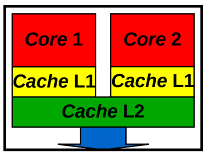
Programmazione di Processori *MultiCore* – V lezione

Federico Massaioli (`federico.massaioli@caspur.it`)

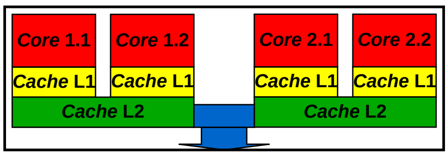
CASPUR e Università degli Studi di Roma "La Sapienza"
Laurea Magistrale in Informatica
Anno accademico 2008-2009

Multicore: tutti i gusti Intel

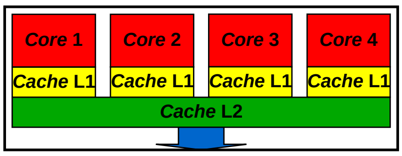
Sottosistema di memoria esterno alla CPU



Dual-Core

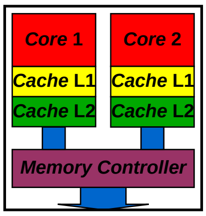


Dual Dual-Core



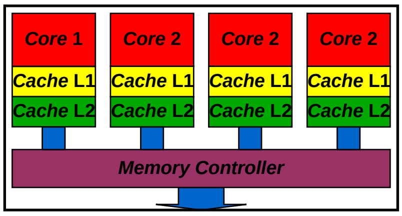
Quad-Core

Multicore: tutti i gusti AMD



Dual-Core

Solo i moduli RAM sono esterni alla CPU



Quad-Core

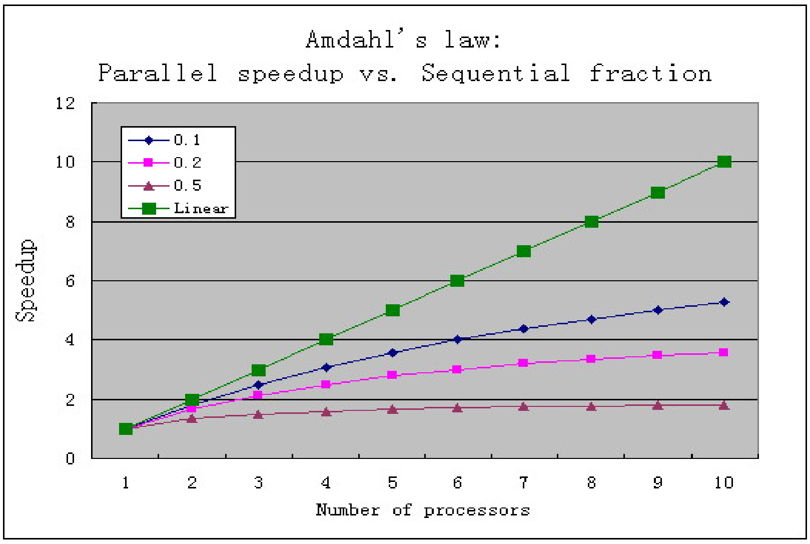
Multicore: come sfruttarli?

- Scopo: velocizzare un’applicazione
- Concorrenza: flussi di esecuzione sincronizzati tra loro
- Parallelismo: i flussi di esecuzione avanzano contemporaneamente
- Con la concorrenza
  - Modularità, separazione funzionale
  - Non necessariamente velocizzazione
- Con il parallelismo
  - Velocizzazione dell’esecuzione

Aspettative?

- Legge di Amdahl
  - T: tempo di elaborazione
  - P: numero di processori utilizzabili
  - αT: parte intrinsecamente seriale
  - (1-α)T: parte dell'elaborazione suddividibile in parti uguali tra i processori
  - $T_P = \alpha T + (1-\alpha)T/P$: tempo su P processori
  - $T_\infty = \alpha T$: limite asintotico
  - $S_P = T/ T_P$: *speedup*
  - $E_P = S_P/P$: efficienza
- Troppo semplice…

La legge di Amdahl a colpo d'occhio



La legge di Amdahl è:

- Ottimista:
  - T_P costante per P>K fissato
  - K differenti in parti diverse dell'applicazione
  - assume una parallelizzazione bilanciata
  - trascura costi di coordinazione e sincronizzazione dei flussi di esecuzione
- Pessimista:
  - più risorse (es. *cache*)
  - più risorse (es. *memory bandwidth*)
  - trascura l'algoritmo: il miglior algoritmo seriale potrebbe essere pessimo in parallelo
  - trascura *layout* dei dati

Esempio: ricerca binaria *vs.* lineare

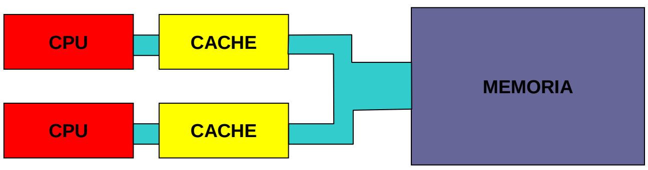
- La ricerca binaria non è parallelizzabile
  - ma richiede dati ordinati
  - il *sorting* è parzialmente parallelizzabile
- La ricerca lineare è parallelizzabile
  - si passa O(N/2) ad O(N/(2×P))
- Approcci possibili a seconda dei dati
  - *sorting* parallelo
  - ricerca lineare parallela
  - suddivisione dei dati, poi ogni processore esegue *sorting* e ricerca binaria locali
- Altre caratteristiche dei dati
  - serve solo il primo *match* o tutti?
  - i dati sono disposti casualmente o *clustered*?

Cache: condivise o disgiunte?

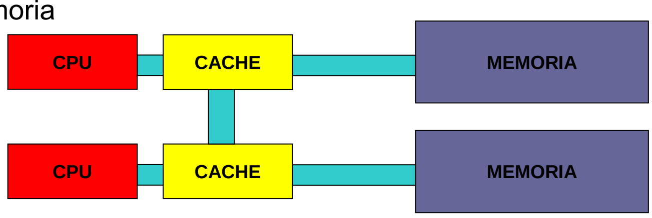
- Diversi flussi di esecuzione possono accedere le stesse locazioni o locazioni diverse
- *Cache* condivise accedendo locazioni:
  - differenti: contesa per la *cache*
  - stesse: *cache* 'efficace' più grande
- *Cache* disgiunte accedendo locazioni:
  - differenti: *cache* 'efficace' più grande
  - stesse: non c'è vantaggio
- Dipende dalle applicazioni

La *bandwidth* verso la memoria

- Le macchine multiprocessore (*single*- o *multi-core*) sono sempre più comuni
- *Uniform Memory Access*: le CPU condividono il *bus* di memoria



- *Non Uniform Memory Access*: ogni CPU ha il suo banco ed il suo *bus* di memoria



UMA, NUMA e *performance*

- Problemi principali:
  - se un processo o *thread* viene trasferito da una CPU ad un'altra, va perso tutto il lavoro speso per usare bene la *cache*
  - macchine UMA: processi "*memory intensive*" su più CPU contendono per il *bus*, rallentandosi a vicenda
  - macchine NUMA: codice eseguito da una CPU con i dati nella memoria di un'altra portano a rallentamenti del 50÷100%
- Dipende, di nuovo, dall'applicazione
- Soluzioni:
  - *binding* di processi o *thread* alle CPU
  - *memory affinity* (Linux su Opteron, **`numactl`**)

Multicore: processi o *thread*?

- Processi + *message passing*
  - processi separati
  - consente uso di più macchine
  - *scheduling* più oneroso
  - inefficiente dentro la singola macchina (copie memoria-memoria)
- *Multithreading*
  - memoria dati condivisa
  - *scheduling* meno oneroso
  - *data race* → mutua esclusione → rischio *deadlock*
  - consistenza della memoria: garantita dalle *cache,* non dai *read/write buffer* delle CPU
- Processi + *shared memory segment*
  - combina alcuni svantaggi e ne aggiunge di altri
  - utile per altri scopi

OK, i *thread*, ma quali?

- *Pthreads, Windows threads*
  - troppo basso livello
  - *scheduler* generico
  - orientati alla concorrenza
  - occorre sviluppare molto codice di infrastruttura
- Java: inefficiente
- Estensioni ‘mirate’ a linguaggi esistenti (Cilk, OpenMP,…)
  - minima intrusività nel codice
  - equivalenza seriale
  - adattamento automatico all’ambiente (*core* disponibili)
  - concepiti per il parallelismo
  - *runtime* (*scheduler*, *routine* di supporto) efficiente
- Altri approcci: Intel *Threading Building Blocks*

OpenMP 2.5
In 8 slides

Federico Massaioli (`federico.massaioli@caspur.it`)

CASPUR e Università degli Studi di Roma "La Sapienza"
Laurea Magistrale in Informatica
Anno accademico 2008-2009

OpenMP

- Oct 1997 – 1.0 Fortran
- Oct 1998 – 1.0 C/C++
- Nov 1999 – 1.1 Fortran (interpretations added)
- Nov 2000 – 2.0 Fortran
- Mar 2002 – 2.0 C/C++
- May 2005 – 2.5 Fortran/C/C++ (mostly a merge)

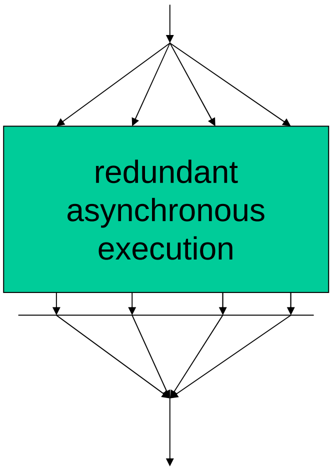
- Main targets:
  - rapid, good enough speedup on small SMPs
  - with an incremental approach to parallelization
  - without preventing scalability on big SMPs
  - and trying to be "compatible" with the serial version

OpenMP in real life

- Rapidly adopted by vendors
  - at least for first specs
  - some slow down for the latest ones
  - some runtimes not up to user expectations
  - interest is catching up
- Mostly used in HPC and scientific computing
  - plus some ISVs
  - plus some numerical libraries
  - but growing interest in other fields
- Typical parallelization approach
  - incrementally parallelize, checking for correctness
  - most frequent pattern: parallel (a.k.a. worksharing) loops
  - then take care of data locality

The parallel construct

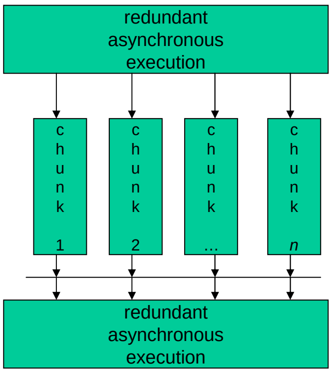
```
#pragma omp parallel
{
// I'm not alone, anymore
// do this…
// do that…

  switch(omp_get_thread_num()) {
    case 0 :
      // …
     case 1:
      // …
  }
}
```



Most users do not even realize how powerful this is in itself

Worksharing: the loop construct

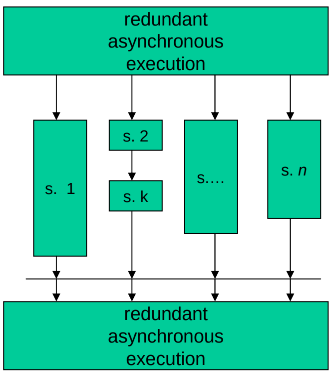
```
#pragma omp for
for (i= 0; i < N; ++i) {
// …
// …
}
```



The number of iterations must be known each time the construct is entered at runtime, and must be the same for each thread

Worksharing: the sections construct

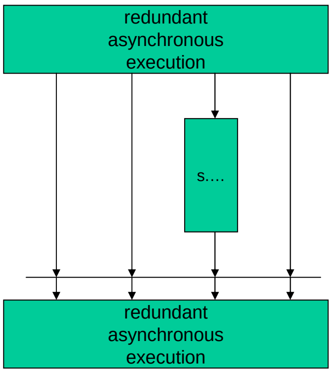
```
#pragma omp sections
{
  #pragma omp section
  {
    // do this …
  }
 #pragma omp section
  {
    // do that …
  }
  // …
}
```



The **`omp section`** directive must be closely nested in a sections construct, where no other worksharing construct can appear.

Worksharing: the single construct

```
#pragma omp single nowait
{
  // I’m the only one working…


}
```



More complex than it appears, if the barrier is removed!

And more stuff…

- Data scoping clauses
  - shared
  - private
- Synchronization and ordering
  - barriers
  - critical sections
  - atomic updates of scalar types
  - ordered sections
- Memory consistency
  - the “dreaded” **`omp flush`**
- Syntactic sugar
  - combined parallel and worksharing constructs
  - initialization of private variables, reductions
  - seems very sweet to average programmers…

OpenMP 2.5: how does it fare?

