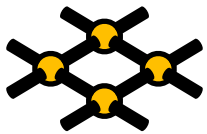
Vláknové programování
část V

Lukáš Hejmánek, Petr Holub
{xhejtman,hopet}@ics.muni.cz



Laboratoř pokročilých síťových technologií

PV192
2011–04–14

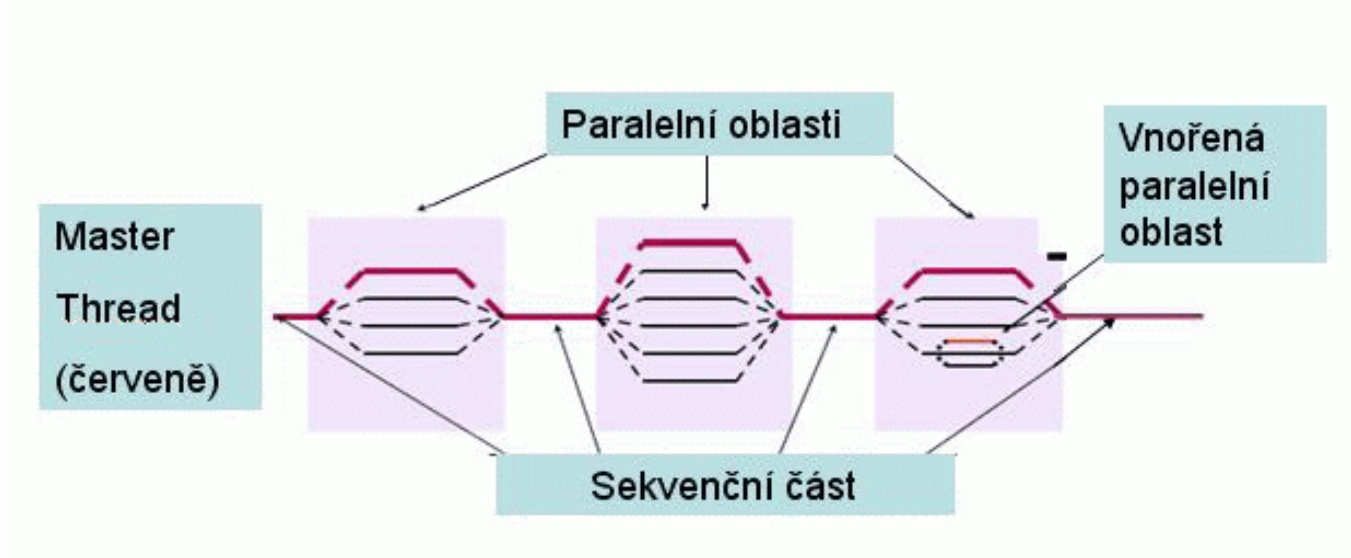
Přehled přednášky

Open MP

- Standard pro programování se sdílenou pamětí
- Podpora v programovacích jazycích Fortran (od r. 1997), C, C++ (od r.1998)
- Současná verze 3.0 je z roku 2008 pro Fotran i C/C++
- Podporováno řadou překladačů vč. gcc a Intel cc
- Podpora paralelního programování pomocí
  - Soustavy direktiv pro překladač
  - Knihovních procedur
  - Proměnných prostředí
- API spadá do tří kategorií:
  - Výrazy pro paralelismus (flow control)
  - Sdílení dat mezi vlákny (communication)
  - Synchronizace (coordination or interaction)

Programovací model OpenMP

- Explicitní paralelismus
- Fork/join model



- Vnořený (nested) paralelismus není vždy dostupný (uvidíme u knihovních funkcí)

Překlad

- **`gcc -g -o foo foo.c -fopenmp -D_REENTRANT`**
- Aplikace je slinkována s knihovnami **`libgomp`** a **`libpthread`**.

Open MP příkazy

- Přehled syntaxe
- Parallel
- Loop
- Sections
- Task (Open MP 3.0+)
- Synchronization
- Reduction

Přehled syntaxe

- Základní formát
  `#pragma omp jméno-příkazu [klauzule] nový_řádek`
- Všechny příkazy končí novým řádkem
- Používá konstrukci pragma (pragma = věc)
- Rozlišuje malá/velká písmena
- Příkazy mají stejná pravidla jako C/C++
- Delší příkazy lze napsat na více řádků pomocí escape znaku \

Parallel

- Blok kódu prováděn několika vlákny
- Syntaxe:

```
1 #pragma omp parallel private(list)\
2         shared(list)
3 {
4         /* parallel section */
5 }
```

Parallel – příklad

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (int argc, char *argv[]) {
5          int tid;
6          printf("Hello world from threads:\n");
7  #pragma omp parallel private(tid)
8          {
9                  tid = omp_get_thread_num();
10                 printf("<%d>\n", tid);
11         }
12         printf("I am sequential now\n");
13         return 0;
14 }
```

- Výstup:

```
Hello world from threads:
<1>
<0>
I am sequential now
```

Loop

- Iterace smyčky budou prováděny paralelně
- Na konci smyčky je implicitní barriéra, není-li řečeno jinak (**nowait**)
- Syntaxe:

```
1 #pragma omp for schedule(type [,chunk]) \
2         private(list) shared(list) nowait
3 {
4         /* for loop */
5 }
```

Které smyčky jsou paralelizovatelné?

Paralelizovatelné

- Počet iterací je znám předem a nemění se
- Iterátory (C++) (platí pro Open MP 3.0 a novější)
- Každá iterace nezávisí na žádné ostatní
- Nezávislost dat

Neparalelizovatelné

- Podmíněné smyčky (často **`while`** smyčky)
- Iterátory (C++) (neplatí pro Open MP 3.0 a novější)
- Závislé iterace
- Závislá data

Lze paralelizovat?

