OpenMP + NUMA

CSE 6230: HPC Tools & Apps
Fall 2014 — September 5

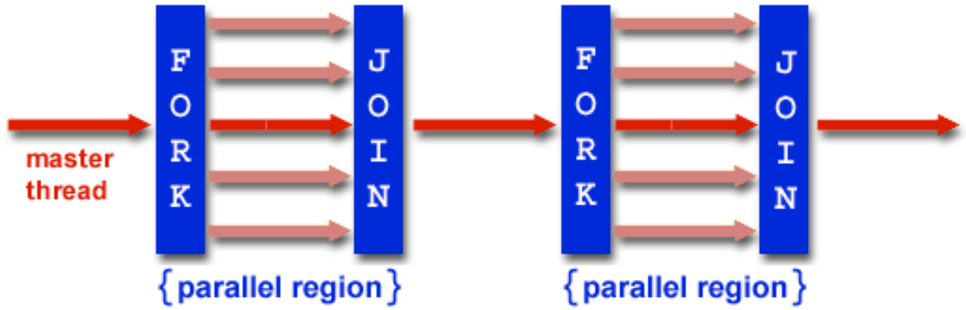
Based in part on the **LLNL tutorial** @ https://computing.llnl.gov/tutorials/openMP/
See also the textbook, Chapters 6 & 7

Georgia Tech College of Computing
Computational Science and Engineering

hpcgarage

OpenMP

- Programmer identifies **serial** and **parallel regions**, not threads



- Library + directives (requires compiler support)
  - Official website: http://www.openmp.org
  - Also: https://computing.llnl.gov/tutorials/openMP/

Simple example

```
int main ()
{

   printf (“hello, world!\n”); // Execute in parallel

   return 0;
}
```

Simple example

```
#include <omp.h>

int main ()
{
  omp_set_num_threads (16); // OPTIONAL — Can also use
                            // OMP_NUM_THREADS environment variable

  #pragma omp parallel
  {
    printf (“hello, world!\n”); // Execute in parallel
  } // Implicit barrier/join
  return 0;
}
```

Simple example

```
#include <omp.h>

int main ()
{
  omp_set_num_threads (16); // OPTIONAL — Can also use
                            // OMP_NUM_THREADS environment variable

  #pragma omp parallel num_threads(8) // Restrict team size locally
  {
    printf (“hello, world!\n”); // Execute in parallel
  } // Implicit barrier/join
  return 0;
}
```

Simple example

```
#include <omp.h>

int main ()
{
  omp_set_num_threads (16); // OPTIONAL — Can also use
                            // OMP_NUM_THREADS environment variable

  #pragma omp parallel
  {
    printf ("hello, world!\n"); // Execute in parallel
  } // Implicit barrier/join
  return 0;
}
```

Compiling:

gcc -fopenmp …

icc -openmp …

Simple example

```
#include <omp.h>

int main ()
{
  omp_set_num_threads (16); // OPTIONAL — Can also use
                            // OMP_NUM_THREADS environment variable

  #pragma omp parallel
  {
    printf (“hello, world!\n”); // Execute in parallel
  } // Implicit barrier/join
  return 0;
}
```

Output:

hello, world!
hello, world!
hello, world!
…

Parallel loops

```
for (i = 0; i < n; ++i) {
  a[i] += foo (i);
}
```

Parallel loops

```
for (i = 0; i < n; ++i) {
  a[i] += foo (i);
}
```

```
#pragma omp parallel  // Activates the team of threads
{
  #pragma omp for shared (a,n) private (i)  // Declares work sharing loop
  for (i = 0; i < n; ++i) {
    a[i] += foo (i);
  } // Implicit barrier/join
} // Implicit barrier/join
```

Parallel loops

```
for (i = 0; i < n; ++i) {
  a[i] += foo (i);
}
```

```
#pragma omp parallel
{
  foo (a, n);
} // Implicit barrier/join
```

```
void foo (item* a, int n) {
  int i;
  #pragma omp for shared (a,n) private (i)
  for (i = 0; i < n; ++i) {
    a[i] += foo (i);
  } // Implicit barrier/join
}
```

Note: if foo() is called *outside* a parallel region, it is *orphaned.*

Parallel loops

```
for (i = 0; i < n; ++i) {
  a[i] += foo (i);
}
```

```
#pragma omp parallel for default (none) shared (a,n) private (i)
for (i = 0; i < n; ++i) {
  a[i] += foo (i);
} // Implicit barrier/join
```

Combining `omp parallel` and `omp for` is just a convenient shorthand for a common idiom.