- Where it shines
  - simplicity
  - productivity (Hochstein et al., SC05)
  - great performance/manWatt
- Where it lags (from HPC caves)
  - manWatt/performance curve too steep at high performances
  - too simple
  - lacks support for cache and CPU affinity
  - not enough control/not expliciting the metal
- Where it lags (from the seaside, on a glorious day)
  - performance/manWatt still too low
  - still too complex
  - need far more flexibility
  - runtimes should do more for users
- Conflicting requirements and tensions

An Overview Of OpenMP 2.5

Ruud van der Pas

Senior Staff Engineer
Sun Microsystems
Menlo Park, CA, USA

IWOMP 2008
Purdue University
West Lafayette, IN, USA
May 12-14, 2008

Outline

- ❑ ***OpenMP Guided Tour***
- ❑ ***OpenMP Overview***
  - *Directives*
  - *Environment variables*
  - *Run-time environment*
- ❑ ***Global Data***
- ❑ ***Wrap-Up***
- ❑ ***Appendix: A First Glimpse Into OpenMP 3.0***

OpenMP Guided Tour

http://www.openmp.org

http://www.compunity.org

Shameless Plug - "Using OpenMP"

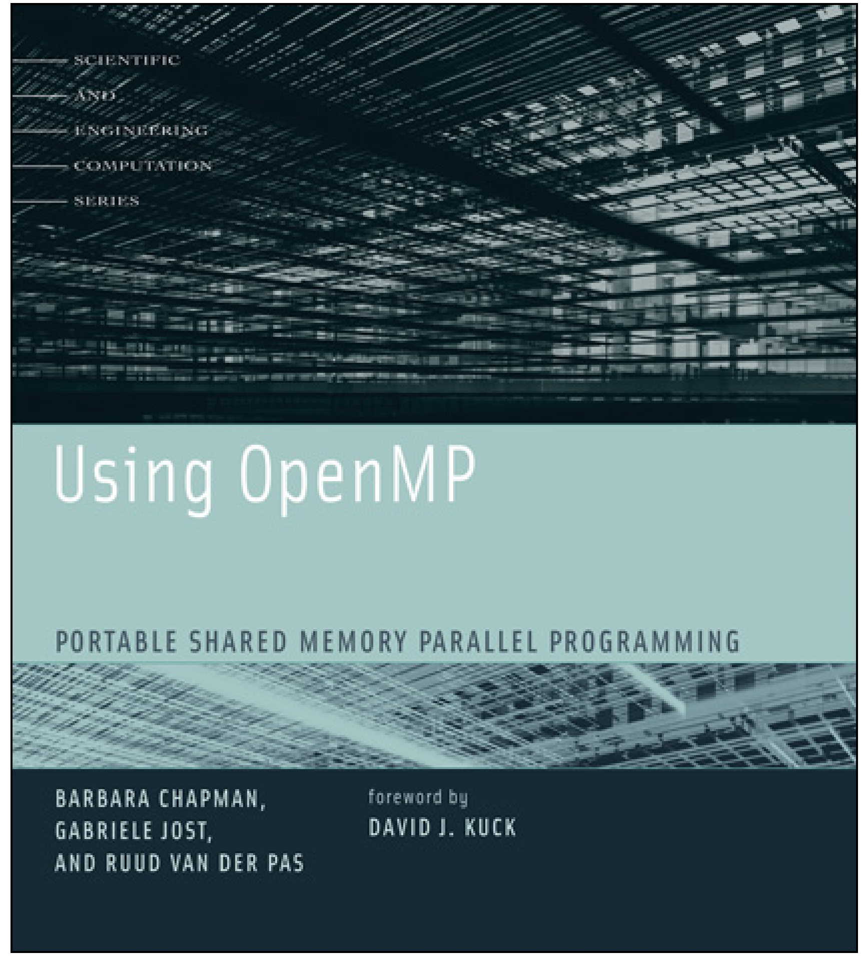
"Using OpenMP"

Portable Shared Memory Parallel Programming

Chapman, Jost, van der Pas

MIT Press, October 2007

ISBN-10: 0-262-53302-2
ISBN-13: 978-0-262-53302-7



What is OpenMP?

- ***De-facto standard API for writing <u>shared memory parallel applications</u> in C, C++, and Fortran***
- ***Consists of:***
  - ***Compiler directives***
  - ***Run time routines***
  - ***Environment variables***
- ***Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)***
- ***Latest Specification: Version 2.5***
- ***Version 3.0 has been in the works since September 2005, draft specification released October 2007***

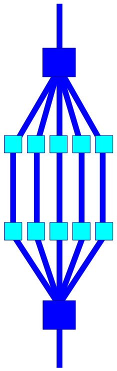
When to consider OpenMP?

- ***The compiler may not be able to do the parallelization in the way you like to see it:***
  - ***A loop is not parallelized***
    - ✔ ***The data dependence analysis is not able to determine whether it is safe to parallelize or not***
  - ***The granularity is not high enough***
    - ✔ ***The compiler lacks information to parallelize at the highest possible level***
- ***This is when explicit parallelization through OpenMP directives and functions comes into the picture***

Advantages of OpenMP

- ***Good performance and scalability***
  - *If you do it right ....*
- ***De-facto standard***
- ***An OpenMP program is portable***
  - *Supported by a large number of compilers*
- ***Requires little programming effort***
- ***Allows the program to be parallelized incrementally***
- ***Maps naturally onto a multicore architecture:***
  - *Lightweight*
  - *Each OpenMP thread in the program can be executed by a hardware thread*

A first OpenMP example



For-loop with independent iterations

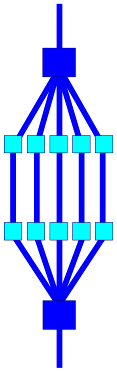
```
for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
```

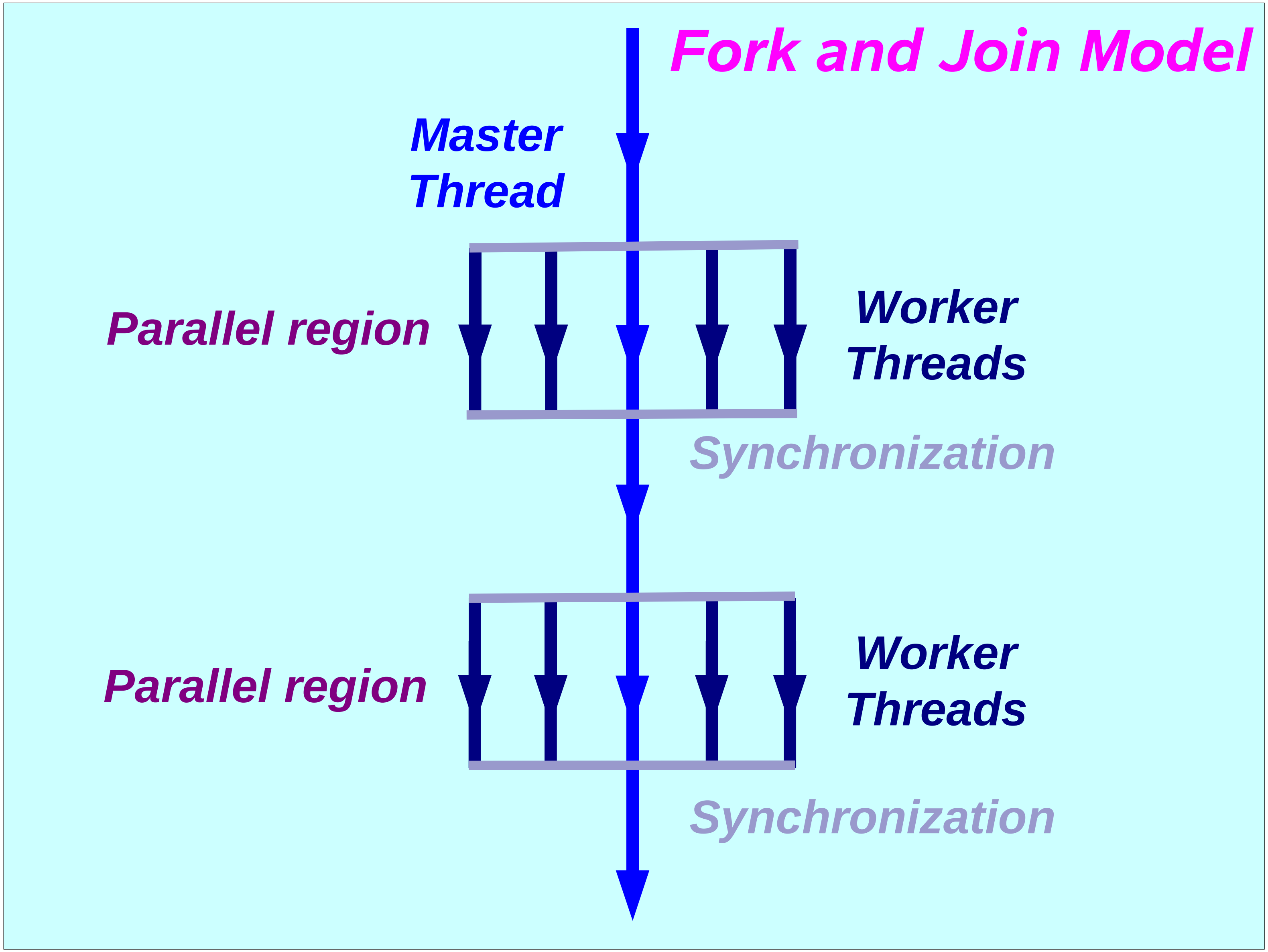
For-loop parallelized using an OpenMP pragma

```
#pragma omp parallel for  \
        shared(n, a, b, c)\
        private(i)
for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
```

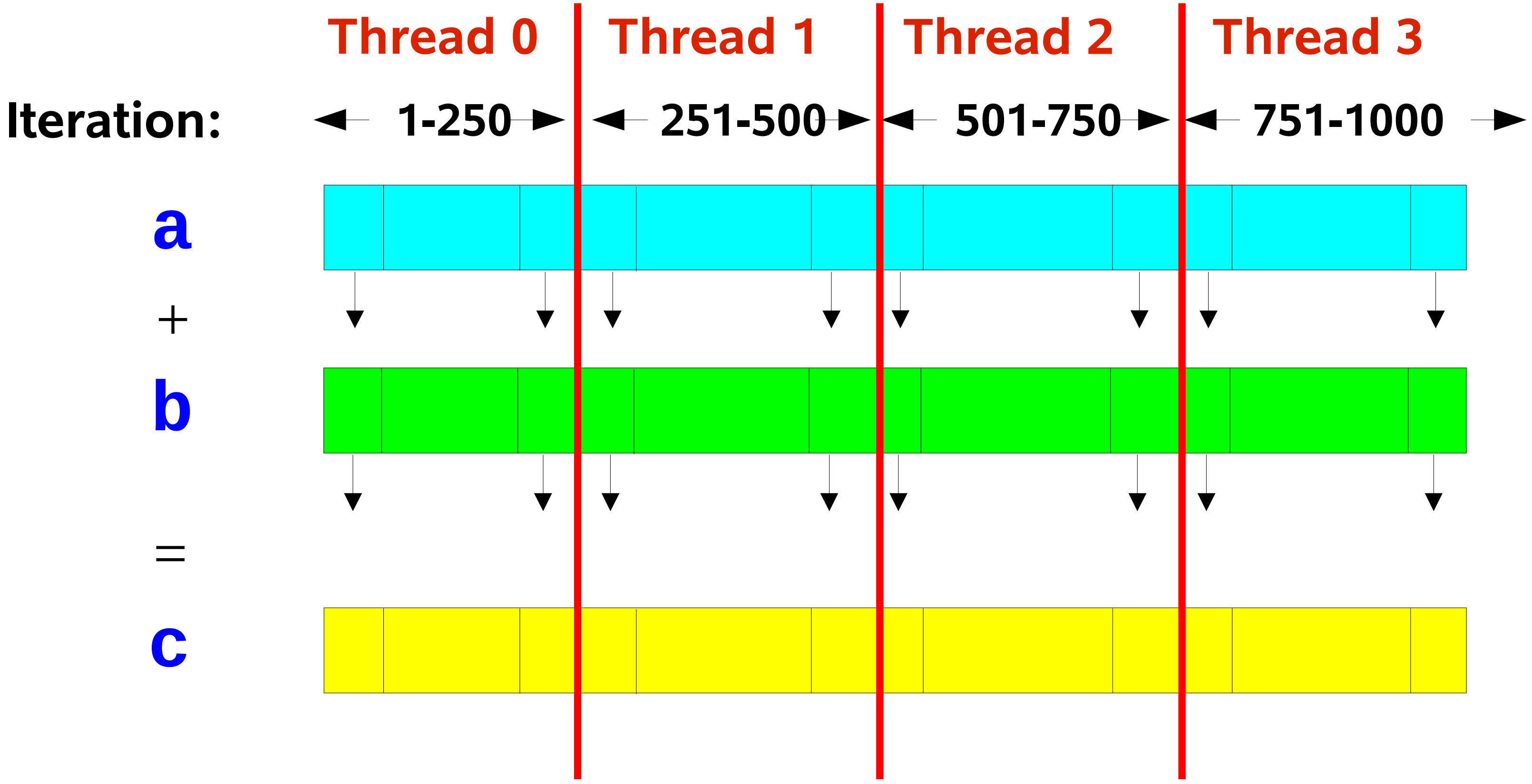
```
% cc -xopenmp source.c
% setenv OMP_NUM_THREADS 4
% a.out
```

