UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA
KATEDRA ZA PRIMENJENE RAČUNARSKE NAUKE

Paralelno računarstvo

Računarske vežbe

Letnji semestar 2020/2021.
Studijski program: Informacioni inženjering

Šabloni

Šabloni

- C++ omogućava definisanje šablona (template) za funkcije, gde je tip podatka argument šablona
- Na osnovu šablona se mogu automatski generisati konkretne funkcije za konkretan tip podataka
- Šabloni mogu da se prave i za klase

Šabloni

- Formalni parametri šablona između zagrada<i> (mogu predstavljati tipove ili konstante)

```
//sablon funkcije:

template< typename T > T maximum

(T a, T b){returna>b?a:b;}
```

Množenje matrice i vektora

Statičko raspoređivanje niti

- Na ovim vežbama ćemo naučiti kako statički rasporediti niti u for petlji u slučaju kada je broj zadataka značajno veći od broja niti tj. broja dostupnih CPU jezgara.
- Kasnije u toku kursa, kada se bude radio OpenMP, videćemo koliko je ovo jednostavnije implementirati korišćenjem direktiva.

Zadatak – sekvencialno množenje

- Neka je *A* matrica oblika dimenzija *m x n*, a *x* vektor dužine *n*. Potrebno je izračunati vektor *b* koji je proizvod matrice *A* i vektora *x* tj. $b := A \cdot x$

$$b_i = \sum_{j=0}^{n-1} A_{ij} \cdot x_j \quad \text{for all} \quad i \in \{0, \ldots, m-1\}$$

Zadatak – sekvencijalno množenje

- Pri množenju vektor posmatramo kao matricu sa jednom kolonom

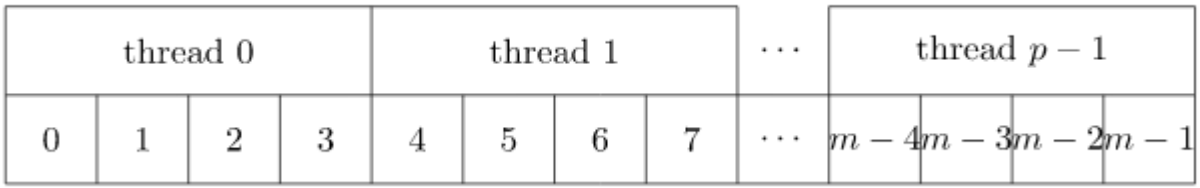
$$\mathbf{Ax} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix}.$$

Paralelizacija problema

- Ukoliko želimo da paralelizujemo ovaj kod najbolje je da to uradimo kroz paralelizaciju skalarnog proizvoda $b_i = \langle a_i | x \rangle$
- Međutim, u našem slučaju, broj redova tj. skalanih proizvoda je $m = 2^{14} = 16\,384$. Naivno gledano to znači da bi trebalo pokrenuti 16 384 niti, što bi dovelo do užasnih performansi usled prekljućivanja velikog broja niti.
- Velika količina niti bi forsirala procesor da secka izvršavanje tako što bi se svaka nit izvršavala kratak vremenski period, pa ustupila resurse drugoj niti.
- Da li je ovo dobar pristup? Koliko niti je optimalno pokrenuti?

Blokovski raspored niti

- Dobar pristup paralelizaciji našeg množenja je pokretanje p niti, gde je p broj procesorskih jezgara.
- Svaka nit u tom slučaju treba da obradi m/p redova (m je ukupan broj redova).



Lambda funkcije

- Često je potrebno napraviti funkciju koja je prosta ili se koristi samo jednom. Nepotrebno i zamarajuće je definisanje te funkcije u nekom delu koda ukoliko se ona posle neće koristiti.
- Elegantno kreiranje lambda funkcija je uvedeno od C++11 stadarda.

```
#include <iostream>

using namespace std;

int main()
{
    auto func = [] () { cout << "Hello world"; };
    func(); // now call the function
}
```

Lambda funkcije

- Pored elegantne sintakse, lambda funkcije nude mogućnost korišćenja promenljivih koje nisu definisane u njima tj. hvatanje promenljivih (*engl. Variable capture*).
- Ukoliko na početku funkcije stavimo [&], signaliziramo kompajleru da uhvati sve promenljive. Prazne zagrade [] signaliziraju da ne želimo da uhvatimo promenljive.
- Za više informacija pročitati : https://www.cprogramming.com/c++11/c++11-lambda-closures.html