“If” clause

```
for (i = 0; i < n; ++i) {
  a[i] += foo (i);
}
```

```
const int B = …;
#pragma omp parallel for if (n>B) default (none) shared (a,n) private (i)
for (i = 0; i < n; ++i) {
  a[i] += foo (i);
} // Implicit barrier/join
```

Parallel loops

- **You** must check dependencies

```
s = 0;
for (i = 0; i < n; ++i)
  s += x[i];
```

Parallel loops

- **You** must check dependencies

```
s = 0;
for (i = 0; i < n; ++i)
  s += x[i];
```

```
#pragma omp parallel for shared(s)
for (i = 0; i < n; ++i)
  s += x[i]; // Data race!
```

Parallel loops

- **You** must check dependencies

```
s = 0;
for (i = 0; i < n; ++i)
  s += x[i];
```

```
#pragma omp parallel for shared(s)
for (i = 0; i < n; ++i)
  #pragma omp critical
  s += x[i];
```

```
#pragma omp parallel for reduction (+:s)
for (i = 0; i < n; ++i)
  s += x[i];
```

Removing implicit barriers: nowait

```
#pragma omp parallel default (none) shared (a,b,n) private (i)
{
  #pragma omp for nowait
  for (i = 0; i < n; ++i)
    a[i] = foo (i);

  #pragma omp for nowait
  for (i = 0; i < n; ++i)
    b[i] = bar (i);
}
```

Contrast with `_Cilk_for`, which does not have such a “feature.”

Single thread

```
#pragma omp parallel default (none) shared (a,b,n) private (i)
{
  #pragma omp single [nowait]
  for (i = 0; i < n; ++i) {
    a[i] = foo (i);
  } // Implied barrier unless “nowait” specified

  #pragma omp for
  for (i = 0; i < n; ++i)
    b[i] = bar (i);
}
```

Only one thread from the team will execute the first loop. Use `single` with `nowait` to allow other threads to proceed while the one thread executes the first loop.

Master thread

```
#pragma omp parallel default (none) shared (a,b,n) private (i)
{
  #pragma omp master
  for (i = 0; i < n; ++i) {
    a[i] = foo (i);
  } // No implied barrier

  #pragma omp for
  for (i = 0; i < n; ++i)
    b[i] = bar (i);
}
```

Synchronization primitives

Critical sections	No explicit locks	**#pragma omp critical** { … }
Barriers		**#pragma omp barrier**
Explicit locks	May require flushing	**omp_set_lock** (l); … **omp_unset_lock** (l);
Single-thread regions	Inside parallel regions	**#pragma omp single** { /* executed once */ }

Loop scheduling

- **Static**: ***k*** iterations per thread, assigned statically

```
#pragma omp parallel for schedule static(k) …
```

- **Dynamic**: ***k*** iters / thread, using logical work queue

```
#pragma omp parallel for schedule dynamic(k) …
```

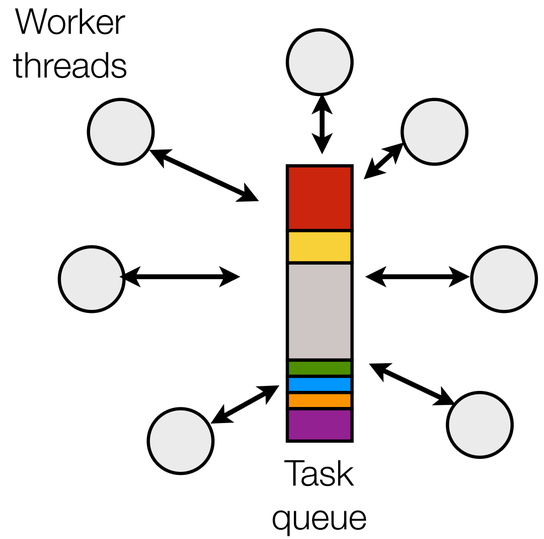
- **Guided**: ***k*** iters / thread initially, reduced with each allocation

```
#pragma omp parallel for schedule guided(k) …
```

- **Run-time (schedule runtime)**: Use value of environment variable, **OMP_SCHEDULE**
- What are all these scheduling things?