The OpenMP execution model

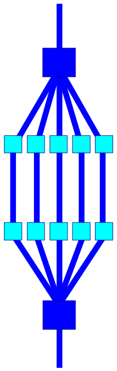




Example parallel execution



A loop parallelized with OpenMP

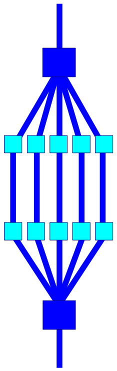


```
#pragma omp parallel default(none) \
              shared(n,x,y) private(i)
{
 #pragma omp for
   for (i=0; i<n; i++)
       x[i] += y[i];
} /*-- End of parallel region --*/
```

```
!$omp parallel default(none)     &
!$omp shared(n,x,y) private(i)
!$omp do
      do i = 1, n
         x(i) = x(i) + y(i)
      end do
!$omp end do
!$omp end parallel
```

clauses

Components of OpenMP



Directives

- *Parallel regions*
- *Work sharing*
- *Synchronization*
- *Data-sharing attributes*
  - *private*
  - *firstprivate*
  - *lastprivate*
  - *shared*
  - *reduction*
- *Orphaning*

Environment variables

- *Number of threads*
- *Scheduling type*
- *Dynamic thread adjustment*
- *Nested parallelism*

Runtime environment

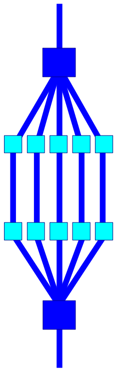
- *Number of threads*
- *Thread ID*
- *Dynamic thread adjustment*
- *Nested parallelism*
- *Timers*
- *API for locking*

Directive format

- ❑ *C: directives are case sensitive*
  - *Syntax:* **#pragma omp directive [clause [clause] ...]**
- ❑ *Continuation: use \ in pragma*
- ❑ *Conditional compilation:* **_OPENMP** *macro is set*

- ❑ *Fortran: directives are case insensitive*
  - *Syntax:* **sentinel directive [clause [[,] clause]...]**
  - *The sentinel is one of the following:*
    - ✔ ***!$OMP** or **C$OMP** or ***$OMP*** *(fixed format)*
    - ✔ ***!$OMP*** *(free format)*
- ❑ *Continuation: follows the language syntax*
- ❑ *Conditional compilation:* **!$** *or* **C$ -> 2 spaces**

A more elaborate example



```
#pragma omp parallel if (n>limit) default(none) \
        shared(n,a,b,c,x,y,z) private(f,i,scale)
{
    f = 1.0;
#pragma omp for nowait

    for (i=0; i<n; i++)
       z[i] = x[i] + y[i];


#pragma omp for nowait

    for (i=0; i<n; i++)
       a[i] = b[i] + c[i];


#pragma omp barrier
            ....
    scale = sum(a,0,n) + sum(z,0,n) + f;
            ....
} /*-- End of parallel region --*/
```

Statement is executed by all threads

parallel loop (work is distributed)

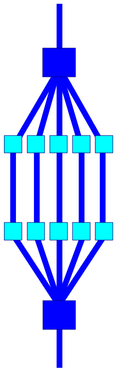
parallel loop (work is distributed)

synchronization

Statement is executed by all threads

parallel region

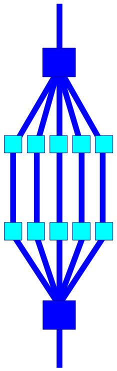
Another OpenMP example

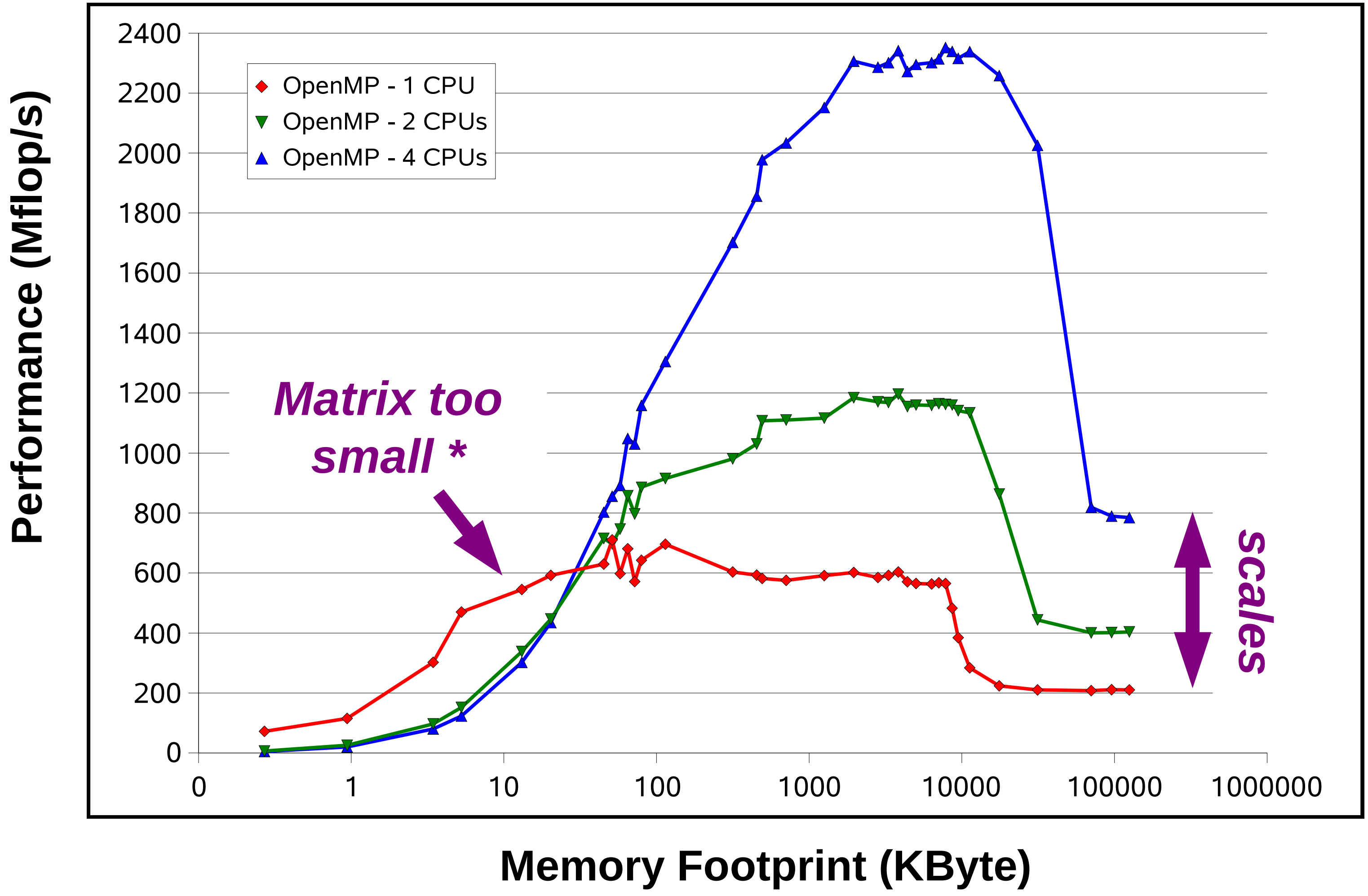


```
 1 void mxv_row(int m,int n,double *a,double *b,double *c)
 2 {
 3  int i, j;
 4  double sum;
 5
 6 #pragma omp parallel for default(none) \
 7             private(i,j,sum) shared(m,n,a,b,c)
 8  for (i=0; i<m; i++)
 9  {
10    sum = 0.0;
11    for (j=0; j<n; j++)
12      sum += b[i*n+j]*c[j];
13     a[i] = sum;
14  } /*-- End of parallel for --*/
15 }
```

```
% cc -c -fast -xrestrict -xopenmp -xloopinfo mxv_row.c
"mxv_row.c", line  8: PARALLELIZED, user pragma used
"mxv_row.c", line 11: not parallelized
```

OpenMP performance



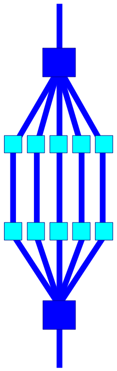


SunFire 6800
UltraSPARC III Cu @ 900 MHz
8 MB L2-cache

***) With the IF-clause in OpenMP this performance degradation can be avoided**

OpenMP Directives

Terminology and behavior



- ***OpenMP Team := Master + Workers***
- ***A <u>Parallel Region</u> is a block of code executed by all threads simultaneously***
  - ***The master thread always has thread ID 0***
  - ***Thread adjustment (if enabled) is only done before entering a parallel region***
  - ***Parallel regions can be nested, but support for this is implementation dependent***
  - ***An "if" clause can be used to guard the parallel region; in case the condition evaluates to "false", the code is executed serially***
- ***A <u>work-sharing construct</u> divides the execution of the enclosed code region among the members of the team; in other words: they split the work***

About OpenMP clauses

- *Many OpenMP directives support clauses*
- *These clauses are used to specify additional information with the directive*
- *For example,* ***private(a)*** *is a clause to the for directive:*
  - **#pragma omp for private(a)**
- *Before we present an overview of all the directives, we discuss several of the OpenMP clauses first*
- *The specific clause(s) that can be used, depends on the directive*

The if/private/shared clauses

if (scalar expression)

- ✔ *Only execute in parallel if expression evaluates to true*
- ✔ *Otherwise, execute serially*

```
#pragma omp parallel if (n > threshold) \
        shared(n,x,y) private(i)
  {
    #pragma omp for
     for (i=0; i<n; i++)
        x[i] += y[i];
  } /*-- End of parallel region --*/
```

private (list)

- ✔ *No storage association with original object*
- ✔ *All references are to the local object*
- ✔ *Values are undefined on entry and exit*

shared (list)

- ✔ *Data is accessible by all threads in the team*
- ✔ *All threads access the same address space*

About storage association

- ***<u>Private variables are undefined on entry and exit of the parallel region</u>***
- ***The value of the original variable (before the parallel region) is <u>undefined</u> after the parallel region !***
- ***A private variable within a parallel region has <u>no storage association</u> with the same variable outside of the region***
- ***Use the first/last private clause to override this behavior***
- ***We illustrate these concepts with an example***

Example private variables

```
main()
{
  A = 10;

 #pragma omp parallel
 {
  #pragma omp for private(i) firstprivate(A) lastprivate(B)...
  for (i=0; i<n; i++)
  {
        ....
        B = A + i;            /*-- A undefined, unless declared
                                   firstprivate --*/
        ....
  }

  C = B;                      /*-- B undefined, unless declared
                                   lastprivate --*/

 } /*-- End of OpenMP parallel region --*/
}
```

Disclaimer: This code fragment is not very meaningful and only serves to demonstrate the clauses

The first/last private clauses

firstprivate (list)

- ✔ ***All variables in the list are initialized with the value the original object had before entering the parallel construct***

lastprivate (list)

- ✔ ***The thread that executes the <u>sequentially last</u> iteration or section updates the value of the objects in the list***

The default clause

default (none | shared | private) — *Fortran*

default (none | shared) — *C/C++*

Note: default(private) is not supported in C/C++

none

- ✔ *No implicit defaults*
- ✔ *Have to scope all variables explicitly*

shared

- ✔ *All variables are shared*
- ✔ *The default in absence of an explicit "default" clause*

private

- ✔ *All variables are private to the thread*
- ✔ *Includes common block data, unless THREADPRIVATE*

The reduction clause - Example

```
      sum = 0.0
!$omp parallel default(none) &
!$omp shared(n,x) private(i)
!$omp do reduction (+:sum)
      do i = 1, n
         sum = sum + x(i)
      end do
!$omp end do
!$omp end parallel
      print *,sum
```

Variable SUM is a shared variable

- ☞ *Care needs to be taken when updating shared variable SUM*
- ☞ *With the reduction clause, the OpenMP compiler generates code such that a race condition is avoided*

The reduction clause

reduction ([operator | intrinsic]) : list)	*Fortran*
reduction (operator : list)	*C/C++*

- ***Reduction variable(s) must be shared variables***
- ***A reduction is defined as:***

Check the docs for details

Fortran

```
x = x operator expr
x = expr operator x
x = intrinsic (x, expr_list)
x = intrinsic (expr_list, x)
```

C/C++

```
x = x operator expr
x = expr operator x
x++, ++x, x--, --x
x <binop> = expr
```

- ***Note that the value of a reduction variable is undefined from the moment the first thread reaches the clause till the operation has completed***
- ***The reduction can be hidden in a function call***

Barrier/1

Suppose we run each of these two loops in parallel over i:

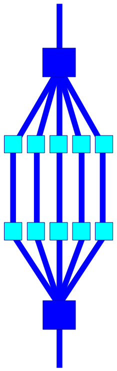
```
for (i=0; i < N; i++)
   a[i] = b[i] + c[i];
```

```
for (i=0; i < N; i++)
   d[i] = a[i] + b[i];
```

This may give us a wrong answer (one day)

Why ?