```
1  /* Gaussian Elimination (no pivoting):
2     x = A\b */
3  for (int i = 0; i < N-1; i++) {
4          for (int j = i; j < N; j++) {
5                  double ratio = A[j][i]/A[i][i];
6                  for (int k = i; k < N; k++) {
7                          A[j][k] -= (ratio*A[i][k]);
8                          b[j] -= (ratio*b[i]);
9                  }
10         }
11 }
```

Lze paralelizovat?

- Vnější smyčka (`i`)
  - `N-1` iterací
  - Iterace závisí na ostatních (hodnoty spočítané v kroku `i-1` jsou použity v kroku `i`)
- Vnitřní smyčka (`j`)
  - `N-i` iterací (konstanta pro konkrétní `i`)
  - Iterace mohou být provedeny v libovolném pořadí
- Nejvnitřnější smyčka (`k`)
  - `N-i` iterací (konstanta pro konkrétní `i`)
  - Iterace mohou být provedeny v libovolném pořadí

```
1  /* Gaussian Elimination (no pivoting):
2     x = A\b */
3  for (int i = 0; i < N-1; i++) {
4  #pragma omp parallel for
5          for (int j = i; j < N; j++) {
6                  double ratio = A[j][i]/A[i][i];
7                  for (int k = i; k < N; k++) {
8                          A[j][k] -= (ratio*A[i][k]);
9                          b[j] -= (ratio*b[i]);
10                 }
11         }
12 }
```

Poznámka: lze kombinovat **`parallel`** a **`for`** do jednoho **`pragma`** příkazu

Plánování smyček

- Výchozí plánování je dáno konkrétní implementací
- Rozlišujeme statické a dynamické plánování
- Statické
  - ID vlákna provádějící konkrétní iteraci je funkcí čísla iterace a počtu participujících vláken
  - Vlákna jsou staticky přidělena před startem smyčky
  - Rozložení zátěže může být problém, pokud iterace nejsou stejně dlouhé
- Dynamické
  - Přiřazení vláken proběhne až při běhu aplikace (round robin princip)
  - Každé vlákno může pokračovat v další práci, pokud současnou dodělalo
  - Rozložení zátěže je možné

```
1  #include <omp.h>
2
3  #define CHUNKSIZE 100
4  #define N 1000
5
6  int main () {
7          int i, chunk;
8          float a[N], b[N], c[N];
9          /* Some initializations */
10         for (i=0; i < N; i++)
11                 a[i] = b[i] = i * 1.0;
12         chunk = CHUNKSIZE;
13 #pragma omp parallel shared(a,b,c,chunk) private(i)
14         {
15 #pragma omp for schedule(dynamic,chunk) nowait
16                 for (i=0; i < N; i++)
17                         c[i] = a[i] + b[i];
18         } /* end of parallel section */
19         return 0;
20 }
```

Section(s)

- Neiterativní spolupráce
- Rozdělení bloků programu mezi vlákna
- Syntaxe:

```
#pragma omp sections
{
#pragma omp section
        /* first section */
#pragma omp section
        /* next section */
}
```

Section(s) příklad

```
1  #include <omp.h>
2  #define N 1000
3  int main () {
4          int i;
5          double a[N], b[N], c[N], d[N];
6          /* Some initializations */
7          for (i=0; i < N; i++) {
8                  a[i] = i * 1.5;
9                  b[i] = i + 22.35;
10         }
11 #pragma omp parallel shared(a,b,c,d) private(i)
12         {
13 #pragma omp sections nowait
14                 {
15 #pragma omp section
16                         for (i=0; i < N; i++)
17                                 c[i] = a[i] + b[i];
18 #pragma omp section
19                         for (i=0; i < N; i++)
20                                 d[i] = a[i] * b[i];
21                 } /* end of sections */
22         } /* end of parallel section */
23         return 0;
24 }
```

Task – Open MP 3.0+

- Koncepce spuštění bloku kódu na „pozadí“
- Některé kusy kódu jdou špatně paralelizovat, např.:

```
1 while(my_pointer) {
2                 (void) do_independent_work (my_pointer);
3                   my_pointer = my_pointer->next ;
4 } // End of while loop
```

- **do_indipendent_work** by mohlo běžet v pozadí
- Pro starší OpenMP – napřed spočítat počet iterací, pak převést *while* na *for*
- Koncepce tasku:
  - Smyčka běží v jediném vlákně (kvůli procházení seznamu)
  - **do_independent_work** se pustí do pozadí
- Syntaxe:
  #pragma omp task

Task – příklad

```
1  my_pointer = listhead;
2  #pragma omp parallel
3  {
4          #pragma omp single nowait
5          {
6                  while(my_pointer) {
7                          #pragma omp task firstprivate(my_pointer)
8                          {
9                                  (void) do_independent_work (my_pointer);
10                         }
11                         my_pointer = my_pointer->next ;
12                 }
13         } // End of single - no implied barrier (nowait)
14 } // End of parallel region - implied barrier
```

Task

- Čekání na potomky (vytvořené tasky)
  `#pragma omp taskwait`
- *Task* má nepatrně vyšší režii než *for*

Task – příklad

```
1  void foo ()
2  {
3      int a, b, c, x, y;
4
5      #pragma omp task shared(a)
6      a = A();
7
8      #pragma omp task if (0) shared (b, c, x)
9      {
10         #pragma omp task shared(b)
11         b = B();
12
13         #pragma omp task shared(c)
14         c = C();
15
16         #pragma omp taskwait
17     }
18     x = f1 (b, c);
19
20     #pragma omp taskwait
21
22     y = f2 (a, x);
23 }
```

Synchronizace

- Kritickým sekcím se nevyhneme ani v OpenMP
  - Závislosti v běhu programu (některé sekce musí být hotové dřív jak jiné)
  - Některé kusy nemohou být prováděny paralelně
- Synchronizační primitiva
  - Critical, Atomic
  - Barrier
  - Single
  - Ordered, Flush

Critical

- Critical
  - Specifikuje sekci v programu, kterou může vykonávat nejvýše jedno vlákno (je jedno které)
  - V podstatě označuje kritickou sekci
  - Syntaxe:
    `#pragma omp ciritcal [jméno]`
  - **`jméno`** je globální identifikátor, kritické sekce stejného jména jsou považovány za identické, tj. žádné bloky stejného jména nepoběží paralelně

Atomic

- Specifikuje sekci v programu, kterou může vykonávat nejvýše jedno vlákno (je jedno které)
- Lehká forma synchronizace, synchronizované jsou pouze čtení a zápisy
- Využívá **`lock`** instrukce na x86/x86_64 architektuře
- Syntaxe:
  `#pragma omp atomic`

```
1 #pragma omp atomic
2     a[indx[i]] += b[i];
```

- Výraz musí být „atomizovatelný“ jinak je ohlášena chyba
- Typicky: **`x++`**, **`x += 2`**
- Jde přeložit: **`*a += *a + 1`** ale nefunguje korektně!

Single, Master

- Podobně jako Critical, Single specifikuje sekci, kterou může provádět pouze jedno vlákno (ale pořád stejné)
- Vhodné pro thread-unsafe sekce, např. I/O
- Syntaxe:
  `#pragma omp single`
- Master je stejné jako Single, sekci provede vždy „master“ vlákno
- Syntaxe:
  `#pragma omp master`

Barrier

- Klasická bariéra, synchronizuje všechna vlákna na bariéře
- Syntaxe:
  `#pragma omp barrier`
- Posloupnost paralelních sekci a bariér musí být stejná pro všechna vlákna
- Příkazy **`single`** a **`master`** nemají implicitní bariéru na vstupu a výstupu!

Ordered, Flush

- Ordered
  - Blok bude vykonán ve stejném pořadí, jako by byl vykonán, kdyby běžel jen v jednom vlákně
  - Může způsobovat serializaci
  - Syntaxe:

    #pragma omp ordered