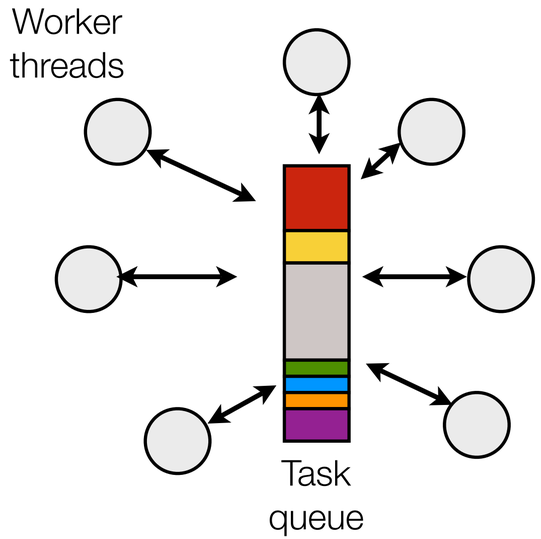
Loop scheduling strategies for load balance

- Centralized scheduling (task queue)
  - Dynamic, on-line approach
  - Good for small no. of workers
  - Independent tasks, known
- For loops: **Self-scheduling**
  - Task = subset of iterations
  - Loop body has unpredictable time
  - Tang & Yew (ICPP ’86)



Self-scheduling trade-off

- Unit of work to grab: balance vs. contention
- Some variations:
  - Grab fixed size chunk
  - Guided self-scheduling
  - Tapering
  - Weighted factoring, adaptive factoring, distributed trapezoid
  - Self-adapting, gap-aware, …