Barrier/2



*We need to have updated all of a[ ] first, before using a[ ] **

```
for (i=0; i < N; i++)
   a[i] = b[i] + c[i];
```

wait !

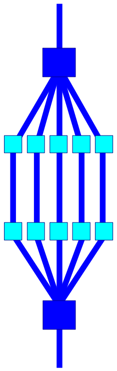
barrier

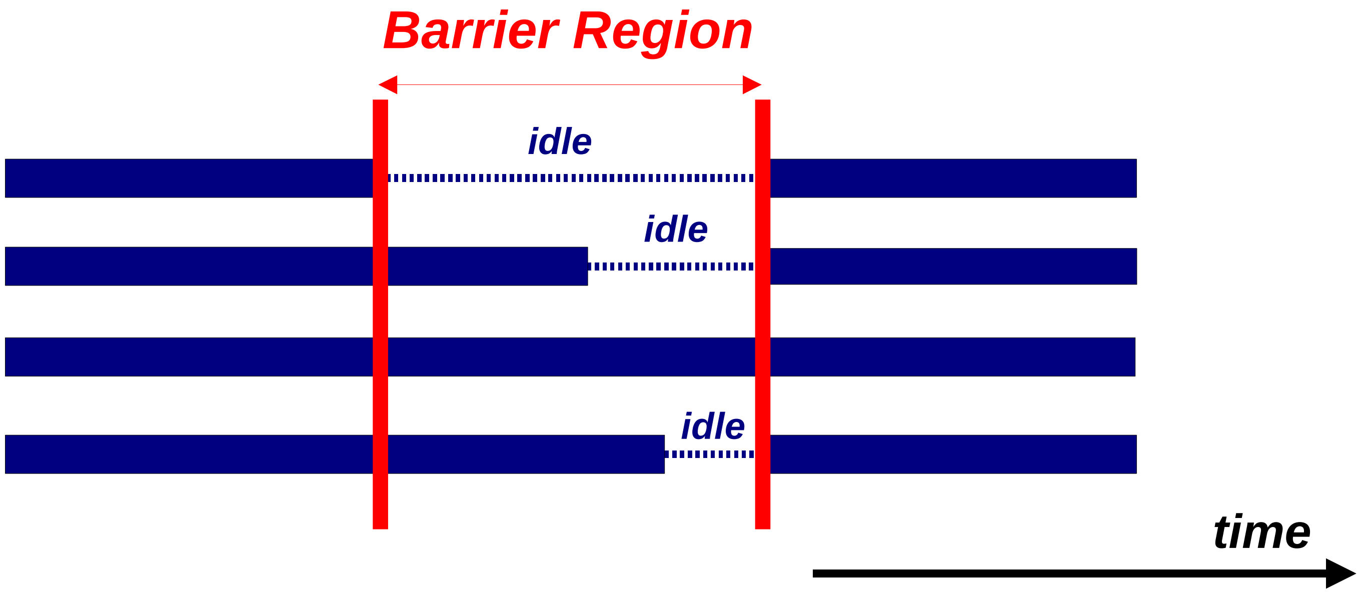
```
for (i=0; i < N; i++)
   d[i] = a[i] + b[i];
```

All threads wait at the barrier point and only continue when all threads have reached the barrier point

***) If there is the guarantee that the mapping of iterations onto threads is identical for both loops, there will not be a data race in this case**

Barrier/3



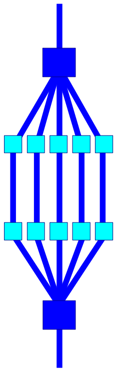


Barrier syntax in OpenMP:

```
#pragma omp barrier
```

```
!$omp barrier
```

When to use barriers ?



- ❑ *When data is updated asynchronously and the data integrity is at risk*
- ❑ *Examples:*
  - *Between parts in the code that read and write the same section of memory*
  - *After one timestep/iteration in a solver*
- ❑ *Unfortunately, barriers tend to be expensive and also may not scale to a large number of processors*
- ❑ *Therefore, use them with care*

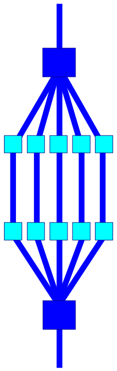
The nowait clause

- *To minimize synchronization, some OpenMP directives/ pragmas support the optional* ***nowait*** *clause*
- *If present, threads do not synchronize/wait at the end of that particular construct*
- *In Fortran the* ***nowait*** *clause is appended at the closing part of the construct*
- *In C, it is one of the clauses on the pragma*

```
#pragma omp for nowait
{
        :
}
```

```
!$omp do
        :
        :
!$omp end do nowait
```

The Parallel Region



A parallel region is a block of code executed by multiple threads simultaneously

```
!$omp parallel [clause[[,] clause] ...]

   "this is executed in parallel"

!$omp end parallel (implied barrier)
```

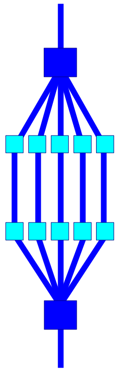
```
#pragma omp parallel [clause[[,] clause] ...]
{
   "this is executed in parallel"

} (implied barrier)
```

The Parallel Region - Clauses

A parallel region supports the following clauses:

if	**(*scalar expression*)**	
private	**(*list*)**	
shared	**(*list*)**	
default	***(none|shared)***	***(C/C++)***
default	***(none|shared|private)***	***(Fortran)***
reduction	**(*operator*: *list*)**	
copyin	**(*list*)**	
firstprivate	**(*list*)**	
num_threads	**(*scalar_int_expr*)**	

Work-sharing constructs



The OpenMP work-sharing constructs

```
#pragma omp for
{
   ....
}

!$OMP DO
   ....
!$OMP END DO
```

```
#pragma omp sections
{
      ....
}

!$OMP SECTIONS
      ....
!$OMP END SECTIONS
```

```
#pragma omp single
{
      ....
}

!$OMP SINGLE
      ....
!$OMP END SINGLE
```

- ***The work is distributed over the threads***
- ***Must be enclosed in a parallel region***
- ***Must be encountered by all threads in the team, or none at all***
- ***No implied barrier on entry; implied barrier on exit (unless nowait is specified)***
- ***A work-sharing construct does not launch any new threads***

The workshare construct

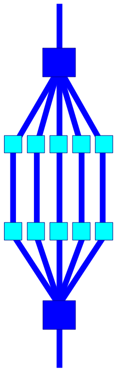
Fortran has a fourth worksharing construct:

```
!$OMP WORKSHARE

    <array syntax>

!$OMP END WORKSHARE [NOWAIT]
```

Example:

```
!$OMP WORKSHARE
    A(1:M) = A(1:M) + B(1:M)
!$OMP END WORKSHARE NOWAIT
```

The omp for/do directive



The iterations of the loop are distributed over the threads

```
#pragma omp for [clause[[,] clause] ...]
   <original for-loop>
```

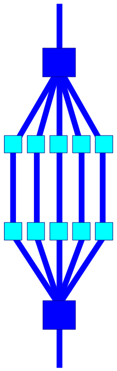
```
!$omp do [clause[[,] clause] ...]
       <original do-loop>
!$omp end do [nowait]
```

Clauses supported:

private	**firstprivate**
lastprivate	**reduction**
ordered*	***schedule*** ← *covered later*
nowait	

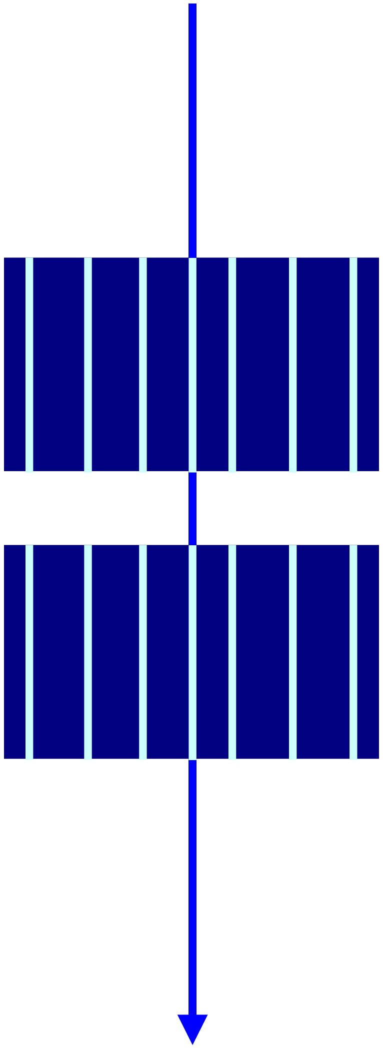
***) Required if ordered sections are in the dynamic extent of this construct**

The omp for directive - Example

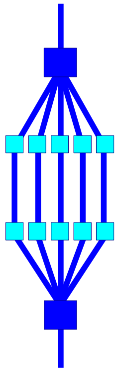


```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
      for (i=0; i<n-1; i++)
          b[i] = (a[i] + a[i+1])/2;
    #pragma omp for nowait
      for (i=0; i<n; i++)
          d[i] = 1.0/c[i];

  } /*-- End of parallel region --*/
```

(implied barrier)



The sections directive



The individual code blocks are distributed over the threads

```
#pragma omp sections [clause(s)]
{
#pragma omp section
          <code block1>
#pragma omp section
          <code block2>
#pragma omp section
            :
}
```

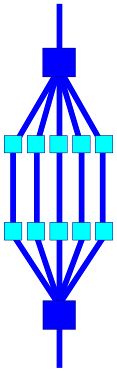
```
!$omp sections [clause(s)]
!$omp section
          <code block1>
!$omp section
          <code block2>
!$omp section
            :
!$omp end sections [nowait]
```

Clauses supported:

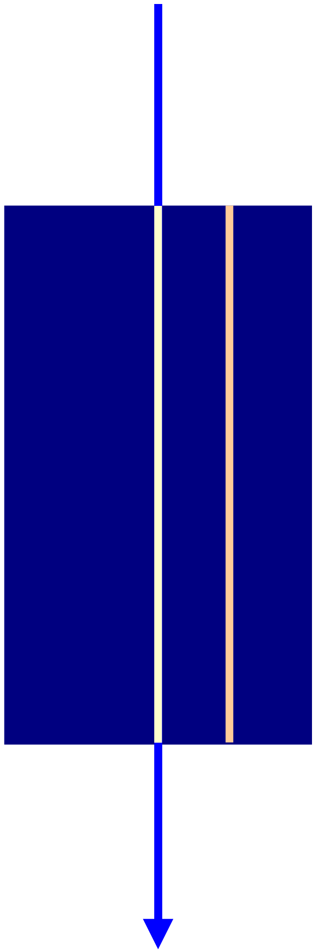
private **firstprivate**
lastprivate **reduction**
nowait

Note: The SECTION directive must be within the lexical extent of the SECTIONS/END SECTIONS pair

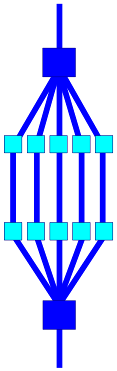
The sections directive - Example



```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
        for (i=0; i<n-1; i++)
            b[i] = (a[i] + a[i+1])/2;

      #pragma omp section
        for (i=0; i<n; i++)
            d[i] = 1.0/c[i];

    } /*-- End of sections --*/

  } /*-- End of parallel region --*/
```



Combined work-sharing constructs



Single PARALLEL loop

```
#pragma omp parallel
#pragma omp for
   for (...)
```

```
#pragma omp parallel for
for (....)
```

```
!$omp parallel
!$omp do
      ...
!$omp end do
!$omp end parallel
```

```
!$omp parallel do
      ...
!$omp end parallel do
```

Single WORKSHARE loop

```
!$omp parallel
!$omp workshare
      ...
!$omp end workshare
!$omp end parallel
```

```
!$omp parallel workshare
      ...
!$omp end parallel workshare
```

Single PARALLEL sections

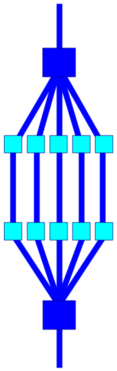
```
#pragma omp parallel
#pragma omp sections
{ ...}
```

```
#pragma omp parallel sections
{ ... }
```

```
!$omp parallel
!$omp sections
      ...
!$omp end sections
!$omp end parallel
```

```
!$omp parallel sections
      ...
!$omp end parallel sections
```

Orphaning



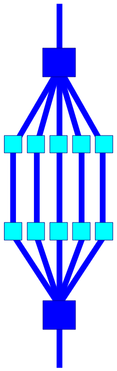
```
              :
!$omp parallel
              :
      call dowork()
              :
!$omp end parallel
              :
```

orphaned work-sharing directive

```
 subroutine dowork()
          :
!$omp do
   do i = 1, n
          :
   end do
!$omp end do
          :
```

- ***The OpenMP standard does not restrict worksharing and synchronization directives (omp for, omp single, critical, barrier, etc.) to be within the lexical extent of a parallel region. These directives can be <u>orphaned</u>***
- ***That is, they can appear outside the lexical extent of a parallel region***

More on orphaning



```
   (void) dowork(); !- Sequential FOR

#pragma omp parallel
{
   (void) dowork(); !- Parallel FOR
}
```

```
void dowork()
{
#pragma omp for
   for (i=0;....)
   {
      :
   }
}
```

