

# Paralelni programski modeli

- ❖ pthreads
- ❖ MPI/OpenMP
- ❖ Cilk
- ❖ IPP, TBB

# Cilk jezik: <http://cilk.mit.edu/>

- ◆ Ključne reči za zadatke
  - cilk\_spawn
  - cilk\_sync
  - cilk\_for
- ◆ Hiperobjekti (Hyperobjects)
  - Reduktori (reducers)
- ◆ Naznake za nizove
- ◆ Osnovne funkcije
- ◆ SIMD pragma direktiva

# Uvod (1/3)

- ◆ Intel Cilk Plus je proširenje jezika C i C++
- ◆ Podržava ga raspoređivač zadataka, koji
  - nije direktno izložen aplikacionom programeru
- ◆ Programeru su vidljivi
  - ◆ Tri ključne reči (spawn, sync i for)
  - ◆ Hiper promenljive – lokalni pogled na globalne promenljive
  - ◆ Naznake za nizove
  - ◆ Osnovne funkcije
  - ◆ Pragma SIMD
- ◆ CPU treba da ima više jezgara i vektorskih jedinica

# Uvod (2/3)

- ◆ Različito iskorišćenje komponenata CPU
  - `clk_spawn` – koristi samo paralelizam jezgara
  - `#pragma simd` – koristi samo vektorske jedinice
  - Neke osnovne funkcije koriste oba
- ◆ Serijalizacija = ponašanje Cilk programa
  - Isto kao ponašanje sličnih C/C++ programa
- ◆ Izvršenje
  - C/C++ programa = linearan niz iskaza
  - Cilk programa = usmeren acikličan graf

# Uvod (3/3)

- ◆ Paralelno upravljanje izvršenjem može dovesti do trke do podataka (data race)
  - Delovi programa pristupaju podacima u nepoznatom redosledu
    - ◆ Bar jedan od pristupa je radi upisa (write access)
- ◆ Dodatno, izuzeća mogu dovesti do izvršenja delova koda, koji se ne bi izvršili u serijskom izvršenju

# Sintaksa

*jump-statement:*

***\_Cilk\_sync ;***

*postfix-expression:*

***\_Cilk\_spawn opt postfix-expression ( expression-list opt )***

*cilk-for-init-decl:*

***decl-specifier-seq init-declarator***

*grainsize-pragma:*

***# pragma cilk grainsize = expression***

*iteration-statement:*

***\_Cilk\_for ( cilk-for-init-decl ; condition ; expression )  
statement***

***\_Cilk\_for ( assignment-expression ; condition ; expression )  
statement***

# Model izvršenja zadatka

## ◆ Linija izvršenja (strand)

- Niz instrukcija bez *spawn* i *sync*

## ◆ Tačka mrešćenja (*spawn*)

- Jedna linija se pretvara u dve

## ◆ Tačka sinhronizacije

- Jedna ili dve linije se pretvaraju u jednu
- Inicijalne linije se mogu izvršavati paralelno
- Inicijalne linije se izvršavaju sekvencijalno sa novom linijom
- Linija se može podeliti na kraće linije
- Max linija je ona koja se ne može uključiti u dužu

# Serijalizacija

- ◆ Ako je nedefinisana, izvršenje programa nije definisano
- ◆ Linije se izvršavaju po redosledu serijalizacije
- ◆ Od dve linije, *ranija* je ona koja se izvršava pre u serijskom izvršenju (za iste ulazne podatke)
  - Bez obzira da li se izvršavaju redno ili paralelno
- ◆ Slično, *najranija*, *najkasnija* i *kasnija* su termini za označavanje linija prema njihovom serijskom redosledu
  - Sinonimi: *levo*, *desno*, *najlevlje* i *najdesnije*



# Pravilo serijalizacije

- ◆ Cilk Plus program bez zadataka je C/C++ program sa istim ponašanjem
- ◆ Ako Cilk Plus program ima deterministično ponašanje, onda je ono isto za C/C++ program
  - Nakon uklanjanja svih instanci `cilk_spawn` i `cilk_sync` ključnih reči i
  - Nakon zamene svake instance `cilk_for` ključne reči sa `for`

# \_Cilk\_for petlje (1/2)

- ◆ Jedna kontrolna promenljiva (inicijalizovana)
  - Može biti int, pokazivač ili tip klase
  - Ne može biti **const** niti **volatile**
- ◆ Uslov može imati dva oblika
  - *var OP shift-expression*
  - *shift-expression OP var*
    - ◆ *OP* je `!=`, `<=`, `<`, `>=`, ili `>`
- ◆ Ograničenja
  - Na tipove i operatore
  - Dodatna dinamička ograničenja (u izvršenju prog.)