```
1 #pragma omp parallel for  ordered private(i) shared(n,a)
2    for (i=0; i<n; i++) {
3        a[i] += i;
4 #pragma omp ordered
5      {printf("Thread␣prints␣value␣of␣a[%d]␣=␣%d\n",i,a[i]);}
6    }  /*-- End of parallel for --*/
```

- Flush
  - Zajistí, že všechna vlákna mají konzistentní pohled na objekty v paměti (paměťová bariéra)
  - Syntaxe:

    #pragma omp flush [(seznam)]
  - Seznam udává, které proměnné mají být synchronizovány
  - Neuvedeme-li žádnou proměnnou, jsou synchronizovány všechny viditelné proměnné

Příklad

```
1  #pragma omp parallel shared(a,b,c)
2  {
3          for(i=0; i < N; i++)
4                  a[i] = b[i] + c[i];
5  }
6
7  #pragma omp parallel shared(a,b,d)
8  {
9          for(i=0; i < N; i++)
10                 d[i] = a[i] + b[i];
11 }
```

Nemusí dát časem správný výsledek

Příklad

```
1  #pragma omp parallel shared(a,b,c)
2  {
3          for(i=0; i < N; i++)
4                  a[i] = b[i] + c[i];
5  }
6
7  #pragma omp barrier
8
9  #pragma omp parallel shared(a,b,d)
10 {
11         for(i=0; i < N; i++)
12                 d[i] = a[i] + b[i];
13 }
```

Příklad

```
1  #include <stdio.h>
2  #include <omp.h>
3  int main()
4  {
5     int n = 9, i, a, b[n];
6     for (i=0; i<n; i++)
7         b[i] = -1;
8  #pragma omp parallel shared(a,b) private(i)
9  {
10    #pragma omp single
11    {
12       a = 10;
13    }
14    #pragma omp barrier
15    #pragma omp for
16    for (i=0; i<n; i++)
17        b[i] = a;
18
19 } /*-- End of parallel region --*/
20    printf("After the parallel region:\n");
21    for (i=0; i<n; i++)
22        printf("b[%d] = %d\n",i,b[i]);
23    return(0);
24 }
```

Příklad

```
1  #include <stdio.h>
2  #include <omp.h>
3  int main()
4  {
5     int i, n = 25, sumLocal;
6     int sum = 0, a[n];
7     int ref =(n-1)*n/2;
8     for (i=0; i<n; i++)
9         a[i] = i;
10    #pragma omp parallel default(none) shared(n,a,sum) \
11            private(sumLocal)
12    {
13       sumLocal = 0;
14       #pragma omp for
15         for (i=0; i<n; i++)
16           sumLocal += a[i];
17       #pragma omp critical (update_sum)
18       {
19         sum += sumLocal;
20         printf("sumLocal = %d sum = %d\n",sumLocal,sum);
21       }
22    } /*-- End of parallel region --*/
23    printf("Value of sum after parallel region: %d\n",sum);
24    printf("Check results: sum = %d (should be %d)\n",sum,ref);
25    return(0);
26 }
```

Reduction

- Redukuje seznam proměnných do jedné za použití konkrétního operátoru
- Syntaxe:
  `#pragma omp reduction (op : list)`
- **`list`** je seznam proměnných a **`op`** je jeden z následujících
  - $+$, $-$, $*$, &, $\widehat{\ }$, $|$, &&, $||$

Reduction – příklad

```
1  #include <stdio.h>
2  #include <omp.h>
3  int main()
4  {
5     int i, n = 25;
6     int sum = 0, a[n];
7     int ref = (n-1)*n/2;
8     for (i=0; i<n; i++)
9         a[i] = i;
10    printf("Value of sum prior to parallel region: %d\n",sum);
11    #pragma omp parallel for default(none) shared(n,a) reduction(+:sum)
12         for (i=0; i<n; i++)
13           sum += a[i];
14    /*-- End of parallel reduction --*/
15
16    printf("Value of sum after parallel region: %d\n",sum);
17    printf("Check results: sum = %d (should be %d)\n",sum,ref);
18
19    return(0);
20 }
```

Klauzule

- Základní formát
  `#pragma omp jméno-příkazu [klauzule] nový_řádek`
- if, private, shared
- firstprivate, lastprivate
- default
- nowait
- copyin, copyprivate
- num_threads
- ordered
- schedule
- collapse (Open MP 3.0+)

if, private, shared

- **`if (výraz)`**
  - omp příkaz bude proveden paralelně, pokud je **`výraz`** vyhodnocen jako pravdivý, jinak je blok proveden sekvenčně
- **`private(list)`**
  - Úložiště objektu není asociováno s původní lokací
  - Všechny reference jsou k lokálnímu objektu
  - Nemá definovanou hodnotu při vstupu a výstupu
- **`shared(list)`**
  - Data jsou přístupná ze všech vláken v týmu
  - Všechna data jsou pro vlákna na stejných lokacích
  - Přístup k datům není synchronizován!

```
1 #pragma omp parallel if (n > threshold) shared(n,x,y) private(i)
2 {
3 #pragma omp for
4         for (i=0; i<n; i++)
5                 x[i] += y[i];
6 }
```

  - Proč je příklad korektní?

firstprivate, lastprivate

- **`firstprivate(seznam)`**
  - Proměnné v **`seznamu`** jsou inicializovány na hodnotu, kterou měl objekt před zahájením paralelní sekce
- **`lastprivate(seznam)`**
  - Vlákno vykonávající poslední iteraci smyčky nebo poslední sekci zapíše obsah proměnných v **`seznamu`** do původního objektu