- *When an orphaned worksharing or synchronization directive is encountered in the <u>sequential part</u> of the program (outside the dynamic extent of any parallel region), it is executed by the master thread only. In effect, the directive will be ignored*

Parallelizing bulky loops

```
for (i=0; i<n; i++) /* Parallel loop */
{
    a = ...
    b = ... a ..
    c[i] = ....
         ......
    for (j=0; j<m; j++)
    {
      <a lot more code in this loop>
    }
         ......
}
```

Step 1: “Outlining”

```
for (i=0; i<n; i++) /* Parallel loop */
{
     (void) FuncPar(i,m,c,...)
}
```

Still a sequential program

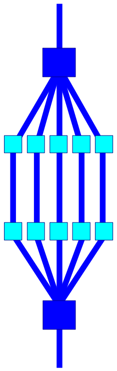
Should behave identically

Easy to test for correctness

But, parallel by design

```
void FuncPar(i,m,c,....)
{
    float a, b; /* Private data */
    int   j;
    a = ...
    b = ... a ..
    c[i] = ....
          ......
    for (j=0; j<m; j++)
    {
      <a lot more code in this loop>
    }
          ......
}
```

Step 2: Parallelize



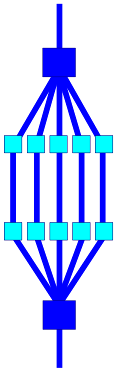
```
#pragma omp parallel for private(i) shared(m,c,..)
   for (i=0; i<n; i++) /* Parallel loop */
   {
        (void) FuncPar(i,m,c,...)
   } /*-- End of parallel for --*/
```

Minimal scoping required

Less error prone

```
void FuncPar(i,m,c,....)
{
    float a, b; /* Private data */
    int   j;
    a = ...
    b = ... a ..
    c[i] = ....
         ......
    for (j=0; j<m; j++)
    {
      <a lot more code in this loop>
    }
         ......
}
```

Single processor region/1



This construct is ideally suited for I/O or initializations

Original Code

```
.....
"read a[0..N-1]";
.....
```

"declare A to be be shared"

```
#pragma omp parallel
{
     .....
     "read a[0..N-1]";
     .....
}
```

one volunteer requested

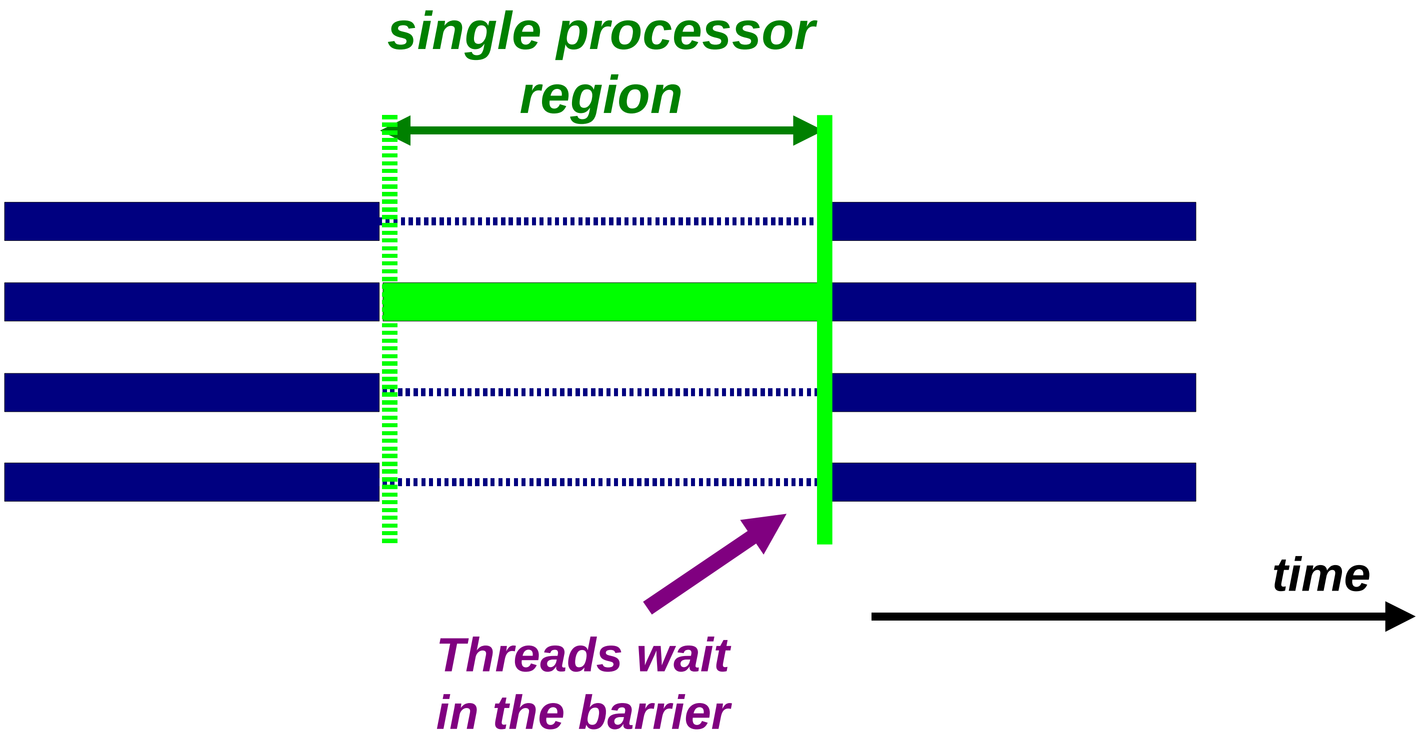
thanks, we're done

May have to insert a barrier here

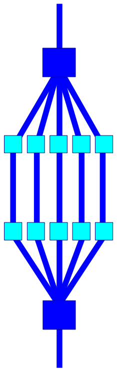
Parallel Version

Single processor region/2

- ***Usually, there is a barrier at the end of the region***
- ***Might therefore be a scalability bottleneck (Amdahl's law)***



SINGLE and MASTER construct



Only one thread in the team executes the code enclosed

```
#pragma omp single [clause[[,] clause] ...]
{
        <code-block>
}
```

```
!$omp single [clause[[,] clause] ...]
   <code-block>
!$omp end single [nowait]
```

Only the <u>master thread</u> executes the code block;

```
#pragma omp master
{<code-block>}
```

```
!$omp master
        <code-block>
!$omp end master
```

There is no implied barrier on entry or exit !

Critical region/1

If sum is a shared variable, this loop can not run in parallel

```
for (i=0; i < N; i++){
      .....
   sum += a[i];
      .....
}
```

We can use a critical region for this:

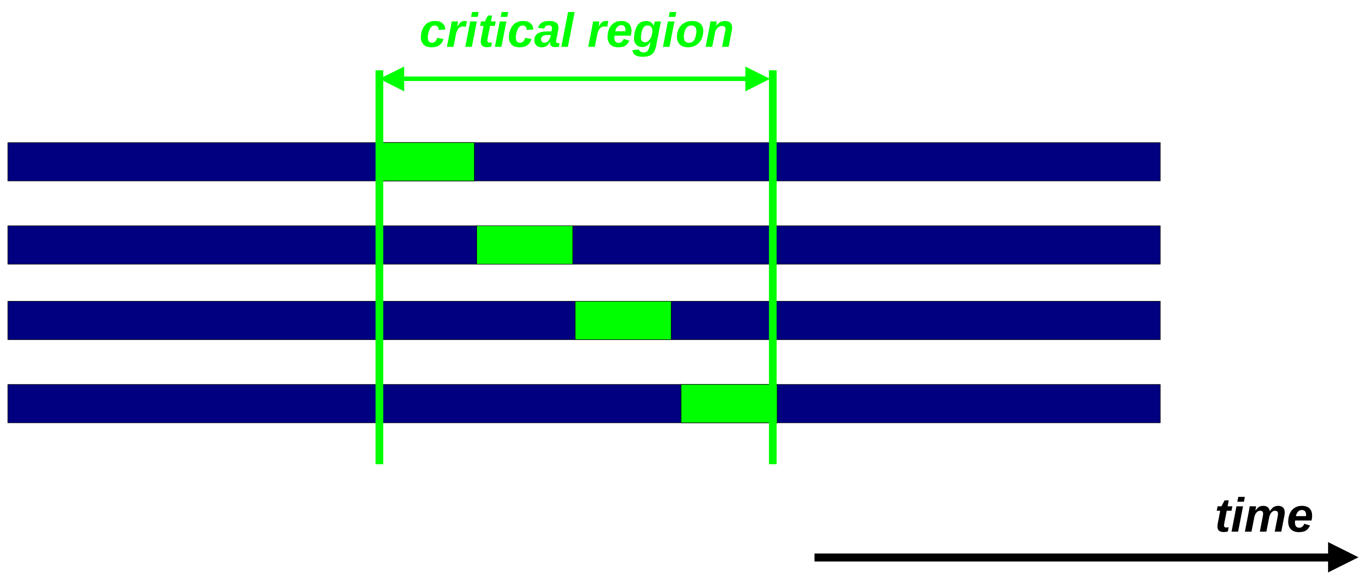
```
for (i=0; i < N; i++){
      .....
   sum += a[i];
      .....
}
```

one at a time can proceed

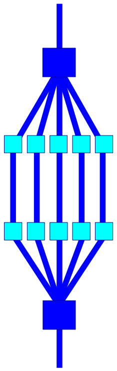
next in line, please

Critical region/2

- ***Useful to avoid a race condition, or to perform I/O (but which still has random order)***
- ***Be aware that your parallel computation may be <u>serialized</u> and so this could introduce a scalability bottleneck (Amdahl's law)***



Critical and Atomic constructs



Critical: All threads execute the code, but only one at a time:

```
#pragma omp critical [(name)]
{<code-block>}
```

```
!$omp critical [(name)]
        <code-block>
!$omp end critical [(name)]
```

There is no implied barrier on entry or exit !

Atomic: only the loads and store are atomic ....

```
#pragma omp atomic
   <statement>
```

```
!$omp atomic
   <statement>
```

This is a lightweight, special form of a critical section

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

More synchronization constructs

The enclosed block of code is executed in the order in which iterations would be executed sequentially:

```
#pragma omp ordered
{<code-block>}
```

```
!$omp ordered
        <code-block>
!$omp end ordered
```

May introduce serialization (could be expensive)

Ensure that all threads in a team have a consistent view of certain objects in memory:

```
#pragma omp flush [(list)]
```

```
!$omp flush [(list)]
```

In the absence of a list, all visible variables are flushed; this could be expensive

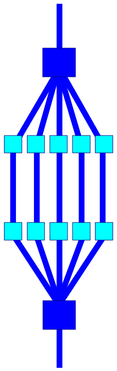
Implied FLUSH - C/C++

The FLUSH pragma is implied on:

```
#pragma omp barrier

exit from parallel region

#pragma omp critical
exit from critical region

#pragma omp ordered
exit from ordered region

exit from for

exit from sections

exit from single
```

The FLUSH pragma is not implied if a nowait clause is present

Implied FLUSH - Fortran



The FLUSH pragma is implied on:

```
!$omp barrier

!$omp critical
!$omp end critical

!$omp parallel [do|sections|workshare]
!$omp end parallel [do|sections|workshare]

!$omp ordered
!$omp end ordered

!$omp end do
!$omp end sections
!$omp end single
!$omp workshare
```

The FLUSH pragma is not implied if a <u>nowait</u> clause is present

NOT implied on:

```
!$omp do
!$omp master
!$omp end master
!$omp single
!$omp workshare
```

Load Balancing

- ***Load balancing is an important aspect of performance***
- ***For regular operations (e.g. a vector addition), load balancing is not an issue***
- ***For less regular workloads, care needs to be taken in distributing the work over the threads***
- ***Examples:***
  - ***Transposing a matrix***
  - ***Multiplication of triangular matrices***
  - ***Parallel searches in a linked list***
- ***For these irregular situations, the* schedule *clause supports various iteration scheduling algorithms***

The schedule clause/1

schedule (static | dynamic | guided [, chunk])
schedule (runtime)

static [, chunk]

- ✔ *Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion*
- ✔ *In absence of "chunk", each thread executes approx. N/P chunks for a loop of length N and P threads*

Example: ***Loop of length 16, 4 threads:***

TID	0	1	2	3
no chunk	1-4	5-8	9-12	13-16
chunk = 2	1-2 9-10	3-4 11-12	5-6 13-14	7-8 15-16

The schedule clause/2

dynamic [, chunk]

- ✔ ***Fixed portions of work; size is controlled by the value of chunk***
- ✔ ***When a thread finishes, it starts on the next portion of work***

