Týden 7.
Programování se sdílenými proměnnými.
Jacobiho a GS iterace.
Multivláknové programování v OpenMP.

Prototyp řešení lin. rovnic (Gauss-Jordan)

```
int n=1000,P=20,pivot=0,lock, barcnt=0, flag; #sdílené prom.
process worker [id = 0 to P-1] {#každý má svoje rovnice již v cache
  int low = id*n/P, high = low+n/P-1, coef;
  for [i=0 to P-1] { # P fází výpočtu
    while (pivot < n) {
      for [m=low to high,m!= pivot] {#pivotní řádek beze změny
        coef = a[m][pivot]/a[pivot][pivot];
        for [ k=pivot+1 to n+1] #pravá strana připojena k matici A
          a[m][k]=a[m][k]-coef*a[pivot][k]; # všechny CPU
      }
      if (id=i) pivot = pivot+1;   #jen řídící   procesor
      barrier(barcnt);
    }
  }
}
```

komunikace se SM přes sběrnici

Jacobiho iterace Laplaceovy rovnice - první **sekvenční** verze

```
real grid[0:n+1,0:n+1],new [0:n+1,0:n+1];
while (true) {
  # vypočti nové hodnoty všech vnitřních bodů
  for [i=1 to n, j=1 to n]
    new[i,j]=(grid[i-1,j]+grid[i+1,j]+
              grid[i,j-1]+grid[i,j+1])/4;
  iters++;
  #vypočti maximální rozdíl nové a staré hodnoty
  maxdiff = 0.0;
  for [i=1 to n, j=1 to n]
    maxdiff = max(maxdiff, abs(new[i,j]-grid[i,j]));
  if (maxdiff < epsilon)    #kontrola zda končit
    break;
  for [i=1 to n, j=1 to n] #přepiš new do grid
    grid[i,j] = new[i,j];  #pro příští běh
}
```

Jacobiho iterace Laplaceovy rovnice - druhá, efektivnější **sekvenční** verze.

argument v příkazovém řádku

```
for [iters = 1 to MAXiters] {
  # vypočti nové hodnoty všech vnitřních bodů
  for [i=1 to n, j=1 to n]
    new[i,j]=(grid[i-1,j]+grid[i+1,j]+
              grid[i,j-1]+grid[i,j+1])* 0.25;
  for [i=1 to n, j=1 to n] #přepiš new do grid
    grid[i,j] = new[i,j];  #pro příští běh
}
#vypočti maximální rozdíl nové a staré hodnoty
maxdiff = 0.0;
for [i=1 to n, j=1 to n]
    maxdiff = max(maxdiff, abs(new[i,j]-grid[i,j]));
```

Jacobiho iterace Laplaceovy rovnice - náhrada kopírování přepínáním matic.

```
real grid[0:1][0:n+1,0:n+1];
int old = 0, new = 1;
for [iters = 1 to MAXiters] {
  # vypočti nové hodnoty všech vnitřních bodů
  for [i=1 to n, j=1 to n]
    grid[new][i,j]=(grid[old][i-1,j]+grid[old][i+1,j]+
              grid[old][i,j-1]+grid[old][i,j+1])* 0.25;
  #zaměň mřížky
    old = 1-old; new = 1-new;
}
#vypočti maximální rozdíl nové a staré hodnoty
maxdiff = 0.0;
for [i=1 to n, j=1 to n]
    maxdiff = max(maxdiff, abs(new[i,j]-grid[i,j]));
```

dvojí indexování je časově náročnější na zpracování!

Jacobiho iterace Laplaceovy rovnice - optimalizovaný **sekvenční** program

```
real grid[0:n+1,0:n+1],new [0:n+1,0:n+1];
real maxdiff = 0.0;
inicializuj mřížky včetně hranic;
for [iters = 1 to MAXiters by 2] {
  # vypočti nové hodnoty všech vnitřních bodů
  for [i=1 to n, j=1 to n]
    new[i,j]=(grid[i-1,j] + grid[i+1,j]+
              grid[i,j-1] + grid[i,j+1]);
  # vypočti opět nové hodnoty vnitřních bodů
  for [i=1 to n, j=1 to n]
    grid[i,j]=(new[i-1,j] + new[i+1,j]+
              new[i,j-1] + new[i,j+1])* 0.625;
}
#vypočti maximální rozdíl
for [i=1 to n, j=1 to n]
    maxdiff = max(maxdiff, abs(new[i,j]-grid[i,j]));
tisk konečné mřížky a maximálního rozdílu;
```

dvojí rozbalení smyčky

volání funkcí a režii s tím spojenou lze ještě odstranit (inlining)