```
1   int n, C, B, A = 10;
2  #pragma omp parallel
3  {
4    #pragma omp for private(i) firstprivate(A) lastprivate(B)...
5    for (i=0; i<n; i++)
6    {
7        /*-- A undefined, unless declared first private */
8        B = A + i;
9    }
10   /*-- B undefined, unless declared lastprivate */
11   C = B;
12 }
```

default

- **`default(none | shared)`**
- **`none`**
  - Žádné výchozí nastavení
  - Všechny proměnné je potřeba explicitně určit jako **`private`** nebo **`shared`**
- **`shared`**
  - Všechny proměnné jsou implicitně **`shared`**
  - Výchozí stav, pokud není přítomna **`default`** klauzule
- Fortran podporuje navíc: **`default(private | firstprivate)`**

nowait

- Za účelem minimalizace synchronizace, některé Open MP příkazy podporují volitelnou **`nowait`** klauzuli
- Pokud je přítomna, vlákna nejsou synchronizována (nečekají) na konci paralelního bloku

copyin, copyprivate

- **`threadprivate(seznam)`**
  - Obdoba **`private(seznam)`**, jde o globální proměnnou pro každé vlákno zvlášť
  - Oproti **`private(seznam)`** je její obsah zachován v rámci všech paralelních bloků
  - Využívá TLS (podobně jako klíčové slovo **`__thread`** v C)
- **`copyin(seznam)`**
  - Aplikuje se na **`threadprivate`** proměnné
  - **`threadprivate`** proměnná je inicializována na hodnotu globální proměnné
  - **`threadprivate`** proměnná má vždy hodnotu zadanou při inicializaci
  - **`copyin`** zařídí promítnutí změn v průběhu vykonávání programu
- **`copyprivate(seznam)`**
  - Vztahuje se na **`single`** blok
  - Po jeho skončení jsou hodnoty proměnných v **`seznamu`** rozkopírovány do ostatních vláken
  - Aplikovatelné na **`threadprivate`** proměnné

Příklad

```
1  #include <stdio.h>
2  #include <omp.h>
3  float x=1, y=1, z=1, v=1, fGlobal = 1.0;
4  #pragma omp threadprivate(x, y, z, v)
5  float get_float() {
6     return (fGlobal += 0.001);
7  }
8  void CopyPrivate() {
9     #pragma omp single copyprivate(x, z)
10    {
11       v += get_float();
12       x += get_float();
13       y += get_float();
14       z += get_float();
15    }
16    printf("Value v=%f, thread = %d\n", v, omp_get_thread_num());
17    printf("Value x=%f, thread = %d\n", x, omp_get_thread_num());
18    printf("Value y=%f, thread = %d\n", y, omp_get_thread_num());
19    printf("Value z=%f, thread = %d\n", z, omp_get_thread_num());
20 }
21 void main() {
22    printf("Sequential:\n");
23    CopyPrivate();
24    printf("Parallel:\n");
25    #pragma omp parallel copyin(x, y)
26    {
27       CopyPrivate();
28    }
```

Výstup příkladu

```
Sequential:
Value v=2.001000, thread = 0
Value x=2.002000, thread = 0
Value y=2.003000, thread = 0
Value z=2.004000, thread = 0
Parallel:
(no copyin or copyprivate) += 1.005
Value v=3.006000, thread = 0
Value v=1.000000, thread = 1
Value v=1.000000, thread = 2
(copyin & copyprivate) += 1.006
Value x=3.008000, thread = 0
Value x=3.008000, thread = 1
Value x=3.008000, thread = 2
(copyin) += 1.007
Value y=3.010001, thread = 0
Value y=2.003000, thread = 1
Value y=2.003000, thread = 2
(copyprivate) += 1.008
Value z=2.008000, thread = 0
Value z=2.008000, thread = 1
Value z=2.008000, thread = 2
```

num_threads, ordered

- **`num_treads(int)`**
  - Nastavuje počet vláken, které mají vykonávat paralelní blok
  - V případě statického plánování je to přesný počet vláken
  - V případě dynamického plánování je to maximální počet vláken
- **`ordered`**
  - Tato klauzule je povinná pro **`for`** příkaz, je-li tento příkaz svázán s **`ordered`** příkazem probíraným na slídě 23

schedule

- **`schedule(způsob, velikost_kusu)`**
- Způsob: **`static`**, **`dynamic`**, **`guided`**, **`runtime`**
- **`schedule(static,`** velikost_kusu**`)`**
  - Iterace cyklu jsou rozděleny do skupiny vláken
  - Každé vlákno vykonává **`velikost_kusu`** iterací
  - Není-li **`velikost_kusu`** zadáno, je zvolen (počet iterací) / (počet vláken)
- **`schedule(dynamic,`** velikost_kusu**`)`**
  - Iterace jsou rozděleny do skupin o velikosti **`velikost_kusu`** a každá skupina je přiřazena volnému vláknu
  - Implicitní velikost **`velikost_kusu`** je 1
- **`schedule(guided,`** velikost_kusu**`)`**
  - Vlákna dostanou kusy o velikosti (počet iterací) / (počet vláken)
  - Velikost kusu se exponenciálně snižuje k **`velikost_kusu`** nebo 1
- **`schedule(runtime)`**
  - Způsob plánování se stanoví až při běhu aplikace.

collapse

- Kolaps smyček
- Mějme kus kódu:

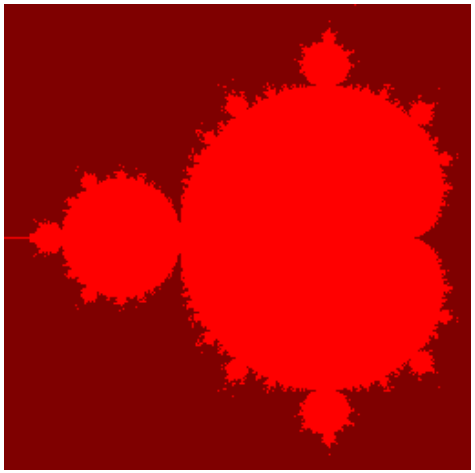
```
1 int y; ...
2 #pragma omp parallel for
3 for (y = 0; y < h; ++y) {
4         int x; ...
5         for (x = 0; x < w; ++x) { ... }
6 }
```

- Paralelně běží vždy celé řádky
- Kolaps nabízí jemnější granularitu paralelizace
- **`collapse(číslo)`** sloučí **`číslo`** smyček do jedné a tu paralelizuje