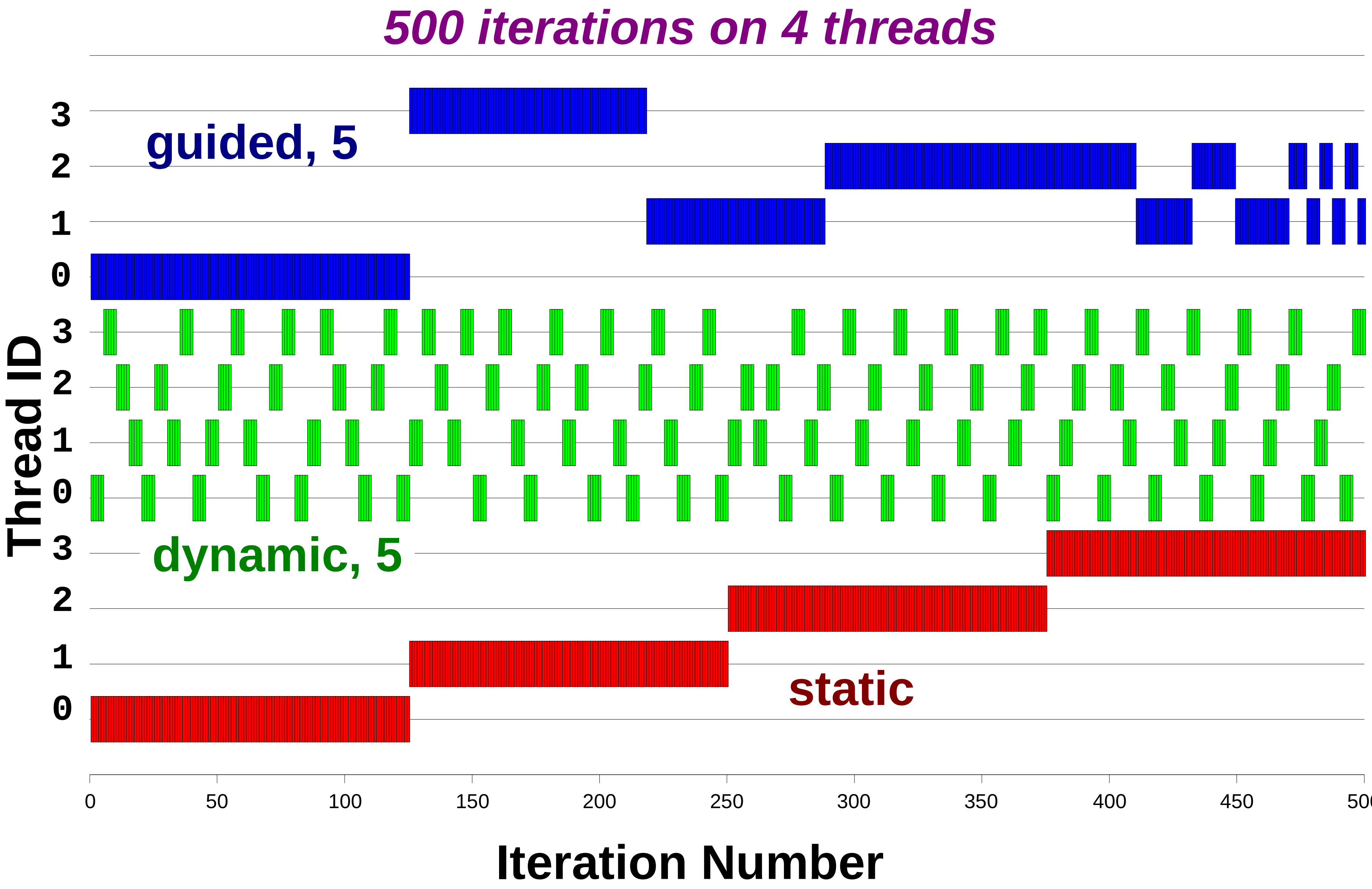
guided [, chunk]

- ✔ ***Same dynamic behavior as "dynamic", but size of the portion of work decreases exponentially***

runtime

- ✔ ***Iteration scheduling scheme is set at runtime through environment variable OMP_SCHEDULE***

The experiment



OpenMP Environment Variables

OpenMP Environment Variables

OpenMP environment variable	Default for Sun OpenMP
OMP_NUM_THREADS n	1
OMP_SCHEDULE "schedule,[chunk]"	static, "N/P"
OMP_DYNAMIC { TRUE | FALSE }	TRUE
OMP_NESTED { TRUE | FALSE }	FALSE

Note: The names are in uppercase, the values are case insensitive

OpenMP Run-time Environment

OpenMP run-time environment

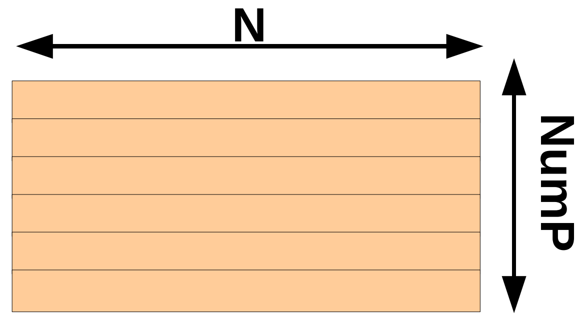
- *OpenMP provides several user-callable functions*
  - *To control and query the parallel environment*
  - *General purpose semaphore/lock routines*
    - ✔ *OpenMP 2.0: supports nested locks*
    - ✔ *Nested locks are not covered in detail here*
- *The run-time functions take precedence over the corresponding environment variables*
- *Recommended to use under control of an #ifdef for _OPENMP (C/C++) or conditional compilation (Fortran)*
- *C/C++ programs need to include <omp.h>*
- *Fortran: may want to use "USE omp_lib"*

Run-time library overview

Name	*Functionality*
omp_set_num_threads	*Set number of threads*
omp_get_num_threads	*Return number of threads in team*
omp_get_max_threads	*Return maximum number of threads*
omp_get_thread_num	*Get thread ID*
omp_get_num_procs	*Return maximum number of processors*
omp_in_parallel	*Check whether in parallel region*
omp_set_dynamic	*Activate dynamic thread adjustment* *(but implementation is free to ignore this)*
omp_get_dynamic	*Check for dynamic thread adjustment*
omp_set_nested	*Activate nested parallelism* *(but implementation is free to ignore this)*
omp_get_nested	*Check for nested parallelism*
omp_get_wtime	*Returns wall clock time*
omp_get_wtick	*Number of seconds between clock ticks*

Example

```
#pragma omp parallel single(...)
   NumP = omp_get_num_threads();

allocate WorkSpace[NumP][N];
#pragma omp parallel for (...)
for (i=0; i < N; i++)
{
    TID = omp_get_thread_num();
     .....

    WorkSpace[TID][i] = .... ;
     .....

    ... = WorkSpace[TID][i];

     .....
}
```



OpenMP locking routines

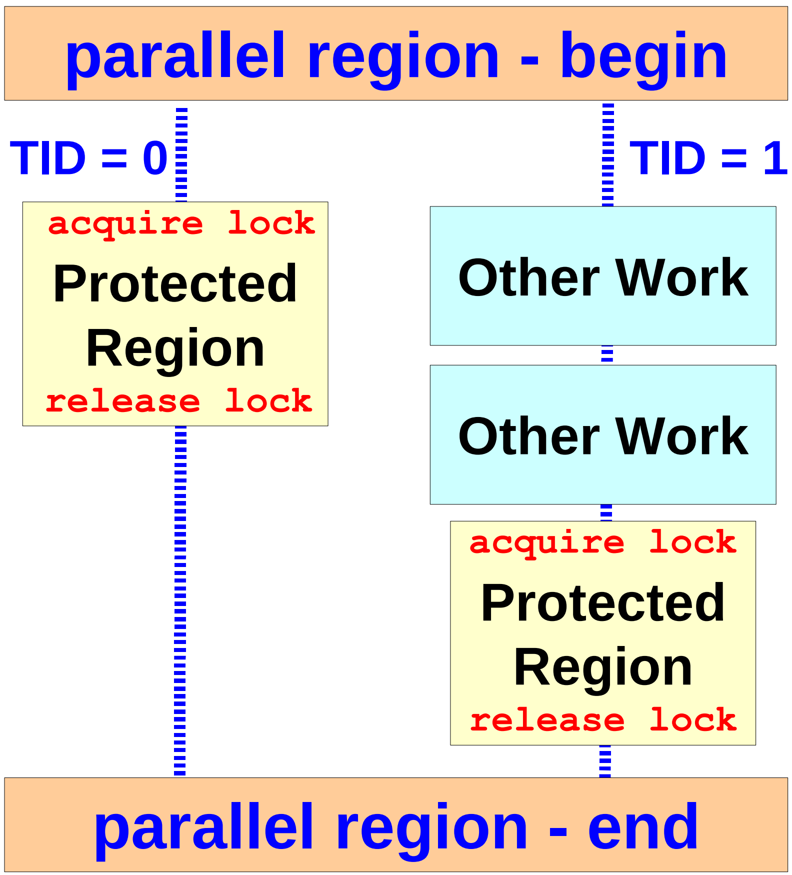
- ***Locks provide greater flexibility over critical sections and atomic updates:***
  - ***Possible to implement asynchronous behavior***
  - ***Not block structured***
- ***The so-called lock variable, is a special variable:***
  - ***Fortran: type INTEGER and of a KIND large enough to hold an address***
  - ***C/C++: type omp_lock_t and omp_nest_lock_t for nested locks***
- ***Lock variables should be manipulated through the API only***
- ***It is illegal, <u>and behavior is undefined</u>, in case a lock variable is used without the appropriate initialization***

Nested locking

- ***Simple locks:*** *may not be locked if already in a locked state*
- ***Nestable locks:*** *may be locked multiple times by the same thread before being unlocked*
- ***In the remainder, we discuss simple locks only***
- ***The interface for functions dealing with nested locks is similar (but using nestable lock variables):***

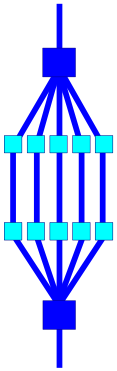
Simple locks	Nestable locks
`omp_init_lock`	`omp_init_nest_lock`
`omp_destroy_lock`	`omp_destroy_nest_lock`
`omp_set_lock`	`omp_set_nest_lock`
`omp_unset_lock`	`omp_unset_nest_lock`
`omp_test_lock`	`omp_test_nest_lock`

OpenMP locking example



- *The protected region contains the update of a shared variable*
- *One thread acquires the lock and performs the update*
- *Meanwhile, the other thread performs some other work*
- *When the lock is released again, the other thread performs the update*

Locking example - The code



```
      Program Locks
            ....
      Call omp_init_lock (LCK)

!$omp parallel shared(LCK)

       Do While ( omp_test_lock (LCK) .EQV. .FALSE. )
          Call Do_Something_Else()
       End Do

       Call Do_Work()

       Call omp_unset_lock (LCK)

!$omp end parallel

      Call omp_destroy_lock (LCK)

      Stop
      End
```

Initialize lock variable

Check availability of lock
(also sets the lock)

Release lock again

Remove lock association

Example output for 2 threads

```
TID:  1 at 09:07:27 => entered parallel region
TID:  1 at 09:07:27 => done with WAIT loop and has the lock
TID:  1 at 09:07:27 => ready to do the parallel work
TID:  1 at 09:07:27 => this will take about 18 seconds
TID:  0 at 09:07:27 => entered parallel region
TID:  0 at 09:07:27 =>   WAIT for lock - will do something else for  5 seconds
TID:  0 at 09:07:32 =>   WAIT for lock - will do something else for  5 seconds
TID:  0 at 09:07:37 =>   WAIT for lock - will do something else for  5 seconds
TID:  0 at 09:07:42 =>   WAIT for lock - will do something else for  5 seconds
TID:  1 at 09:07:45 => done with my work
TID:  1 at 09:07:45 => done with work loop - released the lock
TID:  1 at 09:07:45 => ready to leave the parallel region
TID:  0 at 09:07:47 => done with WAIT loop and has the lock
TID:  0 at 09:07:47 => ready to do the parallel work
TID:  0 at 09:07:47 => this will take about 18 seconds
TID:  0 at 09:08:05 => done with my work
TID:  0 at 09:08:05 => done with work loop - released the lock
TID:  0 at 09:08:05 => ready to leave the parallel region
Done at 09:08:05 - value of SUM is 1100
```

Used to check the answer

Note: program has been instrumented to get this information

Global Data

Global data - An example

```
      program global_data
              ....
      include "global.h"
           ....
!$omp parallel do private(j)
      do j = 1, n
         call suba(j)
      end do
!$omp end parallel do
         ......
```

file global.h

```
common /work/a(m,n),b(m)
```

```
subroutine suba(j)
  .....
include "global.h"
  .....
do i = 1, m
   b(i) = j
end do

do i = 1, m
    a(i,j) = func_call(b(i))
end do

return
end
```

Data Race !

Global data - A Data Race!

