζετε πιθανή υπάρχουσα εγκατάσταση συστήματος)

- Για την αναγνώριση include files και shared libraries κατά τη μεταγλώττιση και την εκτέλεση:

  https://www.intel.com/content/www/us/en/docs/onetbb/get-started-guide/2021-9/overview.html

Lambda expressions (C++11)

- Δυνατότητα δημιουργίας «ανώνυμων» συναρτήσεων
  - Που μπορούν να αποθηκευτούν και να χρησιμοποιηθούν αργότερα με ασύγχρονο τρόπο (δηλ. ως callbacks)
  - Ποιο είναι το πρόβλημα που πρέπει να λυθεί:
    - Οι τιμές των μεταβλητών κατά τη δημιουργία του lambda πρέπει να διατηρηθούν και στη χρήση του (που θα γίνει αργότερα…)
    - “Closure”, το αναλαμβάνει ο μεταγλωττιστής

Το πρόβλημα

```
int main() {
  int k = 77;
  auto lambda2 = [](int x) {
    return x+33+k;
  };
  cout << lambda2(5) << endl;
  return 0;
}
```

Τι τιμή θα έχει το k τη στιγμή της εκτέλεσης του lambda;

Για την ακρίβεια, ο πιο πάνω κώδικας δημιουργεί σφάλμα κατά τη μεταγλώττιση

“Capturing Closure”

```
int main() {
  int k = 77;
  auto lambda2 = [k](int x) {
    return x+33+k;
  };
  cout << lambda2(5) << endl;
  return 0;
}
```

Τη στιγμή της δημιουργίας του lambda η τιμή του k συγκρατείται (captured) σε ένα ξεχωριστό περιβάλλον (closure)

Οι μεταβλητές μπορούν να συγκρατηθούν by value ή by reference.
Μπορούν να συγκρατηθούν επιλεγμένες μεταβλητές ή όλες του περιβάλλοντος της δημιουργίας του lambda

Στο παρασκήνιο ο μεταγλωττιστής δημιουργεί ένα στιγμιότυπο κλάσης με τις αντίστοιχες μεταβλητές και τον τελεστή κλήσης () που περιέχει τον κώδικα του lambda

Απλοί μετασχηματισμοί τύπου map

```
for (i=0;i<N;i++) {
  a[i] = f(b[i],c[i],..)
}
```

- TBB → **tbb::parallel_for()**

tbb::parallel_for(first,last,func)

- Ισοδύναμο με το

  ```
  for(i=first;i<last;i+=1) func(i);
  ```

- Δεν πρέπει να υπάρχουν αλληλεξαρτήσεις μεταξύ των επαναλήψεων
- Προαιρετικά μπορεί να οριστεί και το βήμα (step) της αύξησης του i
- Τα first και last πρέπει να είναι του ίδιου τύπου
  - ints, chars, longs, pointers, size_t...
  - π.χ. αν N είναι τύπου size_t, χρησιμοποιούμε parallel_for(size_t(0),N,..)

tbb::parallel_for(first,last,func)

- Δεν υπάρχει εγγύηση αν/πώς/με ποια σειρά θα εκτελεστούν παράλληλα οι επαναλήψεις
  - Ο partitioner που χρησιμοποιείται καθορίζει πώς θα κατανεμηθούν οι επαναλήψεις στα threads
    - default: auto_partitioner, κατανομή σε τμήματα ανάλογα με τον αριθμό των threads, διαίρεση τμημάτων μόνο αν υπάρχουν idle threads (load balancing)
- Παράδειγμα χρήσης με lambda:

```
tbb::parallel_for(size_t(0),N,[&a](size_t i) {
   a[i] = map_func(a[i]);
 });;
```

tbb::parallel_for(range,body)

- Range: ένα μέρος των συνολικών επαναλήψεων
  - διαιρείται αναδρομικά σε 2 μέρη
  - έως ένα ελάχιστο μέγεθος
- Body: αντιστοιχεί στον κώδικα που θα εκτελεστεί σε κάθε τελικό range

blocked_range

- Ένα είδος Range: μια περιοχή επαναλήψεων [από,έως)
  - διαιρείται αναδρομικά το πολύ μέχρι ένα “grainsize” (default=1) – η τελική κατανομή εξαρτάται όμως και από τον partitioner που χρησιμοποιείται
    - auto_partitioner: διαιρεί μόνο εάν υπάρχουν idle threads

blocked_range

- Iterator interface, π.χ.

```
tbb::parallel_for(tbb::blocked_range<size_t>(0,N),[&a](const tbb::blocked_range<size_t>& r) {
    for (size_t i=r.begin();i!=r.end();++i) {
      a[i] = map_func(a[i]);
    }
  });
```

Απλοί μετασχηματισμοί τύπου reduce

```
double sum = 0.0;
for (size_t i=0;i<N;++i) {
  sum += a[i];
}
```

- TBB → **tbb::parallel_reduce()**

tbb::parallel_reduce(range, identity, func, reduction)

- Η εργασία στην περιοχή range διαιρείται σε υποπεριοχές και σε κάθε μία από αυτές εφαρμόζεται η επεξεργασία func
  - Η func συνδυάζει όλες τις τιμές μιας υποπεριοχής σε ένα αποτέλεσμα, ξεκινώντας από μια αρχική τιμή init
  - Αν π.χ. δουλεύουμε με blocked_range από doubles, η func θα είναι lambda με ορίσματα

    ```
    [&... ](const tbb::blocked_range<size_t>& r,double init) -> double { … }
    ```

tbb::parallel_reduce(range, identity, func, reduction)

- Μετά τη λήξη της επεξεργασίας, τα μερικά αποτελέσματα από τις υποπεριοχές συνδυάζονται μέσω της λειτουργίας reduction
  - Αν π.χ. δουλεύουμε με doubles, η reduction θα είναι lambda με ορίσματα

    [ ](double x,double y) -> double { … }
- identity είναι η αρχική τιμή για κάθε επεξεργασία
  - «ταυτότητα», π.χ. αν δουλεύουμε με ints:
    - το 0 για αθροίσματα
    - το 1 για τον πολλαπλασιασμό
    - το std::numeric_limits<int>::min() για την εύρεση του μέγιστου κ.ο.κ