Work queue

23 

Work queue

23 

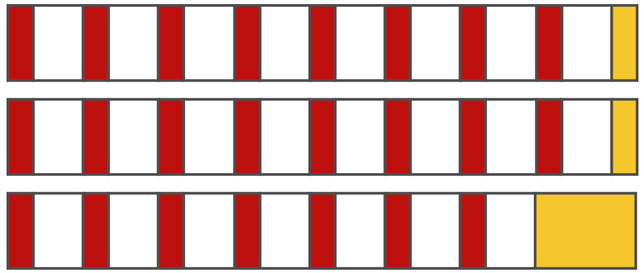
For P=3 procs,

Ideal: 23 / 3 ~ 7.67

Work queue

23

Fixed k=1



12.5

For P=3 procs,

Ideal: 23 / 3 ~ 7.67

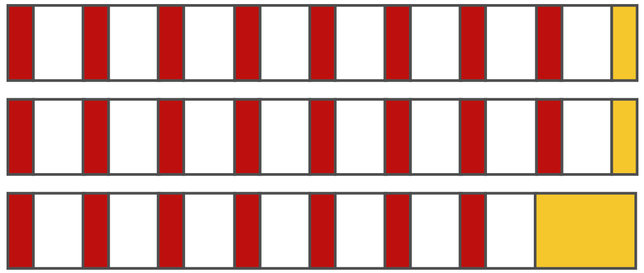
Work queue

23

For P=3 procs,

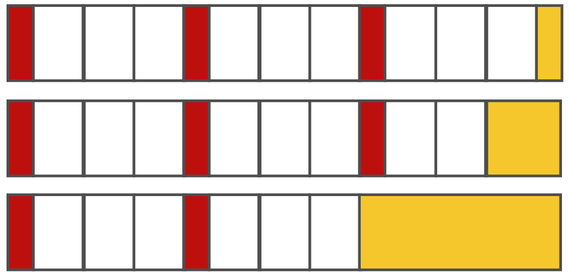
Ideal: 23 / 3 ~ 7.67

Fixed k=1



12.5

Fixed k=3



11

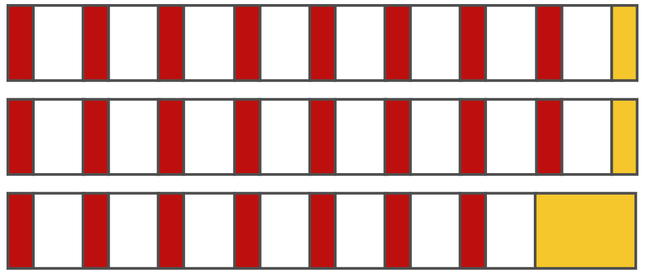
Work queue

23

For P=3 procs,

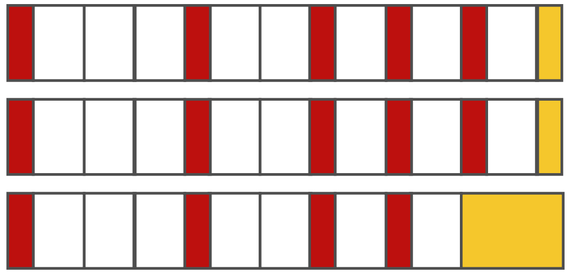
Ideal: 23 / 3 ~ 7.67

Fixed k=1



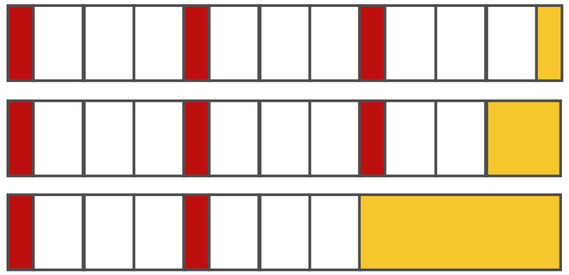
12.5

Tapered, k_0=3



11

Fixed k=3



11

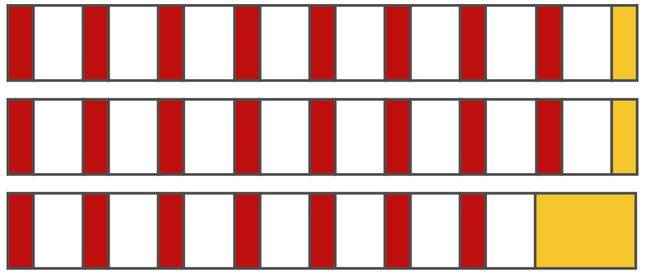
Work queue

23

For P=3 procs,

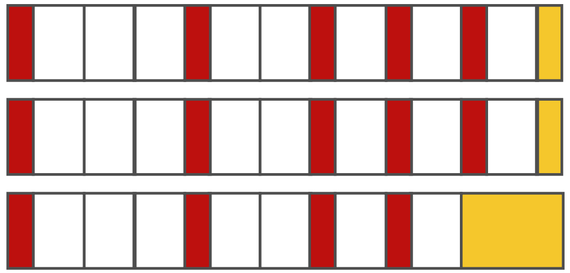
Ideal: 23 / 3 ~ 7.67

Fixed k=1



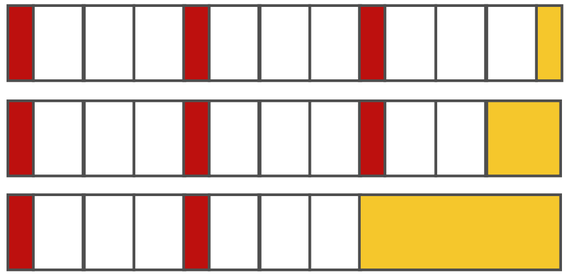
12.5

Tapered, k_0=3



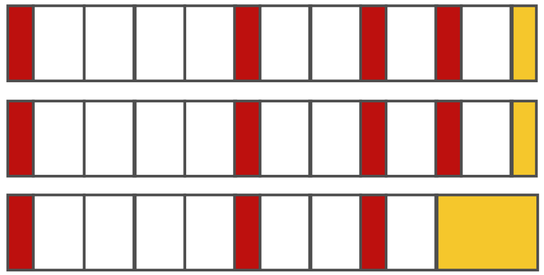
11

Fixed k=3



11

Tapered, k_0=4



10.5

Summary: Loop scheduling

- **Static**: ***k*** iterations per thread, assigned statically

```
#pragma omp parallel for schedule static(k) …
```

- **Dynamic**: ***k*** iters / thread, using logical work queue

```
#pragma omp parallel for schedule dynamic(k) …
```

- **Guided**: ***k*** iters / thread initially, reduced with each allocation

```
#pragma omp parallel for schedule guided(k) …
```

- **Run-time**: Use value of environment variable, **OMP_SCHEDULE**

Tasking (OpenMP 3.0+)

```
int fib (int n) {
  // G == tuning parameter
  if (n <= G) fib__seq (n);
  int f1, f2;
  f1 = _Cilk_spawn fib (n-1);
  f2 = fib (n-2);
  _Cilk_sync;
  return f1 + f2;
}
```


See also: https://iwomp.zih.tu-dresden.de/downloads/omp30-tasks.pdf

Tasking (OpenMP 3.0+)

```
int
   // G == tuning parameter


   f1 =
   f2 = fib (n-2);


}
```

```
int fib (int n) {
   if (n <= G) fib__seq (n);
   int f1, f2;
#pragma omp task default(none) shared(n,f1)
   f1 = fib (n-1);
   f2 = fib (n-2);
#pragma omp taskwait
   return f1 + f2;
}
```


See also: https://iwomp.zih.tu-dresden.de/downloads/omp30-tasks.pdf

Tasking (OpenMP 3.0+)