Thread 1

```
call suba(1)
```

```
subroutine suba(j=1)

do i = 1, m
   b(i) = 1
end do
       ....
do i = 1, m
 a(i,1)=func_call(b(i))
end do
```

Thread 2

```
call suba(2)
```

```
subroutine suba(j=2)

do i = 1, m
   b(i) = 2
end do
       ....
do i = 1, m
  a(i,2)=func_call(b(i))
end do
```

Shared

Example - Solution

```
      program global_data
              ....
      include "global_ok.h"
            ....
!$omp parallel do private(j)
      do j = 1, n
         call suba(j)
      end do
!$omp end parallel do
        ......
```

file global_ok.h

```
integer, parameter:: nthreads=4
common /work/a(m,n)
common /tprivate/b(m,nthreads)
```

```
subroutine suba(j)
  .....
include "global_ok.h"
  .....

TID = omp_get_thread_num()+1
do i = 1, m
   b(i,TID) = j
end do

do i = 1, m
   a(i,j)=func_call(b(i,TID))
end do

return
end
```

- ***By expanding array B, we can give each thread unique access to it's storage area***
- ***Note that this can also be done using dynamic memory (allocatable, malloc, ...)***

About global data

- ***Global data is shared and requires special care***
- ***A problem may arise in case multiple threads access the same memory section simultaneously:***
  - *Read-only data is no problem*
  - *Updates have to be checked for race conditions*
- ***It is your responsibility to deal with this situation***
- ***In general one can do the following:***
  - *Split the global data into a part that is accessed in serial parts only and a part that is accessed in parallel*
  - *Manually create thread private copies of the latter*
  - *Use the thread ID to access these private copies*
- ***Alternative:* Use OpenMP's threadprivate directive**

The threadprivate directive

- ❑ *OpenMP's threadprivate directive*

```
!$omp threadprivate (/cb/ [,/cb/] ...)
```

```
#pragma omp threadprivate (list)
```

- ❑ *Thread private copies of the designated global variables and common blocks are created*
- ❑ *Several restrictions and rules apply when doing this:*
  - *The number of threads has to remain the same for all the parallel regions (i.e. no dynamic threads)*
  - *Initial data is undefined, unless* ***copyin*** *is used*
  - *......*
- ❑ *Check the documentation when using threadprivate !*

Example - Solution 2

```
      program global_data
             ....
      include "global_ok2.h"
           ....
!$omp parallel do private(j)
      do j = 1, n
         call suba(j)
      end do
!$omp end parallel do
         ......
      stop
      end
```

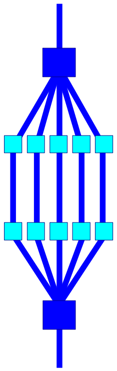
file global_ok2.h

```
common /work/a(m,n)
common /tprivate/b(m)
!$omp threadprivate(/tprivate/)
```

```
subroutine suba(j)
  .....
include "global_ok2.h"
  .....

do i = 1, m
   b(i) = j
end do

do i = 1, m
   a(i,j) = func_call(b(i))
end do

return
end
```

- ☞ ***The compiler creates thread private copies of array B, to give each thread unique access to it's storage area***
- ☞ ***Note that the number of copies is automatically adjusted to the number of threads***

The copyin clause



copyin (list)

- ✔ ***Applies to THREADPRIVATE common blocks only***
- ✔ ***At the start of the parallel region, data of the master thread is copied to the thread private copies***

Example:

```
      common /cblock/velocity
      common /fields/xfield, yfield, zfield

! create thread private common blocks

!$omp threadprivate (/cblock/, /fields/)

!$omp parallel             &
!$omp default (private)  &
!$omp copyin ( /cblock/, zfield )
```

Wrap-up OpenMP 2.5

- ❑ *OpenMP provides for a small, but yet powerful, programming model*
- ❑ *It can be used on a shared memory system of any size*
  - *This includes a single socket multicore system*
- ❑ *Compilers with OpenMP support are widely available*
- ❑ *OpenMP 3.0 (briefly covered later):*
  - *Extends the language with the tasking model*
    - ✔ *This allows to parallelize less regular constructs*
    - ✔ *Adds tremendous flexibility*
  - *Various additional features introduced*
  - *Addresses several gaps in OpenMP 2.5*

What's New in OpenMP 3.0

Particularly, the New OpenMP Tasking Model

Federico Massaioli (`federico.massaioli@caspur.it`)

CASPUR e Università degli Studi di Roma "La Sapienza"
Laurea Magistrale in Informatica
Anno accademico 2008-2009

OpenMP 3.0

- 2÷3 improvements
  loop collapse, STATIC schedule reuse,
  AUTO schedule (allows runtimes to dynamically adapt)
- 7 fixes
  internal control variables, unsigned int in for
  stack size control, thread wait policy,
  storage reuse (forbidden!), constructor/destructor issues,
  allocatable arrays and Fortran pointers
- 2÷3 new mistakes
  AUTO schedule (will vendors care?),
  an incredible amount of new API calls
  a complex description of the memory model
- 1 new big thing: TASKS!!!

Loop collapse

```
#pragma omp for collapse(3)
for(i = 1; k < nx(ib); ++i)
  for(j = 1; j < ny(ib); ++j)
    for(k = 1; k < nz(ib); ++k) {
      …
    }
```

- Multiblock, CFD code
- For many interposed blocks, one, two or all directions have not enough points to parallelize
- Changing from block to block

STATIC schedule reuse

```
#pragma omp parallel
{
  #pragma omp for schedule(static)
  for(i = 1; i < N; ++i)      will get the same schedule
    //…
  //…
 #pragma omp for schedule(static)
  for(i = 1; i < N; ++i)
    //…
}
//…
#pragma omp parallel
{
 #pragma omp for schedule(static)
  for(i = 1; i < N; ++i)      in general, will get a different one
    //…
  //…
}
```

Why tasks?

- According to the Specs, OpenMP 1.0 – 2.5 "is somewhat tailored for large array-based applications"
- Arrays: not the fanciest and coolest data structures, not even in STC and HPC
- Arrays
  → mostly regular forms of parallelism
- Lists, trees, …
  → frequently irregular forms of parallelism
- Lists of lists, hash tables, dequeues, btrees, …
  → lots of very well hidden concurrency
- "Threads are so 1990s! Geez!!!"
- But is this new tasking stuff really needed?

Pointer chasing in OpenMP 2.5?

```
for(p = list; p; p = p->next) {
  process(p->item);
}
```

- Doesn't suit an **`omp for`**: amount of iterations not known in advance
- Transformation to a "canonical" loop can be very labour-intensive/inefficient

`omp single nowait` is our friend

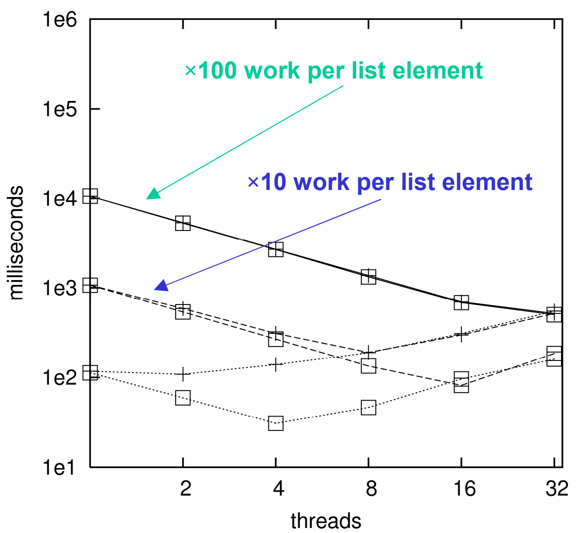
```
#pragma omp parallel private(p)
{
  for(p = list; p; p = p->next)
    #pragma omp single nowait
      process(p->item);
}
```

- Each thread redundantly iterates through the loop
- For each single value of **p**, only one thread is allowed to enter the single construct
- Very few people realize this
- Compiler/runtime developers sometimes among them, unfortunately

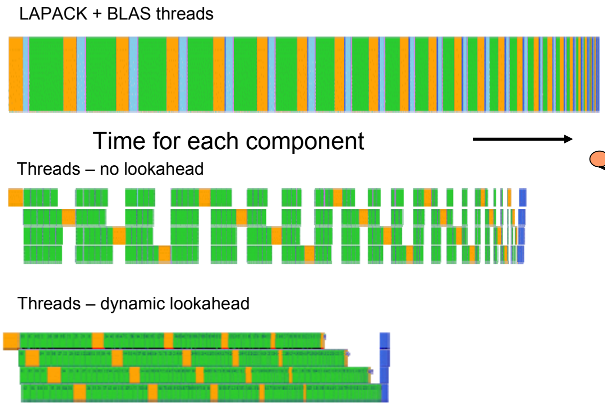
Some experimental measurements

□ **`omp for`**
(including time to build an array of pointers)

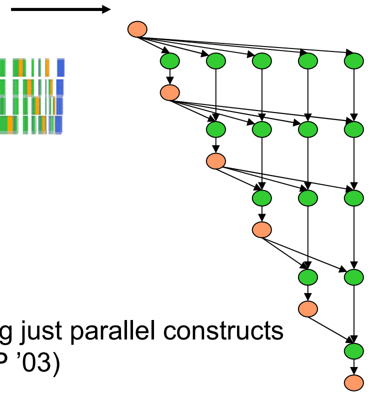
+ **`omp single nowait`**



LU factorization, dynamic lookahead



(Images courtesy of J. Dongarra)



Actually implemented in OpenMP using just parallel constructs and barriers (S. Salvini, NAG, EWOMP '03)

Something unfeasible

```
void preorder(node *p) {
  process(p->data);
  #pragma omp parallel num_threads(2)
  {
    if (p->left)
      #pragma omp single nowait
        preorder(p->left);
    if (p->right)
      #pragma omp single nowait
        preorder(p->right);
  }
}
```

- Because worksharing constructs can't be closely nested
- And stressing nested parallelism so much is not a good idea…

More good reasons

- Multiblock grids on complex topologies, multiresolution grids, immersed grids, AMR
- Fluid-structure interactions in presence of moving parts
- Agent based models
  - immunology
  - financial markets simulations
- Complex bodies, hierarchical assemblies of moving parts
  - robotics
  - manoeuvering
- Interaction of many different components
  - SPICE3, T. Weng, R. Perng, B. Chapman, IWOMP '06

Enter OpenMP tasks

- Separation of threads and what threads execute
- Threads are execution contexts
- Tasks are well defined units of work
- Threads execute tasks, from start to end
- Add true concurrency to parallelism

First and foremost tasking construct

```
#pragma omp parallel [clauses]
{
  // structured block
}
```

- Creates both threads and tasks
- These tasks are “implicit”
- Each one is immediately executed by one thread
- Each of them is tied to the assigned thread

- Did you ever think of it this way?

New tasking construct

```
#pragma omp task [clauses]
{
  // structured block
}
```

where *clauses* is zero or more of:

`private, firstprivate, shared, default, if, untied`

- Immediately creates a new task but no new thread
- This task is “explicit”
- It will be executed by a thread in the current team
- It can be deferred until a thread is available to execute
- The data environment is built at creation time

- It isn’t a worksharing construct!!

How many tasks are created?

```
#pragma omp parallel
{
  // …
  #pragma omp task
  {
    // …
  }
  //…
}
```

- A new one whenever a threads encounters a task construct
- In this case, as many as there are threads in the team
- Thus, # total tasks = 2*omp_get_thread_num()
- It's symmetrical with **`parallel`** directive
- It's powerful

- It's ovbious: it isn't a worksharing construct!!

Load balancing on lists with tasks

```
#pragma omp parallel private (p)
{
  #pragma omp for
  for (i=0; i<num_lists; i++) {
    p = listheads[i];
    while(p) {
      #pragma omp task firstprivate(p)
        process(p);
      p=next(p);
    }
  }
}
```

- **`sections`** will also do
- It's possible: it isn't a worksharing construct!!

And if you have a lonely list…

```
#pragma omp parallel
  #pragma omp single
  {
     p = listhead;
     while(p) {
       #pragma omp task firstprivate(p)
         process(p);
       p=next(p);
     }
  }
```

- **`master`** will also do
- It can be done: it isn't a worksharing construct!!

If you have recursive data structures

```
void preorder(node *p) {
  process(p->data);
  if (p->left)
    #pragma omp task
      preorder(p->left);
  if (p->right)
    #pragma omp task
      preorder(p->right);
}
```

- It's easy: it isn't a worksharing construct!!
- But what about postorder traversal?

Postorder tree traversal

```
void postorder(node *p) {
  if (p->left)
    #pragma omp task
      postorder(p->left);
  if (p->right)
    #pragma omp task
      postorder(p->right);
  #pragma omp taskwait    <- suspend point

  process(p->data);
}
```

- Parent task suspended until children tasks complete

Postorder or parallel tree traversal

```
void traverse(node *p, bool postorder) {
  if (p->left)
    #pragma omp task
      traverse(p->left);
  if (p->right)
    #pragma omp task
      traverse(p->right);
  if (postorder) {
    #pragma omp taskwait    <- suspend point
  }
  process(p->data);
}
```

- Parent task suspended until children tasks complete
- Unstructured directive, trades optimization for flexibility

When/where explicit tasks complete?