## \_Cilk\_for petlje (2/2)

- ◆ Petlji može prethoditi grainsize pragma
- ◆ Sugerše broj serijskih iteracija u bloku paralelne petlje
- ◆ Ako ne postoji, veličina se bira heuristički
- ◆ Ako je vrednost izraza negativna, ponašanje nije specificirano (RFU)
- ◆ Veličina grainsize utiče samo na prvi sledeći for
  - Nema uticaja na naredne for petlje

# Mrešćenje zadatka (Spawn) (1/2)

- ◆ Sugerije da se iskaz može izvršavati u paraleli sa sledećim iskazima
- ◆ Posledica je moguće nedefinisano ponašanje
  - koje ne postoji u sekvencijalnom izvršenju programa
- ◆ Izvršenje `cilk_spawn` se naziva *mrešćenje*
- ◆ Izvršenje `cilk_sync` se naziva *sinhronizacija*, `sync`
- ◆ Sledeći `sync` je onaj koji sledi u izvršenju u istom Cilk bloku
  - A ne leksički (tj. redosledno u izvornom kodu)
  - Cilk blok ima implicitni `sync` na svom kraju

# Mrešćenje zadatka (Spawn) (2/2)

- ◆ Sve operacije u izrazu za mrešćenje, koji ne moraju biti redne obaviće se pre mrešćenja
- ◆ Linija koja započinje neposredno nakon tačke mrešćenja se naziva *nastavak* (iza mrešćenja)
- ◆ Niz operacija u iskazu mrešćenja je *potomak*
- ◆ Raspređivač može izvršiti potomka i nastavak u paraleli
- ◆ *Predak* je Cilk blok koji sadrži početnu liniju, iskaze mrešćenja i njihove nastavke, bez potomaka

# Sync

- ◆ Sync iskaz u Cilk bloku označava da se svi potomci moraju završiti pre nastavka izvršenja
- ◆ Nova linija koja izlazi iz sync može biti paralelna sa predkom i rođacima (drugi potomci predka)
- ◆ Ako se spawn pojavi u try bloku, implicitni sync je na kraju tog try bloka
- ◆ Ako nema potomaka u trenutku sync-a, on nema nikakvog efekta

# Hiperobjekti

- ◆ Omogućavaju siguran pristup deljenim objektima dajući svakoj paralelnoj liniji posebnu instancu
- ◆ Obraćanje hiperobjektu rezultuje u referenci
  - koja se naziva *pogled*
    - ◆ Pogled se stvara pozivom callback funkcije tipa hiperobjekta
    - ◆ Pogledi se spajaju, pri sync, u drugoj callback funkciji
    - ◆ Identitet (adresa) pogleda u jednoj liniji se ne menja
    - ◆ Pogled pre spawn i nakon sync je isti
      - Mada ID programske niti (thread) ne mora biti isti
    - ◆ Pogled pre i posle cilk\_for je isti
    - ◆ Specijalan, najraniji, pogled se stvara pri stvaranju hiperobjekta

# Reduktori (1/3)

- ◆ Hiperobjekti najčešće spadaju u reduktore
- ◆ Tip reduktora definiše callback operaciju *reduce*
  - koja spaja dva pogleda svojstveno reduktoru
  - $\text{reduce}(V1, V2)$  se označava kao  $V1 \circ V2$
  - Klasičan reduce je asocijativan  $(a \circ b) \circ c == a \circ (b \circ c)$
- ◆ Definiše i callback operaciju *identity*
  - koja inicijalizuje novi pogled,  $I$
  - $I \circ v == v$  i  $v \circ I == v$ , za bilo koji  $v$  tipa `value_type`
- ◆ Trojka  $(\text{value\_type}, \circ, I)$  opisuje matematički *monoid*



# Reduktori (2/3)

- ◆ Monoidi:  $(\text{int}, +, 0)$ ,  $(\text{list}, \text{concatenate}, \text{empty})$ ...
- ◆ Ako se svaki pogled reduktora modifikuje isključivo operacijama  $R \leftarrow R \circ v$ 
  - ◆ gde je  $v$  tipa `value_type`
  - Reduktor dolazi do istog rezultata u paralelnom programu, kao da je on serijalizovan
  - Operacija  $\circ$  može odgovarati skupu operacija, npr.  $+=$ ,  $-=$  su asocijativne.
  - Npr. telo `cilk_for` petlje može dodavati elemente na kraj reduktor liste – rezultatna lista je ista kao ona koja bi se generisala serijskim izvršenjem