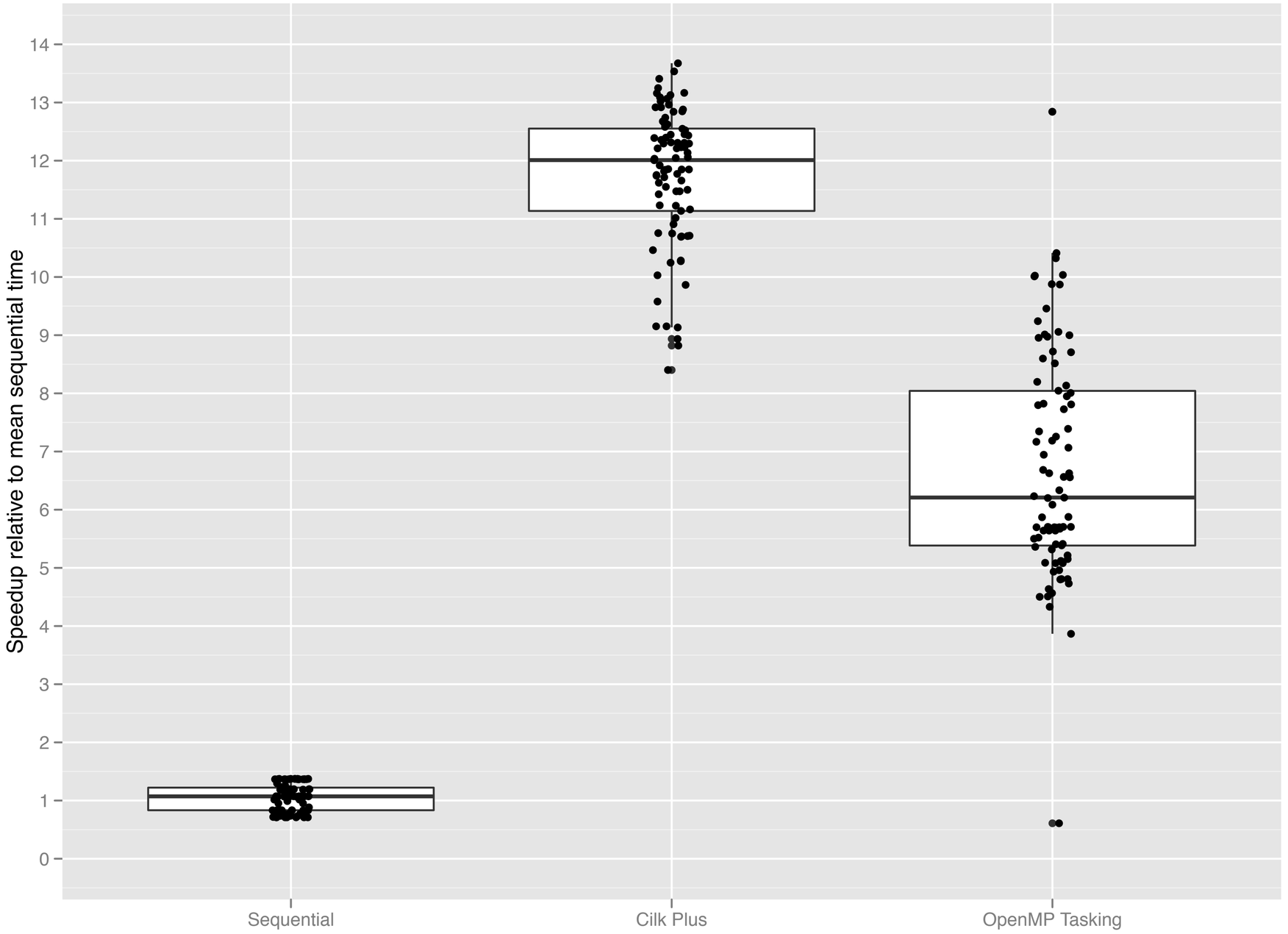
```
int
   // G == tuning parameter


   f1 =
   f2 = fib (n-2);


}
```

```
int fib (int n) {
   if (n <= G) fib__seq (n);
   int f1, f2;
#pragma omp task default(none) shared(n,f1)
   f1 = fib (n-1);
   f2 = fib (n-2);
#pragma omp taskwait
   return f1 + f2;
}
```

```
// At the call site:
#pragma omp parallel
#pragma omp single nowait
  answer = fib (n);
```


See also: https://iwomp.zih.tu-dresden.de/downloads/omp30-tasks.pdf

```
// At the call site:
#pragma omp parallel
#pragma omp single nowait
  answer = fib (n);

  // …

int fib (int n) {
   if (n <= G) fib__seq (n);
   int f1, f2;
#pragma omp task …
   f1 = fib (n-1);
   f2 = fib (n-2);
#pragma omp taskwait
   return f1 + f2;
}
```



Note: Parallel run-times are highly variable! For your assignments and projects, you should gather and report suitable statistics.