```
1 int x, y; ...
2 #pragma omp parallel for collapse(2)
3 for (y = 0; y < h; ++y)
4         for (x = 0; x < w; ++x) { ... }
```

Sequential performance:

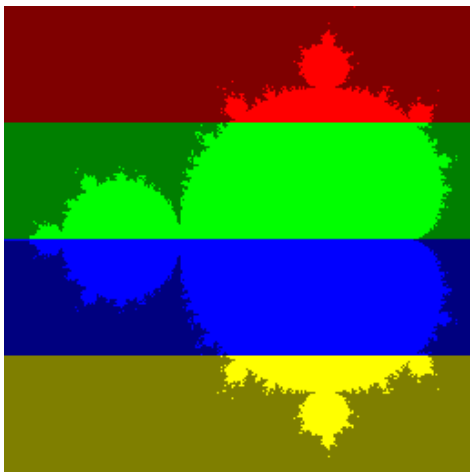
4000 pixels: 2.59 seconds, 99% CPU



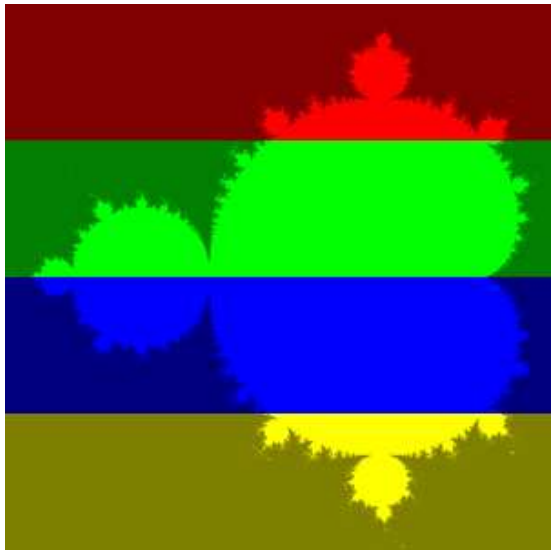
Measurements use a Xeon 4-CPU machine, gcc 4.1.2, and Linux 2.6.23

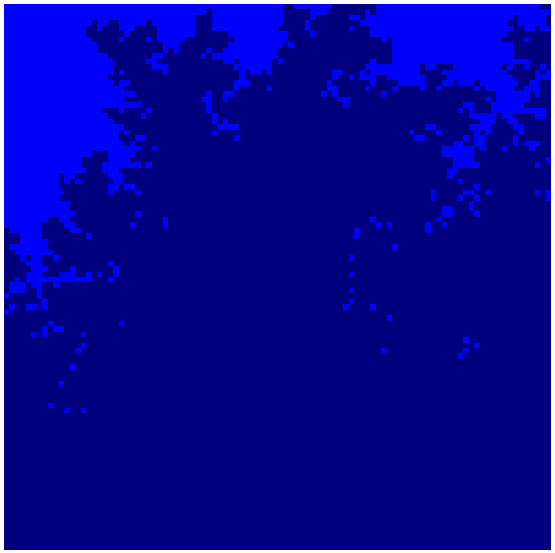
4 processors, `schedule(auto)`

01.04 seconds, 239% CPU



256 pixels

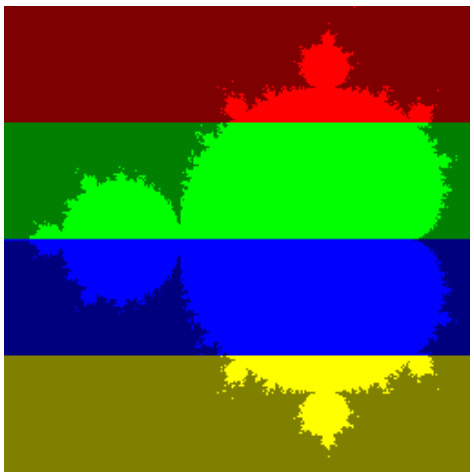




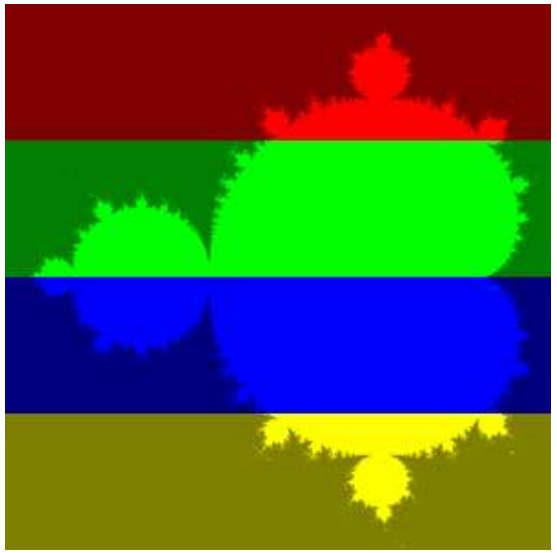
2000 pixels

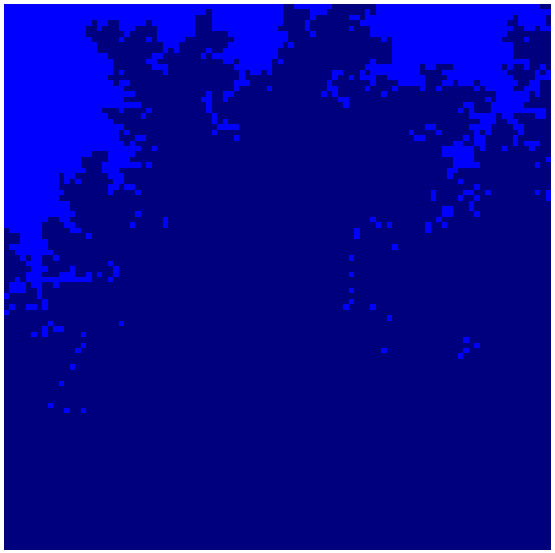
4 processors, `schedule(static)`

00.93 seconds, 267% CPU



256 pixels

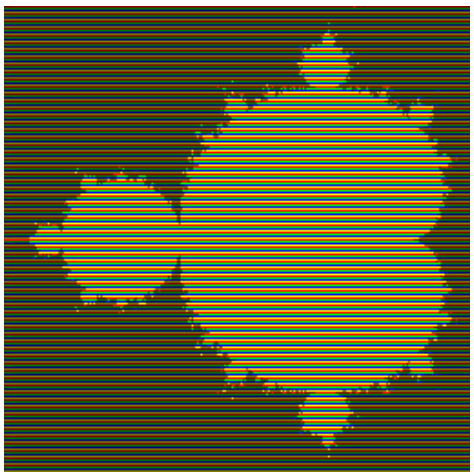




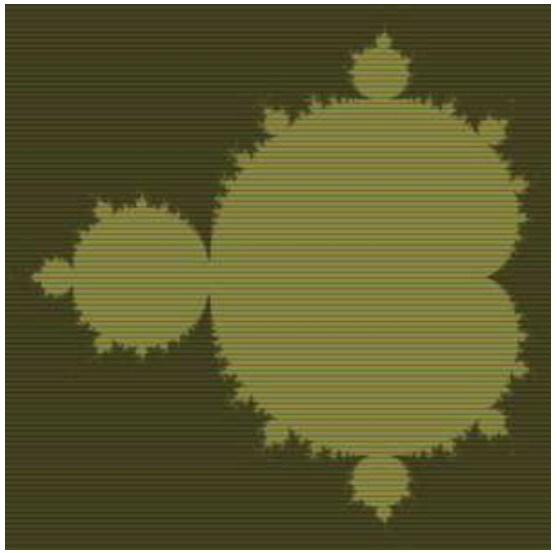
2000 pixels

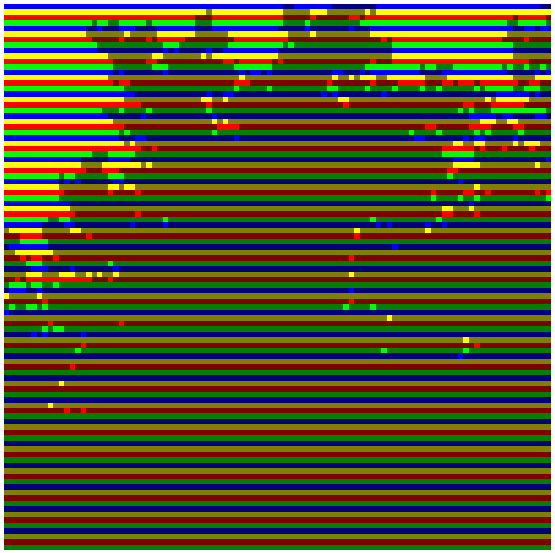
4 processors, `schedule(static, 1)`

00.86 seconds, 289% CPU



256 pixels

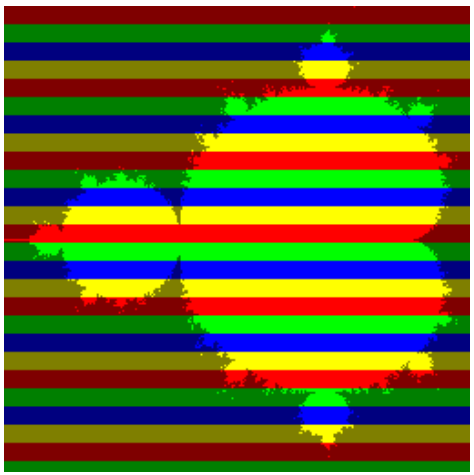




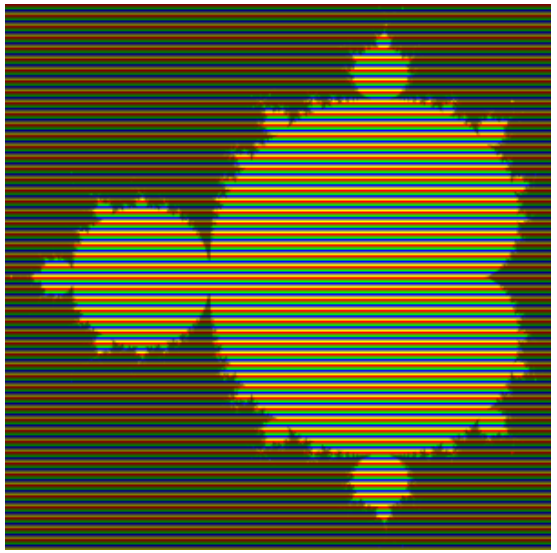
2000 pixels

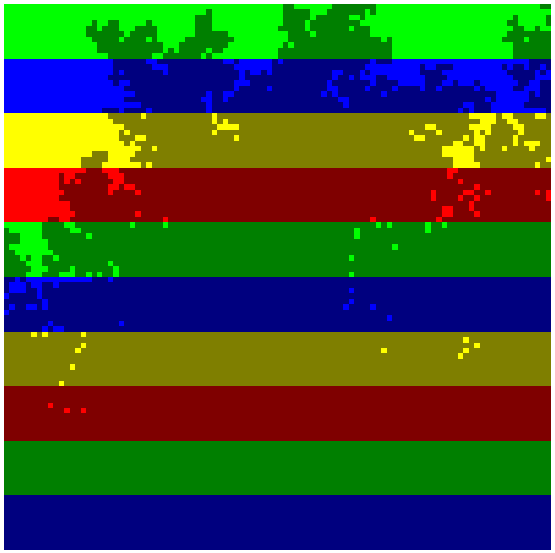
4 processors, `schedule(static, 10)`

00.85 seconds, 294% CPU



256 pixels

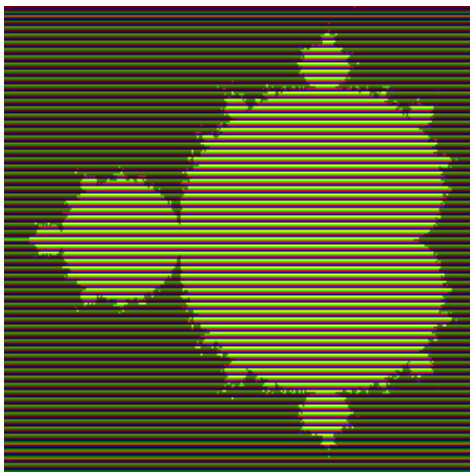




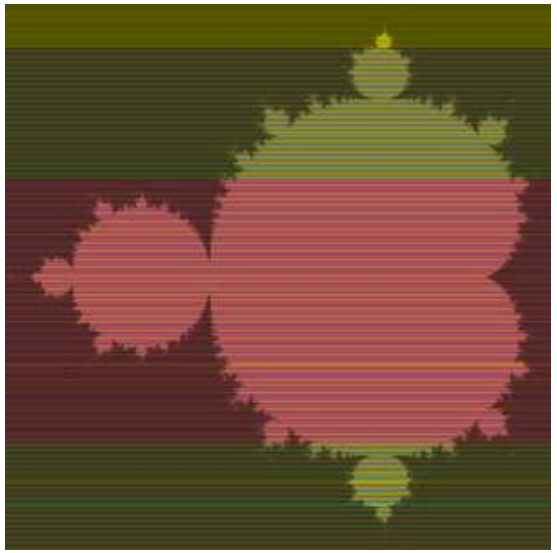
2000 pixels

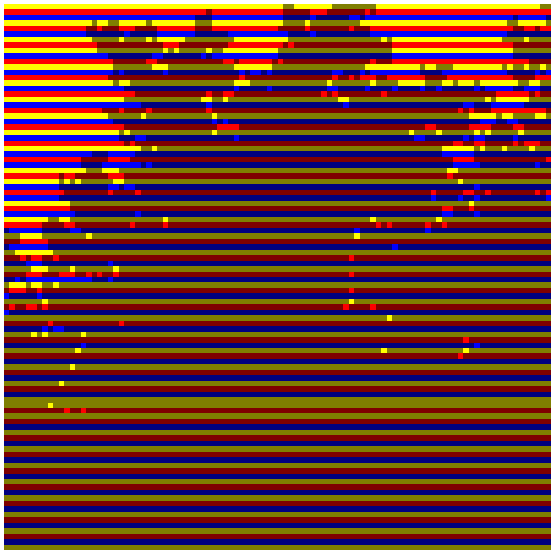
4 processors, `schedule(dynamic)`

00.72 seconds, 350% CPU



256 pixels