Jacobiho iterace Laplaceovy rovnice - **paralelní** program se SV ve stylu SPMD.

```
real grid[0:n+1,0:n+1],new [0:n+1,0:n+1]; #uloženy po řádcích
real maxdiff[1:P] = ([P] 0.0);
int rows1p = n/P; # předpokládáme, že P dělí n

process worker [w = 1 to P] {
  int firstRow = (w-1)*rows1p + 1;
  int lastRow = firstRow + rows/P - 1;
  real mydiff = 0.0;
  inicializuj si svoje proužky grid a new, včetně hranic;
  barrier(w); #obecně je kód (motýlkové) bariéry funkcí id

for [iters = 1 to MAXiters by 2] {
  # vypočti nové hodnoty ve svém proužku
  for [i=firstRow to lastRow, j=1 to n]
    new[i,j]=(grid[i-1,j] + grid[i+1,j]+
              grid[i,j-1] + grid[i,j+1])* 0.25;
  barrier(w); #protože sousední procesy sdílejí hraniční body při čtení
```

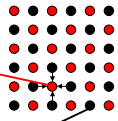
Jacobiho iterace Laplaceovy rovnice - **paralelní** program se SV, pokrač.

```
    #vypočti zase nové hodnoty ve svém proužku
    for [i=firstRow to lastRow, j=1 to n]
      grid[i,j]=(new[i-1,j] + new[i+1,j]+
                 new[i,j-1] + new[i,j+1])* 0.25;
    barrier(w); #volání funkce může kompilátor nahradit kódem „in-line“
  }
  # vypočti maximální rozdíl pro můj proužek
  for [i=firstRow to lastRow, j=1 to n]
      mydiff = max(mydiff, abs(new[i,j]-grid[i,j]));
      maxdiff[w] = mydiff;
      barrier(w);
  # maximální rozdíl je maximální prvek maxdiff[*].
  #vypočte si jej každý procesor sám nebo jeden pro všechny
}
```

Použití lokální proměnné mydiff a vektoru **maxdiff[*]** :
1. nemusí se použít CS pro vstup do jedné globální proměnné **maxdiff**
2. zamezí se falešnému sdílení **maxdiff[*]**

Příklad: Gauss-Seidelovy iterace, červeno-černé schéma

```
.....začátek stejný jako u Jacobiho iterací, navíc je deklarována int jStart;
for [iters = 1 to MAXiters by 2]
   for [i=firstRow to lastRow] {
   if (i%2==1) jStart = 1; #liché řádky
   else jStart = 2;        #sudé řádky
   for [j = jStart to n by 2]
     grid[i,j]=(grid[i-1,j] + grid[i+1,j]+
         grid[i,j-1] + grid[i,j+1])* 0.25;
   }
   barrier(w);
   for [i=firstRow to lastRow] {
     if (i%2==1) jStart = 2; #liché řádky
     else jStart = 1;        #sudé řádky
     for [j = jStart to n by 2]
       grid[i,j]=(grid[i-1,j] + grid[i+1,j]+
                  grid[i,j-1] + grid[i,j+1])* 0.25;
  }
  barrier(w);      # dál pokračuje jako Jacobiho iterace
```



OpenMP C/C++ API

http://www.openmp.org

- bylo vyvinuto konsorciem hlavních výrobců hw a sw pro výpočty s vysokou výkonností, rozhraní pro C/C++ je z r. 1998 (pro Fortran z r. 1997),
- slouží k vyjádření paralelismu se sdílenou pamětí, důraz je na paralelizaci smyček
- hlavní část rozhraní OpenMP tvoří direktivy pro kompilátor; programátor je vkládá do sekvenčního programu, aby
  - sdělil kompilátoru které úseky provádět souběžně,
  - specifikoval body synchronizace,
- direktivy je možné přidávat inkrementálně, což umožňuje paralelizovat již existující sw;
- Pthreads a MPI byly postaveny na knihovních programech, spojených s a volaných ze sekvenčního programu; to vyžadovalo, aby programátor rozdělil práci ručně (pracnější).

Principy OpenMP API

Model paralelní činnosti: fork-join. Hlavní vlákno (master) vytvoří tým vláken, členové týmu provádějí příkazy paralelně a synchronizují se na bariéře (implicitní, explicitní).
Program se může při běhu větvit (fork) a zase spojit (join) mnohokrát.