Tasking (OpenMP 3.0+)

```
// Computes: f2 (A (), f1 (B (), C ()))
int a, b, c, x, y;

a = A();
b = B();
c = C();
x = f1(b, c);
y = f2(a, x);
```

```
// Assume a parallel region
#pragma omp task shared(a)
a = A();
#pragma omp task if (0) shared (b, c, x)
{
  #pragma omp task shared(b)
  b = B();
  #pragma omp task shared(c)
  c = C();
  #pragma omp taskwait
}
x = f1 (b, c);
#pragma omp taskwait
y = f2 (a, x);
```


See also: https://iwomp.zih.tu-dresden.de/downloads/omp30-tasks.pdf

Check: What does this code do?

```
// At some call site:
#pragma omp parallel
#pragma omp single nowait
foo ();
```

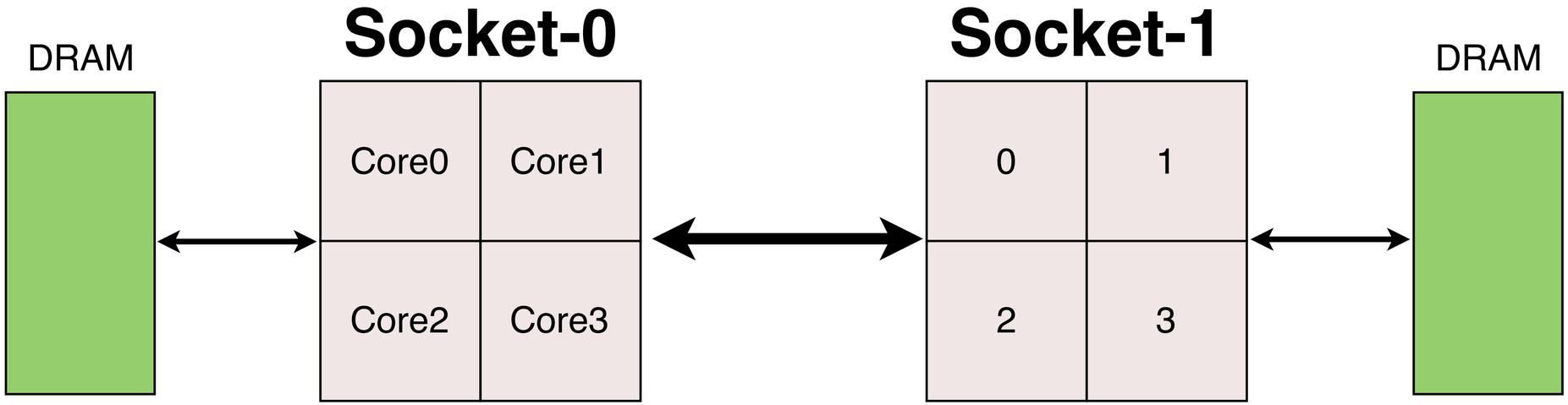
```
foo ()
{
#pragma omp task
  bar ();
  baz ();
}

bar ()
{
#pragma omp for
  for (int i = 1; i < 100; ++i)
    boo ();
}
```


See also: https://iwomp.zih.tu-dresden.de/downloads/omp30-tasks.pdf

Performance tuning tip: Controlling the number of threads

```
#pragma omp parallel num_threads(4)
{
  #pragma omp for default (none) shared (a,n) private (i)
  for (i = 0; i < n; ++i) {
    a[i] += foo (i);
  }
}
```

Performance tuning tip:
Exploit non-uniform memory access (NUMA)



Example: Two quad-core CPUs with logically shared but physically distributed memory

```
--------------------------------------------------------------
CPU type:       Intel Core Westmere processor
**************************************************************
Hardware Thread Topology
**************************************************************
Sockets:        2
Cores per socket:       6
Threads per core:       2
--------------------------------------------------------------
HWThread        Thread          Core            Socket
0               0               0               0
1               0               0               1
2               0               8               0
3               0               8               1
4               0               2               0
5               0               2               1
6               0               10              0
7               0               10              1
8               0               1               0
9               0               1               1
10              0               9               0
11              0               9               1
12              1               0               0
13              1               0               1
14              1               8               0
15              1               8               1
16              1               2               0
17              1               2               1
18              1               10              0
19              1               10              1
20              1               1               0
21              1               1               1
22              1               9               0
23              1               9               1
--------------------------------------------------------------
Socket 0: ( 0 12 8 20 4 16 2 14 10 22 6 18 )
Socket 1: ( 1 13 9 21 5 17 3 15 11 23 7 19 )
--------------------------------------------------------------
```

```
**************************************************************
NUMA domains: 2
--------------------------------------------------------------
Domain 0:
Processors:  0 2 4 6 8 10 12 14 16 18 20 22
Memory: 10988.6 MB free of total 12277.8 MB
--------------------------------------------------------------
Domain 1:
Processors:  1 3 5 7 9 11 13 15 17 19 21 23
Memory: 10986.1 MB free of total 12288 MB
--------------------------------------------------------------

**************************************************************
Graphical:
**************************************************************
Socket 0:
+-------------------------------------------------------------------------+
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  0  12 | |  8  20 | |  4  16 | |  2  14 | | 10  22 | |  6  18 | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +---------------------------------------------------------------+ |
| |                              12MB                             | |
| +---------------------------------------------------------------+ |
+-------------------------------------------------------------------------+
Socket 1:
+-------------------------------------------------------------------------+
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  1  13 | |  9  21 | |  5  17 | |  3  15 | | 11  23 | |  7  19 | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
```

```
jinx9 $  /nethome/rvuduc3/local/jinx/likwid/2.2.1-ICC/bin/likwid-topology -g
```