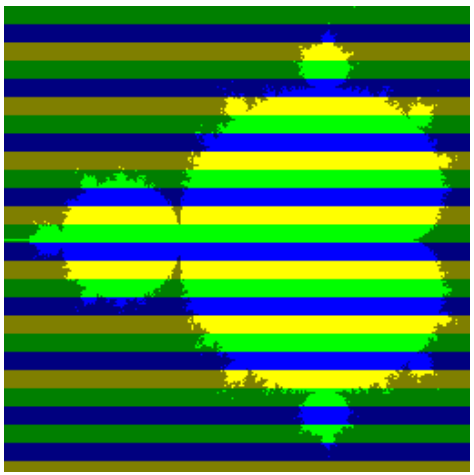




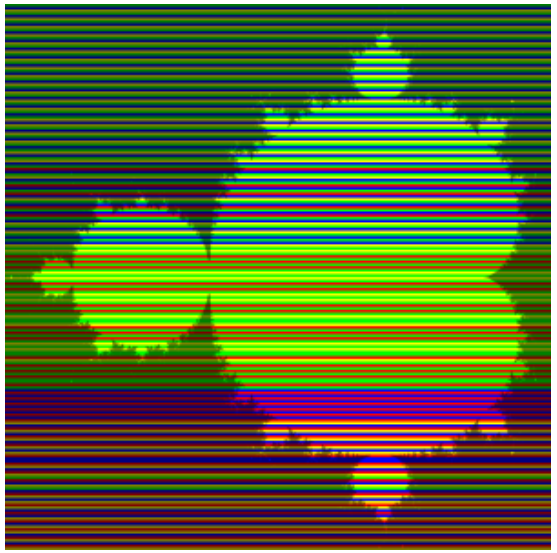
2000 pixels

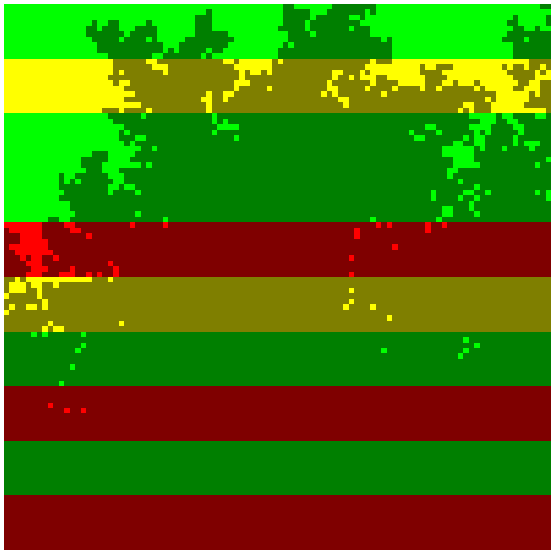
4 processors, `schedule(dynamic, 10)`

00.66 seconds, 381% CPU



256 pixels

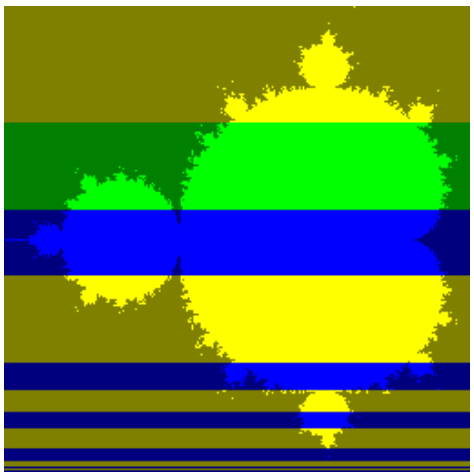




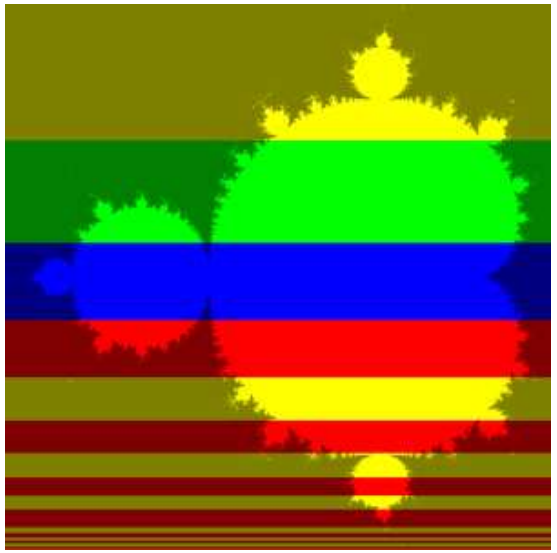
2000 pixels

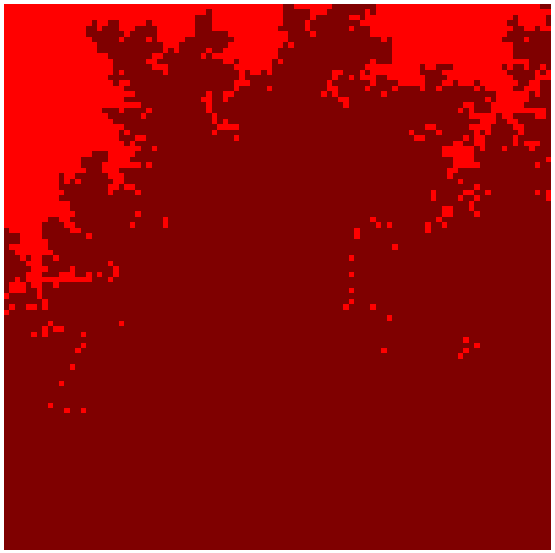
4 processors, `schedule(guided)`

00.83 seconds, 302% CPU



256 pixels

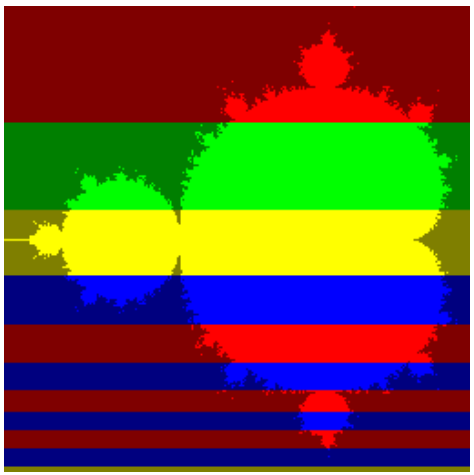




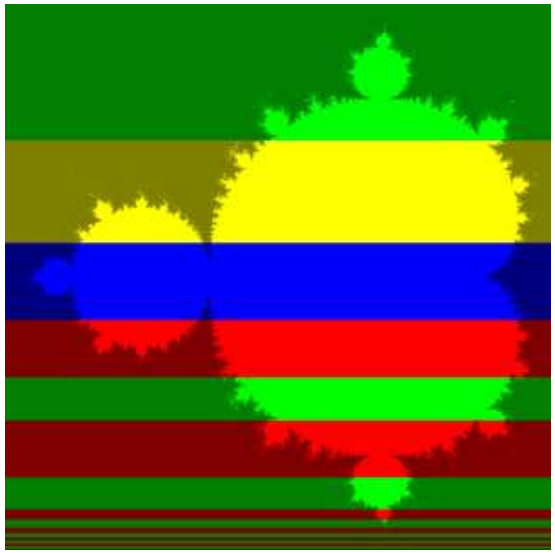
2000 pixels

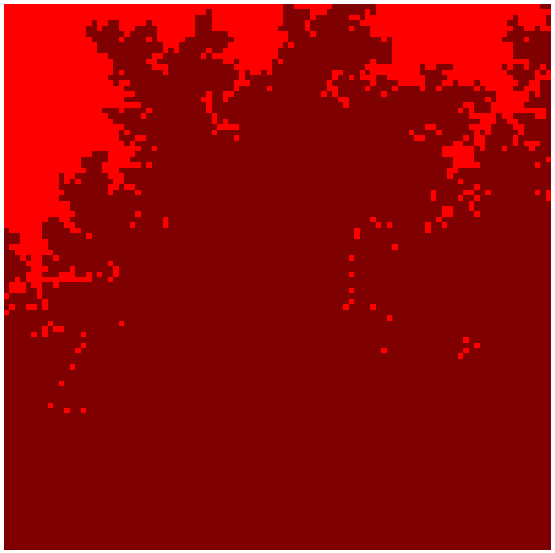
4 processors, `schedule(guided, 10)`

00.78 seconds, 318% CPU



256 pixels





2000 pixels

Funkce knihovny Open MP

- Řízení prostředí
- Zamykání
- Časové funkce

Řízení prostředí I

- **`void omp_set_num_threads(int)`** explicitně nastaví počet vláken pro paralelní sekce
- **`int omp_get_num_threads(void)`** vrátí počet vláken v týmu
- **`int omp_get_max_threads(void)`** vrátí maximální počet vláken
- **`int omp_get_thread_num(void)`** vrátí identifikaci vlákna (0 – n)
- **`int omp_get_num_procs(void)`** vrátí maximální počet procesorů, které jsou aplikaci k dispozici
- **`int omp_in_parallel(void)`** test, zda jsme v paralelním bloku
- **`void omp_set_dynamic(int)`** nastaví dynamické plánování vláken (implementace může nastavení ignorovat)