OpenMP:
- direktivy :
  - konstrukty SPMD ,
  - konstrukty sdílení práce,
  - synchronizační konstrukty
- knihovní funkce
- proměnné prostředí

✓ aktivace volbami na příkazovém řádku,
✓ umožňuje přenositelnost mezi architekturami SM,
✓ pokrývá pouze paralelizmus řízený uživatelem,
✓ uživatel je odpovědný za správnou činnost, absenci deadlocků, konfliktů a souběhů.

OpenMP: direktiva parallel

- definuje paralelní oblast (region) prováděnou || více vlákny
- zahajuje paralelní provádění

#pragma omp parallel [clause[clause] …]
strukturovaný blok

- **if**-clause jedinečná u || direktivy; tým vláken se vytvoří jen když scalar-exp ≠ 0
- hlavní vlákno má číslo 0;
- počet vláken v týmu zůstává uvnitř paralelní oblasti konstantní. Může jej změnit - uživatel
  - runtime systém
- na konci paralelní oblasti je automaticky bariéra

if(scalar-exp)
private(list)
firstprivate(list)
lastprivate(list)
default(shared | none)
shared(list)
copyin(list)
reduction(operator: list)

OpenMP: datové prostředí

- privátní proměnné vlákna: `#pragma omp threadprivate (list)`
- `copyin(list)` … privátní kopie hodnot proměnných jsou vytvořeny na začátku paralelní oblasti
- `private(list)` … privátní proměnné každého vlákna v týmu
- `firstprivate(list)` … pro inicializaci jsou použity původní hodnoty těchto proměnných
- `lastprivate(list)` … hodnota proměnné z poslední iterace je přiřazena původnímu objektu proměnné
- `shared(list)` … proměnné sdílené všemi vlákny v týmu
- nespecifikovaná proměnná je sdílená, stejně jako když je použit `default(shared)`);
- proměnná musí být v některém `(list)` když je použit `default(none)`;
- paměť alokovaná jako hromada je sdílená, ukazatel do této paměti může být buď privátní nebo sdílený.

Direktiva OpenMP pro sdílení práce : for

`#pragma omp for` [clause[clause] …]
for-loop

- počet iterací smyčky je spočitatelný na vstupu do smyčky
- schedule kind: `static, dynamic, guided, runtime` (příklady později)
- `nowait` - na konci smyčky není automatická bariéra
- `ordered` clause je jedinečná pro tuto direktivu, provedení v pořadí sekvenční smyčky!

```
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
ordered
schedule(kind[, chunk_size])
nowait
```

Plánování smyček v OpenMP

```
#pragma omp parallel for schedule(static[, chunk_size])
   for (i=0; i<n; i++) {
     the_same_work(i);
}
```

- i = implicitně privátní proměnná, nemusí být na privátním seznamu
- direktivy `parallel` a `for` jsou zkombinovány do jedné
- každé vlákno umí určit své iterace bez exkluzivního přístupu ke sdílenému i
- když není udán chunk_size, dostane každé vlákno cca stejný počet iterací. Je-li `n`= t*q - r, pak je možné rozdělení iterací
  {q, q, …, q, q-r} nebo {q, q, …, q, q-1, q-1,…, q-1}.
- je-li `work1(i)` následovaná `work2(i)`, může být přednější, aby každé vlákno pracovalo se stejnými objekty i když vlákna nepředstavují přesně stejný objem práce, a to kvůli charakteristikám paměťového systému (velikost cache, NUMA nebo UMA) → také použij schedule(static).

Plánování smyček v OpenMP - pokrač.

```
#pragma omp parallel for schedule(dynamic[, chunk_size])
   for (i=0; i<n; i++) {
     unpredictable_amount_of_work(i);
}
```

- žádné vlákno nečeká na bariéře déle než trvá jinému vláknu dokončit jeho posledních k (= chunk_size) iterací
- iterace jsou přidělovány po blocích k ≥ 1, se synchronizací při každém přidělení
- chunk_size k > 1 : menší synchronizační režie

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

- schedule kind a chunk_size mohou být zvoleny v době běhu pomocí proměnné prostředí `OMP_SCHEDULE`; to je pohodlné pro experimentování, nevyžaduje to rekompilaci.