Exploiting NUMA: Linux “first-touch” policy

```
a = /* … allocate buffer … */;

for (i = 0; i < n; ++i) {
  a[i] = /* … initial value … */ ;
}
```

```
#pragma omp parallel for …
for (i = 0; i < n; ++i) {
  a[i] += foo (i);
}
```

Exploiting NUMA: Linux “first-touch” policy

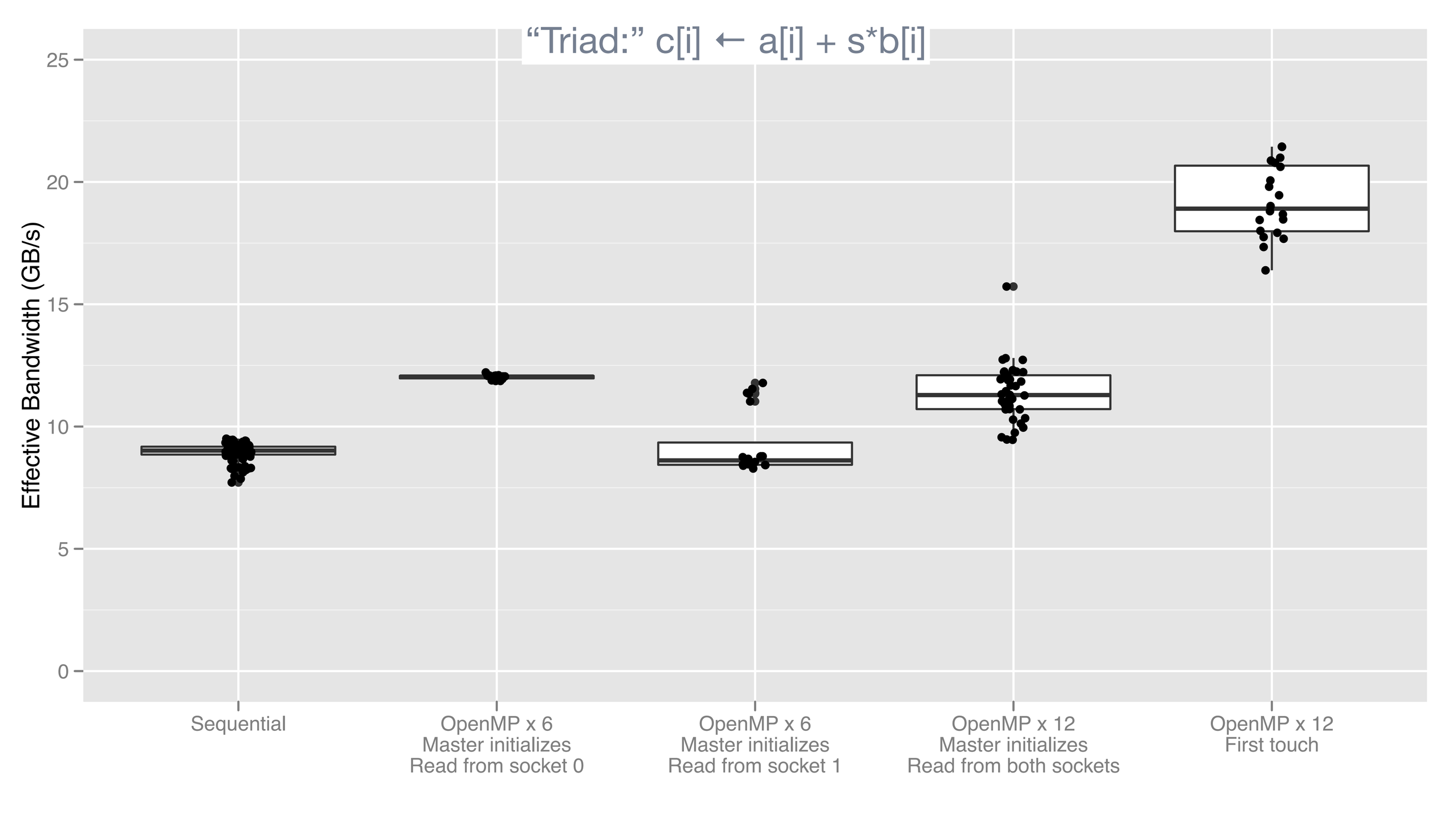
```
a = /* … allocate buffer … */;
#pragma omp parallel for … schedule(static)
for (i = 0; i < n; ++i) {
  a[i] = /* … initial value … */ ;
}
```

```
#pragma omp parallel for … schedule(static)
for (i = 0; i < n; ++i) {
  a[i] += foo (i);
}
```