- **`int omp_get_dynamic(void)`** test, zda je nastaveno dynamické plánováni
- **`void omp_set_nested(int)`** povolí vnořený paralelismus
- **`int omp_get_nested (void)`** test, zda je zapnutý vnořený paralelismus

Řízení prostředí II

- **`void omp_set_schedule(omp_sched_t, int)`** nastavení plánování
- **`void omp_get_schedule(omp_sched_t *, int *)`** dotaz na nastavení plánování
- **`int omp_get_thread_limit(void)`** vrací max. počet vláken pro program
- **`void omp_set_max_active_levels(int)`** nastaví max. počet aktivních (tj. souběžných) paralelních sekcí
- **`int omp_get_max_active_levels(void)`** vrátí max. počet aktivních paralelních sekcí
- **`int omp_get_level(void)`** vrátí max. počet vnořených paralelních sekcí
- **`int omp_get_active_level(void)`** vrátí max. počet aktivních vnořených paralelních sekcí
- **`int omp_get_ancestor_thread_num(int)`** vrátí identifikaci vlákna předchůdce
- **`int omp_get_team_size(int)`** vrátí počet vláken v týmu (v paralelní sekci)
- **`omp_sched_t`**: **`omp_sched_static`**, **`omp_sched_dynamic`**, **`omp_sched_guided`**, **`omp_sched_auto`**

Zamykání

- Dva druhy zámků
- **`omp_lock_t`** – jednoduchý zámek
  - Jednoduché zámky nemohou být zamčeny, jsou-li již zamčeny
- **`omp_nest_lock_t`** – vnořený zámek