Plánování smyček v OpenMP - pokrač.

```
#pragma omp parallel for schedule(guided[, chunk_size])
   for (i=0; i<n; i++) {
     the_same_work(i)
}
```

- užitečné v případě, že vlákna přijdou k `for` v různých časech a každá iterace vyžaduje zhruba stejnou práci
- žádné vlákno nečeká na bariéře déle než trvá jinému vláknu dokončit jeho posledních k (= chunk_size) iterací
- vyžaduje nejmenší počet synchronizací
- pro tým t vláken a chunk_size = k, typická implementace
  - přidělí prvnímu vláknu které je k dispozici q = ⌈n/t⌉ iterací,
  - položí n := max (n - q, t*k) a
  - opakuje 2 předchozí kroky až jsou všechny iterace rozdány.
- velikost bloku iterací se snižuje až na k (default: k = 1).

Direktivy OpenMP pro sdílení práce : sections

- specifikuje množinu příkazů, která se má rozdělit mezi vlákna týmu
- neiterativní sdílení práce, každá sekce je provedena jednou nějakým vláknem v týmu

```
#pragma omp sections [clause[ clause] …]
{
   [#pragma omp section]
      xaxis();
   [#pragma omp section
      yaxis();
   ... ]
}  /** bariéra je implicitní pokud se nepoužije nowait **/
```

```
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
nowait
```

Zkratka : `#pragma omp parallel sections` =
`#pragma omp parallel`
`#pragma omp sections`

OpenMP: zpracování jedním vláknem

- není to nutně hlavním vláknem (master), ale prvním vláknem, které dojde k direktivě
  `single`
- užitečné pro tisk zpráv o postupu řešení
- obsahuje implicitní bariéru pokud se nepoužije `nowait`

```
#pragma omp single [clause[ clause] …]
   printf ("Work done.\n");
```

```
private(list)
firstprivate(list)
nowait
```

```
#pragma omp master
```

strukturovaný blok

implicitní bariéru na vstupu nebo výstupu neobsahuje.

Synchronizační příkazy a direktivy

V daném čase provádí kritickou sekci jediné vlákno:

```
#pragma omp critical [(name)]
```

Na konci bloku:

```
#pragma omp barrier
```

Lepší optimalizaci než `critical` poskytuje

```
#pragma omp atomic
   expression-stmt
```

Příklad:

```
#pragma omp parallel for shared (x,y,index,n)
    for (i=0; i<n; i++) {
      #pragma omp atomic
        x[index[i]] += work1(i);
      y[i] += work2(i);
    }
```

Aktualizace dvou **různých** prvků x *nemohou* *mohou* proběhnout paralelně

"atomic" se aplikuje jen na tento příkaz

OpenMP: vnořený paralelismus

- Direktiva parallel uvnitř jiného parallel stanoví dynamicky nový tým složený jen ze současných vláken, pokud není povolen vnořený paralelismus (nastavením env. proměnné `OMP_NESTED` na TRUE nebo voláním funkce runtime knihovny `omp_set_nested(int nested)`;
- direktivy `for`, `sections`, a `single` které jsou vázány k téže direktivě `parallel` nesmí být vnořovány;
- direktiva `barrier` se váže k nejbližší obklopující direktivě `parallel`. Bariéry nejsou dovoleny v dynamickém dosahu `for`, `ordered`, `sections`, `single`, `master`, a úseků `critical`.
- direktivy `critical` se stejným jménem nemohou být vnořovány, `ordered` není dovoleno v dynamickém dosahu úseků `critical`.

OpenMP: direktiva flush

- specifikuje sekvenční bod napříč vlákny, na kterém všechna vlákna týmu vidí konsistentně udané objekty v paměti;
- předchozí vyhodnocování výrazů odkazující na tyto objekty je dokončeno a následné vyhodnocování ještě nezačalo;
- explicitní flush:
  `#pragma omp flush [(list)]`
- flush bez seznamu (více režie než se seznamem!) synchronizuje všechny sdílené objekty s výjimkou nepřístupných objektů s automatickým trváním v paměti; pokud se nepoužije `nowait`, je `flush` implicitní
  - pro `barrier`,
  - na výstupu z `parallel`, `for`, `sections`, `single`
  - na vstupu do a na výstupu z `critical`, `ordered`.
- direktiva `flush` musí být obsažena v bloku; `flush` není příkaz.
- odkaz/modifikace volatilního objektu ≈ `flush` specifikující ten objekt na předchozím/následujícím sekvenčním bodě.