Lambda funkcije

- Povratna vrednost lambda funkcije se navodi na kraju i opciona je

  [] () -> int { return 1; }

- Kako je C++ veoma obazriv na performanse postoji dosta mogućnosti hvatanja promenljivih:
  - [] Ne hvataju se promenljive
  - [&] Uhvati svaku varijablu i prenesi u funkciju po referenci
  - [=] Uhvati promenljive i prenesi po vrednosti
  - [=, &foo] Uhvati promenljive i prenesi po vrednosti, ali uhvaćenu *foo* promenljivu prenesi po referenci.
  - [bar] Uhvati *bar* i prenesi po vrednosti, ostale promenljive nemoj kopirati

Zadatak – blokovski raspored niti

- Napraviti novu funkciju koja paralelizuje sekvencijalni kod iz prethodnog zadatka. Pri implementaciji koristi blokovski raspored niti i lambda funkcije.

Ciklični raspored niti

- Problem množenja matrice i vektora je moguće rešiti i upotrebom statičkog cikličnog raspoređivanja niti.
- Pri cikličnom raspoređivanju *c* zadataka je raspoređeno na p niti sa korakom (razmakom) od *p* zadataka
- Nit nula bi obradila zadatak *0* pa zadatak *p*, *2p* itd.

threads	0	1	2	$\cdots$	$p-1$	0	1	2	$\cdots$	$p-1$	$\cdots$
tasks	0	1	2	$\cdots$	$p-1$	p	$p+1$	$p+2$	$\cdots$	$2p-1$	$\cdots$

Zadatak – ciklični raspored niti

- Napraviti novu funkciju koja paralelizuje sekvencijalni kod iz prethodnog zadatka. Implementirati ciklično raspoređivanje niti uz korišćenje lambda funkcija.

Lažno deljenje (*engl. False sharing*)

- Statički cikličan raspored niti je lakši za implementaciju od blokovskog rasporeda i približno istom brzinom rešavamo problem množenja u oba slučaja.
- Ukoliko napišemo kod cikličnog raspoređivanja kao na listingu ispod, da li će doći do smanjena performansi?

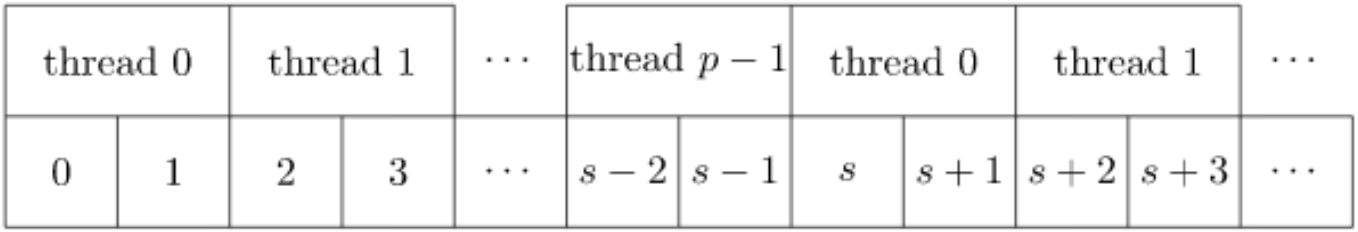
```
auto cyclic = [&] (const index_t& id) -> void {
   for (index_t row = id; row < n; row += num_threads) {
      // initialize result vector to zero
      b[row ] = 0;
      // directly accumulate in b[row]
         for (index_t col = 0; col < n; col++)
            b[row] += A[row*n+col]*x[col];
   }
};
```

Lažno deljenje

- Prethodni kod ne koristi posebnu promenljivu za agregiranje b[i], već akumulaciju vrši direktno u memorijskoj lokaciji vektora.
- Kako postoji više niti koje vrše izmene nad vektorom *b*, različite niti menjaju isti deo memorije koji svaka posebno kešira.
- Ukoliko nit 1 isčita liniju keša i promeni je, nit 2 pri promeni iste linije keša mora da sinhronizuje svoju kopiju sa promenom koju je izvršila nit 1.
- Ova pojava se naziva lažno deljenje i umanjuje efikasnost keširanja čime usporava izvršavanje algoritma.

Blok-cikličan raspored niti

- U ovom pristupu dodeljujemo fiksnu količinu *c* zadataka svakoj od *p* niti
- Na ovaj način na početku započnemo paralelno izvršavanje $s=p \cdot c$ zadataka
- Ukoliko na ovaj način ne obradimo sve zadatke (m>s), prosto ponovimo isti proces sve dok ne obradimo svih *m* zadataka



Zadatak – blok-ciklični raspored niti

- Napraviti novu funkciju koja paralelizuje sekvencijalni kod iz prethodnog zadatka. Pri implementaciji koristi blok-ciklični raspored niti i lambda funkcije.