  - Vnořené zámky mohou být zamčeny jedním vláknem několikrát

Zamykání

- **`void omp_init_lock(omp_lock_t *)`** inicializace zámku
- **`void omp_destroy_lock(omp_lock_t *)`** zrušení zámku
- **`void omp_set_lock(omp_lock_t *)`** zamčení zámku
- **`void omp_unset_lock(omp_lock_t *)`** odemčení zámku
- **`int omp_test_lock(omp_lock_t *)`** zamčení zámku, je-li to možné (varianta trylock)
- **`void omp_init_nest_lock(omp_nest_lock_t *)`** inicializace zámku
- **`void omp_destroy_nest_lock(omp_nest_lock_t *)`** zrušení zámku
- **`void omp_set_nest_lock(omp_nest_lock_t *)`** zamčení zámku
- **`void omp_unset_nest_lock(omp_nest_lock_t *)`** odemčení zámku
- **`int omp_test_nest_lock(omp_nest_lock_t *)`** zamčení zámku, je-li to možné (varianta trylock)

Časové funkce

- **`double omp_get_wtime(void)`** vrací počet vteřin od startu systému, není úplně konzistentní mezi všemi vlákny
- **`double omp_get_wtick(void)`** vrací délku trvání jednoho tiku hodin – míra přesnosti časovače (na Linuxu s gcc 4.4.3 vrací 0.0)

Proměnné prostředí

- **`OMP_NUM_THREADS n`**
- **`OMP_SCHEDULE "schedule,[chunk]`**
- **`OMP_DYNAMIC {TRUE | FALSE}`**
- **`OMP_NESTED {TRUE | FALSE}`**
- **`OMP_STACKSIZE size [B|K|M|G]`**
- **`OMP_WAIT_POLICY [ACTIVE|PASSIVE]`** – aktivní čekání pomocí busy loop
- **`OMP_MAX_ACTIVE_LEVELS n`**
- **`OMP_THREAD_LIMIT n`**