Dvoubodová synchronizace

specifických objektů mezi dvojicemi vláken použitím direktivy flush

```
#pragma omp parallel private(iam, neighbor) shared(work,
sync)
{
  iam = omp_get_thread_num(); /*muj index*/
  sync[iam] = 0;
  #pragma omp barrier
/*do computation into my portion of work array */
work[iam]= ...;
#pragma omp flush(work) /*moje práce zviditelněna*/
sync[iam] = 1;
#pragma omp flush(sync) /*sync je zviditelněna*/
neighbor =(iam >0? iam : omp_get_num_threads()) - 1
while (sync[neigbor]==0) {
  #pragma omp flush(sync) /*abych uviděl co nejdříve 1*/
... = work[neighbor];  /*sousedova práce viditelná*/
}
```

OpenMP: direktiva ordered

- musí být uvnitř dynamického rozsahu `for` nebo `parallel` s klauzulí `ordered`
- užitečné pro sekvenční uspořádání výstupu výpočtů prováděných || :

```
#pragma omp for ordered schedule(dynamic)
  for(i=lb; i< ub; i+= step)
    work(i);

void work(int k)
{
  #pragma omp ordered
    printf(" %d", k);
}
```

OpenMP: příklad použití zámků

```
#include <omp.h>
int main()
{
  omp_lock_t lck;  /* typ zámkové proměnné */
  int id;
  omp_init_lock(&lck); /* požaduje pointer na lck */
  #pragma omp parallel shared(lck) private(id)
  {
    id = omp_get_thread_num();
    while (! omp_test_lock(&lck)) { /*neblokuje*/
     skip(id);  /*pokud test_lock vrací 0, testuj*/
    }
    work(id);  /*teď máme zámek a můžeme dělat CS */
    omp_unset_lock(&lck); /*duální set_lock blokuje */
  }
  omp_destroy_lock(&lck);
}
```

OpenMP: klauzule reduction

- **reduction** (*op*: *list*) provádí redukci skalárních proměnných které jsou na seznamu *list*, operátorem *op;*
- direktivou lze specifikovat libovolný počet redukcí
- proměnné vyskytující se v redukci musí být v okolním kontextu sdílené; je vytvořena privátní kopie každé proměnné ze seznamu *list* (jedna pro každé vlákno) a inicializována. Na konci je aktualizována sdílená proměnná

Příklad:

```
#pragma omp parallel for reduction(+: a,y)
reduction(||: am)
for (i=0; i<n; i++)  {
     a += b[i]
     y = sum(y, c[i]);
     am = am || b[i] == c[i];
}
```

Příklad: Jacobiho iterace v pseudoC+OpenMP

(bez deklarací proměnných a paralelní inicializace grid a new)

```
#pragma omp parallel shared (n,maxiters,grid,new,
            maxdiff) private(i,j,iters,tempdiff)
for [iters = 1 to maxiters by 2] {
  #pragma omp for schedule(static)
    for [i = 1 to n] {
      for [j = 1 to n] {
        new(i,j) = ...
      }
    }
  #pragma omp for schedule(static)
     for [i = 1 to n] {
       for [j = 1 to n] {
         grid(i,j) = ...
       }
     }
```

vnější smyčky jsou naplánovány staticky, každé vlákno si vezme cca n/P řádků mřížky

Počet vláken v týmu je zvolen
- uživatelem staticky (env. variable)
- uživatelem dynamicky, zavoláním knihovního programu
- kompilátorem (default)

Jacobiho iterace v OpenMP - pokrač.

```
   #pragma omp for schedule(static)
                  reduction(max: maxdiff)
    for [i = 1 to n] {
      for [j = 1 to n] {
        tempdiff = abs(new[i,j]-grid[i,j]);
        maxdiff = max(maxdiff,tempdiff) #atomicky
      }
    }
}
```

OpenMP implementuje atomickou aktualizaci použitím privátních proměnných tempdiff v každém procesu worker; hodnoty těchto privátních proměnných jsou potom atomicky spojeny do jedné hodnoty na implicitní bariéře na konci smyčky.

Maticové násobení, sekvenční program

```
#Matrix-Matrix-Multiplication, MMM
double a[n,n], b[n,n], c[n,n];

for [i = 0 to n-1] {
  for [j = 0 to n-1] {
    #spočítej skalární součin a[i,*] a b[*,j]
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Paralelní MMM (proužky, bloky)

```
double a[n,n], b[n,n], c[n,n];

process worker[w = 1 to P] {#proužky paralelně
  int first=(w-1) * n/P;    #první řádek proužku
  int last=first + n/P -1;  #poslední řádek proužku
  for [i = first to last] {
    for [j = 0 to n-1] {
      c[i,j] = 0.0;
      for [k = 0 to n-1] #smyčka skalárního součinu
        c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
  }
}
```

statické rozdělení práce, v OpenMP schedule (static, n/P)