Thread binding

- Key environment variables
  - **OMP_NUM_THREADS**: Number of OpenMP threads
  - **GOMP_CPU_AFFINITY**: Specify thread-to-core binding

- Consider: 2-socket x 6-core system, main thread initializes data and ‘p’ OpenMP threads operate
  - env OMP_NUM_THREADS=6 GOMP_CPU_AFFINITY=“0 2 4 6 … 22” ./run-program …
    (shorthand: GOMP_CPU_AFFINITY=“0-22:2”)
  - env OMP_NUM_THREADS=6 GOMP_CPU_AFFINITY=“1 3 5 7 … 23” ./run-program …
    (shorthand: GOMP_CPU_AFFINITY=“1-23:2”)


“Triad:” c[i] ← a[i] + s*b[i]
Effective Bandwidth (GB/s)
25
20
15
10
5
0
Sequential
OpenMP x 6
Master initializes
Read from socket 0
OpenMP x 6
Master initializes
Read from socket 1
OpenMP x 12
Master initializes
Read from both sockets
OpenMP x 12
First touch

Backup

Expressing task parallelism (I)

```
#pragma omp parallel sections [… e.g., nowait]
{
  #pragma omp section
  foo ();

  #pragma omp section
  bar ();
}
```

Expressing task parallelism (II): Tasks (new in OpenMP 3.0)

Like Cilk's *spawn* construct

```
void foo () {
  // A & B independent
  A ();
  B ();
}
```

```
// Idiom for tasks
void foo () {
#pragma omp parallel
  {
  #pragma omp single nowait
    {
    #pragma omp task
      A ();
    #pragma omp task
      B ();
    }
  }
}
```


http://wikis.sun.com/display/openmp/Using+the+Tasking+Feature

Expressing task parallelism (II): Tasks (new in OpenMP 3.0)

Like Cilk's *spawn* construct

```
void foo () {
  // A & B independent
  A ();
  B ();
}
```

```
// Idiom for tasks
void foo () {
#pragma omp parallel
  {
  #pragma omp single nowait
    {
    #pragma omp task
      A ();
      // Or, let parent run B
      B ();
    }
  }
}
```


http://wikis.sun.com/display/openmp/Using+the+Tasking+Feature

Variation 1: Fixed chunk size

- Kruskal and Weiss (1985) give a model for computing optimal chunk size
  - Independent subtasks
  - Assumed distributions of running time for each subtask (*e.g.*, IFR)
  - Overhead for extracting task, also random
- Limitation: Must know distribution, though '*n / p*' works (~ .8x optimal for large *n/p*)
- Ref: "Allocating independent subtasks on parallel processors"

Variation 2: Guided self-scheduling

- Idea
  - Large chunks at first to avoid overhead
  - Small chunks near the end to even-out finish times
  - Chunk size $K_i = ceil(R_i / p)$, R_i = number of remaining tasks
- Polychronopoulos & Kuck (1987): “Guided self-scheduling: A practical scheduling scheme for parallel supercomputers”

Variation 3: Tapering

- Idea
  - Chunk size $K_i = f(R_i; \mu, \sigma)$
  - (μ, σ) estimated using history
  - High-variance ⇒ small chunk size
  - Low-variance ⇒ larger chunks OK
  - A little better than guided self-scheduling
- *Ref*: S. Lucco (1994), “Adaptive parallel programs.” PhD Thesis.

$$
\begin{aligned}
\kappa &= \text{min. chunk size} \\
h &= \text{selection overhead} \\
\implies K_i &= f\left(\frac{\sigma}{\mu}, \kappa, \frac{R_i}{p}, h\right)
\end{aligned}
$$

Variation 4: Weighted factoring

- What if hardware is heterogeneous?

- Idea: Divide task cost by computational power of requesting node
- *Ref*: Hummel, Schmit, Uma, Wein (1996). “Load-sharing in heterogeneous systems using weighted factoring.” In *SPAA*

When self-scheduling is useful

- Task cost unknown
- Locality not important
- Shared memory or “small” numbers of processors
- Tasks without dependencies; can use with, but most analysis ignores this