- At **`#pragma omp taskwait`**
  - applies only to tasks generated in the current task, not to “descendants”
  - structured flavour (**`#pragma omp taskgroup`**) deferred to 3.1 release, in doubt it should be “shallow” like **`taskwait`**, or “deep”
- At **`#pragma omp barrier`**
  - applies to all tasks generated in the current parallel region up to the barrier
  - matches user expectation
  - obviously applies to implicit barriers too

Data scoping in explicit tasks

- **`private`** and **`firstprivate`**: business as usual
- **`shared`**: same business, from a new perspective
  - shared among all tasks (“horizontal”)
  - shared among a task and a descendant (“vertical”)
  - different synchronizations are required in the two cases
- Most implicitly determined variables: firstprivate!
  - safety by default, programmers have full control
  - spares programmers a lot of keystrokes
  - can be altered with a **`default`** clause

“Vertical” sharing

```
void fibonacci(int n) {
  int i, j;
  if (n<2) return n;

  #pragma omp task shared(i)
    i = fibonacci(n-1);
  #pragma omp task shared(j)
    j = fibonacci(n-2);
  #pragma omp taskwait

  return i+j;
}
```

synchronization

- Allow results to be returned to parent

Enter *task switching*

- What: the ability of a thread to suspend a task and execute another one before resuming
- Where:
  - at *task scheduling points:* **`task`**, **`taskwait`**, **`barrier`** directives, and implicit barriers
  - at a **`taskyield`** construct deferred to OpenMP 3.1
- When:
  - whenever is needed or useful
  - up to the implementation
- Why:
  - to lift pressure on runtime data structures
  - because it can’t be dispensed with completely
- Consequence: locks owned by tasks!

Lifting pressure on runtime

```
#pragma omp single
{
  for (i=0; i<ONEZILLION; i++)
    #pragma omp task
      process(item[i]);
}
```

- Too many tasks generated in an eye-blink
- Generating task will have to suspend for a while
- With task switching, the executing thread can:
  - execute an already generated task (draining the "*task pool*")
  - dive into the encountered task (could be very cache-friendly)

There's no life without task switching

```
#pragma omp parallel
{
    #pragma omp task
      // first task
    // …
    #pragma omp task
      // n-th task
  #pragma omp taskwait
  // some more tasks
}
```

- Without task switching, no threads available to execute the tasks that are waited for
- Without task switching, a deadlock is granted at **`taskwait`**!!!

Enter *thread switching*

```
#pragma omp single
{
  #pragma omp task untied
    for (i=0; i<ONEZILLION; i++)
      #pragma omp task
        process(item[i]);
}
```

- Eventually, too many tasks are generated
- Generating task is suspended and executing thread switches to a long and boring task
- Other threads get rid of all already generated tasks, and start starving…
- With thread switching, the generating task can be resumed by a different thread, and starvation is over
- Too new to be the default, the programmer is responsible!
- Task/thread switching can happen anywhere in an untied task

Who's afraid of task switching?

```
int tp;
#pragma omp threadprivate(tp)
int var;
void work() {
  #pragma omp task
  {
    // do work here…
    #pragma omp task
    {
      tp = 1;
      // do work here…
      #pragma omp task
        // no modification of tp here…
      var = tp; //value of tp can be 1 or 2
    }
    tp = 2;
  }
}
```

- If a thread switches task, a race among tasks could happen
- Look at scheduling restrictions to understand if usage is safe or not

The `if` clause

- When the **`if`** clause argument is false
  - the encountered task is executed immediately by the encountering thread, and the enclosing task is suspended up to its end
  - the data environment is still local to the new task
  - and it's still a different task wrt. synchronization
- It's a user directed optimization
  - when the cost of the task is comparable to the runtime overhead
  - to control cache and memory affinity

And now for something more real…

- Writing a spec without experimental proof is bad
- Our friends in Barcelona wrote a simplified implementation
  - no thread switching
  - no if clause
  - no task-specific optimizations
- Q#1: are tasks expressive enough?
  - yes, but some problems with reductions and capturing pointees, good food for OpenMP 3.1
- Q#2: are tasks able to perform?
  - let's have a look

Evaluation

- Run on a SGI Altix 4700 with 128 procs
- Comparison between:
  - Worksharing version (using icc)
  - Nested version (using icc)
  - OpenMp tasks (using mcc+icc)
  - Intel Taskqueues (using icc)
  - Cilk (uses gcc)
- Baseline for speedups is serial time

sparseLU

- blocked matrix, blocks can be empty

```
for (kk=0; kk<NB; kk++) {
    lu0(A[kk][kk]);
    for (jj=kk+1; jj<NB; jj++)
      if (A[kk][jj] != NULL)
          fwd(A[kk][kk], A[kk][jj]);
    for (ii=kk+1; ii<NB; ii++)
      if (A[ii][kk] != NULL)
        bdiv (A[kk][kk], A[ii][kk]);
    for (ii=kk+1; ii<NB; ii++) {
      if (A[ii][kk] != NULL)
        for (jj=kk+1; jj<NB; jj++)
          if (A[kk][jj] != NULL) {
            if (A[ii][jj]==NULL) A[ii][jj]=allocate_clean_block();
            bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
          }
      }
}
```

Parallelize the outer loops
- heavy imbalance

Parallelize the inner loops
- grain too small
- high overhead

sparseLU, the tasking way

```
for (kk=0; kk<NB; kk++) {
      lu0(A[kk][kk]);
      for (jj=kk+1; jj<NB; jj++)
        if (A[kk][jj] != NULL)
#pragma omp task
            fwd(A[kk][kk], A[kk][jj]);
      for (ii=kk+1; ii<NB; ii++)
        if (A[ii][kk] != NULL)
#pragma omp task
          bdiv (A[kk][kk], A[ii][kk]);
#pragma omp taskwait
      for (ii=kk+1; ii<NB; ii++) {
        if (A[ii][kk] != NULL)
          for (jj=kk+1; jj<NB; jj++)
            if (A[kk][jj] != NULL) {
              if (A[ii][jj]==NULL) A[ii][jj]=allocate_clean_block();
#pragma omp task
              bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
            }
        }
#pragma omp taskwait
}
```

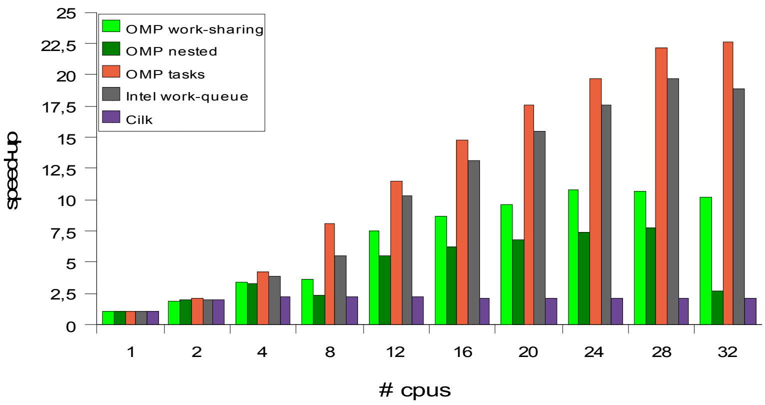
Only spawn work for non-empty blocks
- better balance

Synchronization across phases

They could be removed if there was support for explicit dependencies between tasks

SparseLU evaluation

sparseLU (50 blocks, 100x100 elements)



Protein sequences alignment

blocks can be empty

```
for ( si = 0 ; si < nseqs ; si+) {
    len1 = compute_sequence_length( si +1);
    /* compa re to the other sequences */
    for ( sj = si +1 ; sj < nseqs ; sj++ ) {
        len2 = compute_sequence_length ( sj +1);
        compute_score_penalties ( . . . ) ;
        forward_pass ( . . . ) ;
        reverse_pass ( . . . ) ;
        diff ( . .. ) ;
        mm_score = trace_path ( . . . ) ;
        i f ( len1 == 0 | | len2 == 0 ) mm_score=0.0 ;
        else mm_score /= ( double ) MIN( len1 , len2 ) ;
        print_score();
    }
}
```

Parallelize the outer loops
- phases depend on the length of sequences
- heavy imbalance

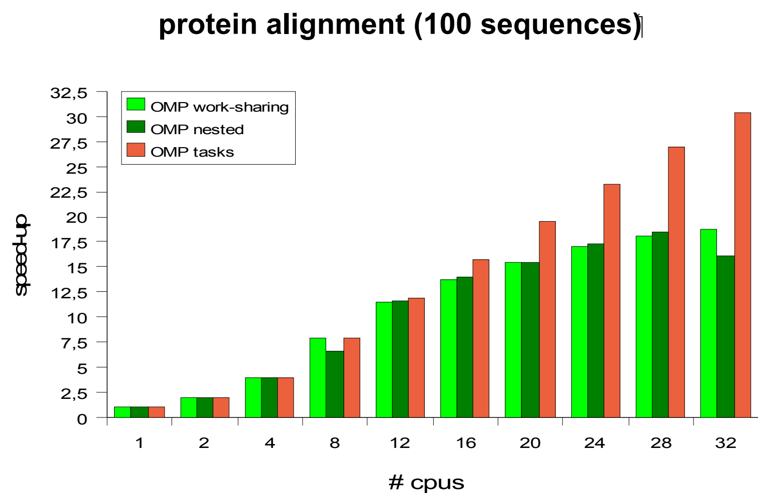
Parallelize the inner loop
- higher overhead

Protein alignment

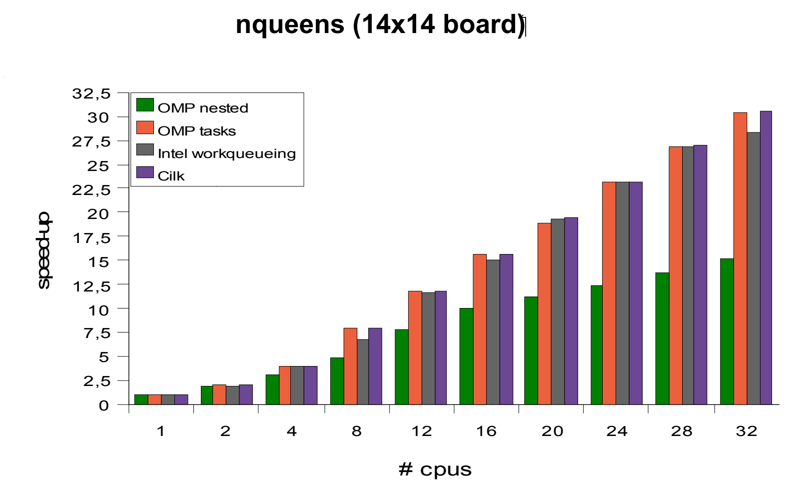
```
#pragma omp parallel for
for ( si = 0 ; si < nseqs ; si+) {
    len1 = compute_sequence_length( si +1);
    /* compa re to the other sequences */
    for ( sj = si +1 ; sj < nseqs ; sj++ ) {
#pragma omp task {
        len2 = compute_sequence_length ( sj +1);
        compute_score_penalties ( . . . ) ;
        forward_pass ( . . . ) ;
        reverse_pass ( . . . ) ;
        diff ( . .. ) ;
        mm_score = trace_path ( . . . ) ;
        i f ( len1 == 0 | | len2 == 0 ) mm_score=0.0 ;
        else mm_score /= ( double ) MIN( len1 , len2 ) ;
        print_score();
      }
    }
}
```

Combines data and task parallelism

Protein alignment: evaluation

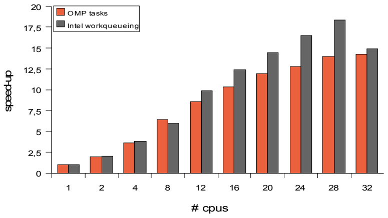


N-Queens: evaluation

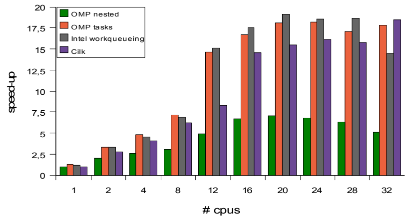


Other experiments

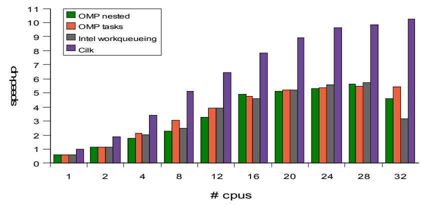
strassen (1280x1280 matrix)



FFT (32MB of complex)

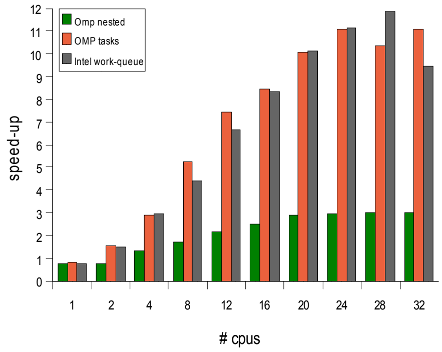


multisort (32MB of ints)



An unfortunate case

floorplan (20 cells)



connected components (500000 nodes,100000 edges)

- Very big slowdown
  - we couldn't complete most runs
- Granularity is very fine

Conclusions

- The new OpenMP tasks allow to express a lot of irregular parallelism
- Some issues can be improved in the language
  - reductions, data capturing, dependencies, ...
- Performance-wise, with a prototype, surpasses all 2.5 versions
  - encouraging results against similar models