# Reduktori (3/3)

- ◆ Kada niz linija  $S1\ S2... S_n$  sa pogledima  $V1\ V2... V_n$  ulazi u sync, rezultat je
  - jedan pogled  $W \leftarrow V1 \circ V2 \circ ... \circ V_n$
  - održava se redosled s-leva-na-desno
  - grupisanje operacija (asocijativnost) nije poznato
  - dinamika (timing) ove redukcije nije poznata
- ◆ Ako reduce nije asocijativna ili identity ne vraća pravi identitet, rezultat je nedeterminističan
- ◆ Reduce ne mora biti komutativna

# Naznake za nizove

- ◆ CEAN (C/C++ Extension for Array Notation)
- ◆ Direktno izražavanje paralelnih operacija nad nizovima na visokom nivou apstrakcije
  - Kompajler radi analizu zavisnosti, vektorizaciju i auto-paralelizaciju koda
- ◆ Jednostavno izražavanje
  - Operacija nad nizovima
  - Pojednostavljeno paralelno preslikavanje osnovnih funkcija preko nezavisnih ulazno-izlaznih tokova

# Operator sekcije (1/2)

- ◆ Bira više elemenata niza za paralelnu operaciju
- ◆ Opšti format
  - `<array base>[<lower bound>:<length>:<stride>]...`
  - Indeksna trojka: (prvi indeks, br. elemenata, korak)
    - ◆  $I_0, I_0+K, I_0+2K, I_0+(L-1)K$ ;  $L$ =br. elem.,  $K$ =korak
  - Korak  $K$  je opcion i podrazumevano je 1
  - Br. elemenata (length) ne sme biti manji od 1
  - Cela dimenzija niza se označava sa :
  - Korak može biti negativan, a ne može biti jednaka 0
  - Npr. `A[:,:]`, `A[1:5:2][:]`, `A[1:5][2:4]`

# Operator sekcije (2/2)

- ◆ *Sekcija niza* je određena indeksnim trojkama
  - $A[0:3][0:4]$  = 12 elemenata od  $A[0,0]$  do  $A[2,3]$
  - $A[0:2:3]$  =  $A[0]$  i  $A[3]$
- ◆ *Oblik* sekcije je određen sa n indeksnih trojki
  - $((\text{length}_0, \text{stride}_0), \dots, (\text{length}_{n-1}, \text{striden}_{n-1}))$
- ◆ *Rang* je jednak broju indeksnih trojki
  - $A[3:4][0:10]$ ,  $A[3][0:10]$ ,  $A[3:4][0]$ ,  $A[:, :]$  i  $A[3][0]$   
rang je 2, 1, 1, 2 i 0
- ◆ *Relativni rang indeksne trojke* je njen redni broj
  - U  $A[1][0:10][0]$ ,  $A[0:10][1][2]$ ,  $A[2][x][0:10]$ ,  
relativni rang trojke 0:10 je 2, 1 i 3

# Operacije nad sekcijama niza

## ◆ Dodela

- Paralelna operacija za sve elemente se leve strane
- Npr.  $A[4:3] = A[3:3];$ 
  - ◆  $A[3], A[4], A[5]$  se kopiraju u  $A[4], A[5], A[6]$
  - ◆ Kompajler obezbeđuje dodatni prostor tako da upisi u  $A[4]$  i  $A[5]$  ne ometaju čitanja iz istih lokacija
- Još primera:
  - // Copy elements 10->19 in A to elements 0->9 in B.
  - $B[0:10] = A[10:10];$
  - // Error. Triplets 0:10 and 0:100 are not the same size.
  - $B[0:10] = A[0:100];$

# Aritmetičke operacije nad nizovima

◆ `+, -, *, /, %, <, ==, >, <=, !=, >=, ++, --, |, &, ^, &&, ||, !, -(unarno), +(unarno)`

## ◆ Primeri

```
// Set all elements of A to 1.0.
```

```
A[:] = 1.0;
```

```
// Element-wise addition of all elements in A and B, result in C.
```

```
C[:] = A[:] + B[:];
```

```
// Matrix addition of the 2x2 matrices in A and B starting at
```

```
// A[3][3] and B[5][5].
```

```
C[0:2][0:2] = A[3:2][3:2] + B[5:2][5:2];
```

```
// ??? – za domaći
```

```
C[0:9][0][0:9] = A[0][0:9][0:9] + B[0:9][0:9][4];
```

# Operacije redukcije nad nizom

## Elementi se akumuliraju zadatom funkcijom

*type* **fn**(*type* in1, *type* in2); // declaration of scalar reduction function  
*type* in[N], out; // array input and scalar output result

// accumulate successive elements in *in* with a user function *fn*,  
// resulting in a single value *out*

**out = \_\_sec\_reduce(fn, identity\_value, in[x:y:z]);**

**out = \_\_sec\_reduce\_add(in[x:y:z]); // out = sum of all values**

**out = \_\_sec\_reduce\_mul(in[x:y:z]); // out = product of all values**

**out = \_\_sec\_reduce\_all\_zero(in[x:y:z]); // 1 if all values are zero**

**out = \_\_sec\_reduce\_all\_nonzero(in[x:y:z]); // 1 if all values nonzero**

**out = \_\_sec\_reduce\_any\_nonzero(in[x:y:z]); // 1 if any nonzero**

**out = \_\_sec\_reduce\_max(in[x:y:z]); // max value of values in *in***

**out = \_\_sec\_reduce\_min(in[x:y:z]); // min value of values in *in***

**out = \_\_sec\_reduce\_max\_ind(in[x:y:z]); // index of maximum value**

**out = \_\_sec\_reduce\_min\_ind(in[x:y:z]); // index of minimum value**



# Operacije razbacivanja i skupljanja (scatter-gather)

- ◆ Skupljanje: elementi iz *in[]* specificirani sa *index[x:y:z]* skupljaju se u *out[a:b:c]*
- ◆ Razbacivanje: elemente iz *in[a:b:c]* razbacaju u *out[]* po rasporedu *index[x:y:z]*

```
unsigned int index[N];  
type out[M],in[O];
```

```
// gather elements from in[], given by index[x:y:z], into out[]  
out[a:b:c] = in[index[x:y:z]];
```

```
// scatter elements from in[] into various locations in out[],  
// given by index[x:y:z]  
out[index[x:y:z]] = in[a:b:c];
```

# Operacije preslikavanja (Map)

- ◆ Ako se skalarna C/C++ funkcija poziva sa sekcijama nizova kao argumentima, ona se *preslikava*
  - Sukcesivno se poziva za odgovarajuće elemente

```
type fn(type arg1, type2 arg2); // declaration of scalar function  
type in[N], out[N];  
type2 in2[N];
```

```
out[x:y:z] = fn(in[x:y:z], in2[x:y:z]);
```

# Sekcije nizova kao parametri

- ◆ CEAN podržava vektorsko programiranje
  - Programski kod je ugrađen u funkciji
    - ◆ Dužina vektora je parametrizovana
    - ◆ Sve paralelne operacije se obavljaju u telu funkcije
  - Primer: vektorizacija u kontekstu OpenMP
    - ◆ Vektorizacija u telu funkcije, m=256

```
void saxpy_vec(int m, float a, float restrict (&x)[m], float (&y)[m]) {  
    y[:] += a * x[:];  
}  
  
void main(void) {  
    int a[2048], b[2048];  
    #pragma omp parallel  
    for (int i = 0; i < 2048; i += 256)  
        saxpy_vec(256, 2.0, &(a[i]), &(b[i]));  
}
```

# Osnovne (elemental) funkcije

- ◆ Podržavaju paralelnu obradu podataka
- ◆ Korišćenje osnovnih funkcija – 3 koraka:
  - programer piše skalarnu funkciju
    - ◆ koja opisuje operaciju nad jednim elementom
  - dodaje `__declspec(vector)` sa dodatnim klauzulama
    - ◆ Kompajler projektuje skalarne operacije na vektorske implementacije koje operišu nad vektorom elemenata
  - piše pozive funkcija sa nizovima kao argumentima
    - ◆ umesto pojedinačnih elemenata
    - ◆ funkcija se poziva iterativno dok se ne obrade svi elementi
    - ◆ Svaki takav poziv je jedna *instanca* funkcije

# Semantika osnovnih funkcija

## ◆ Zavisi od mesta poziva

- Iz C/C++ petlje: kompajler može izvršiti zamenu
  - ◆ zavisno od implementacije i heuristike za performansu
- `#pragma simd` pre C/C++ petlje: uvek se zamenjuje
  - ◆ Instance funkcije se pozivaju u jednoj liniji (strand) izvršenja
- Iz Cilk petlje: uvek se zamenjuje
  - ◆ Instance funkcije se izvršavaju paralelno u više linija
- Ako se koriste naznake za nizove, izvršenje kao kad je zadata `#pragma simd`

## ◆ Klauzule: `processor(cpuid)`, `vectorlength(n)`, itd.

# Pragma SIMD

sugeriše kompajleru da koristi vektorske instrukcije

## ◆ Klauzule SIMD pragme

- `vectorlength(num1, num2, ..., numN)`
  - ◆ Izaberi jednu od datih dužina
- `private(var1, var2, ..., varN)`
  - ◆ Privatne promenljive za svaku iteraciju petlje
- `linear(var1:step1, var2:step2, ..., varN:stepN)`
  - ◆ Za svaku iteraciju *var* se menja za zadati *step*
- `reduction(operator:var1, var2,..., varN)`
  - ◆ Primeni redukciju tipa *operator* na zadate promenljive
- `[no]assert`
  - ◆ (ne)reaguj ako se ne može generisati vektorski kod