OpenMP

Jesús Labarta

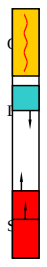
OpenMP

- **Introductory example**
- **Directives**
  - Parallel regions
  - Worksharing
  - sincronization
  - Data scope
- **Run time calls**
- **OpenMP 3.0**
  - Tasks
  - schedule
- **Implementation and system issues**

OpenMP: introduction

- Standard promoted by main manufacturers
  - http://www.openmp.org , http://www.compunity.org
  - Fortran
    - ✓ V1.0: Oct. 1997
    - ✓ V2.0: Nov. 2000
  - C
    - ✓ V1.0: Oct. 1998
    - ✓ V2.0: March 2002
  - V2.5: May 2005
  - V3.0: draft october 2007
  - Structure: Directives, clauses and run time calls

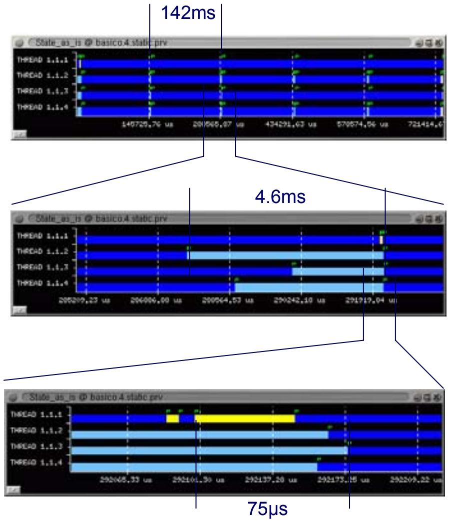
Basic example



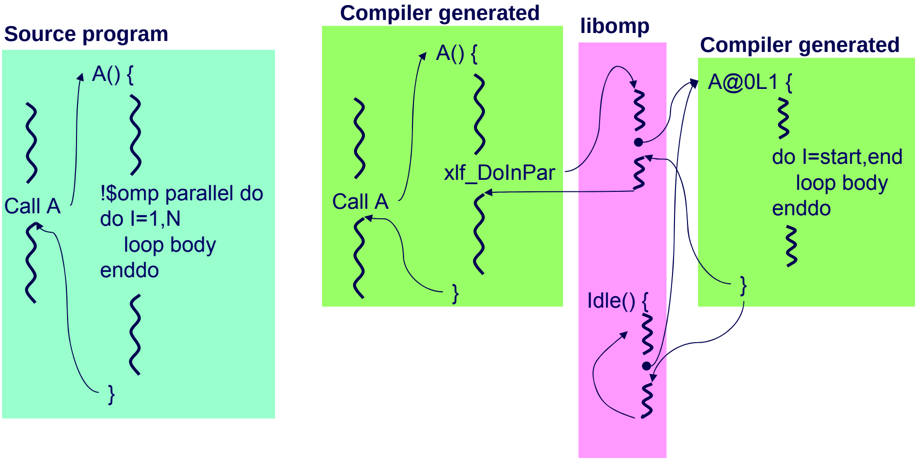
```
PROGRAM test
PARAMETER (N=1024)
REAL dummy(N), factor
REAL a(64000)
INTEGER i,iter,time
common /varios/a

factor=1/1.0000001
time = 10

DO iter=1,5


   DO i=1,N
      dummy(i)= dummy(i)*factor
      call delay(time)
   ENDDO
ENDDO

END
```

Basic example

```
      PROGRAM test
      PARAMETER (N=1024)
      REAL dummy(N), factor
      REAL a(64000)
      INTEGER i,iter,time
      common /varios/a

      factor=1/1.0000001
      time = 10

      DO iter=1,5
C$OMP PARALLELDO SCHEDULE(STATIC)
C$OMP+ SHARED(dummy) PRIVATE(I)
         DO i=1,N
            dummy(i)= dummy(i)*factor
            call delay(time)
         ENDDO
      ENDDO

      END
```



OpenMP compilation and Run Time



Source program

A() {

Call A

!$omp parallel do
do I=1,N
  loop body
enddo

}

Compiler generated

A() {

xlf_DoInPar

Call A

}

libomp

Idle() {

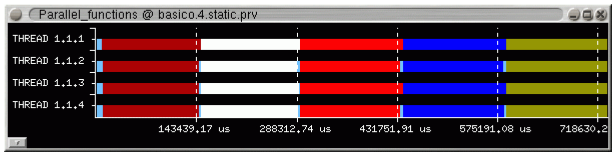
Compiler generated

A@0L1 {

do I=start,end
  loop body
enddo

}

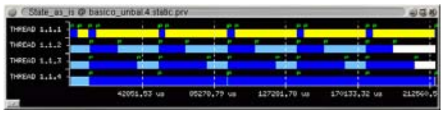
OpenMP outlined routines

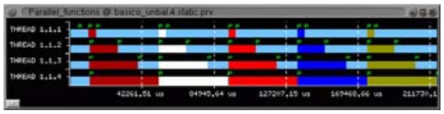


- **Display of the identifier (color encoded) of the outline routine being executed**
- **The compiler unrolls the iter loop !!**

Unbalanced loop

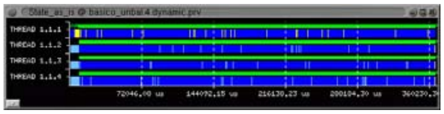
```
      PROGRAM test
      PARAMETER (N=1024)
      REAL dummy(N), factor
      REAL a(64000)
      INTEGER i,iter,time
      common /varios/a

      factor=1/1.0000001

      DO iter=1,5
C$OMP PARALLELDO SCHEDULE(STATIC)
C$OMP+ SHARED(dummy) PRIVATE(I,time)
         DO i=1,N
            dummy(i)= dummy(i)*factor
            time = i/100
            call delay(time)
         ENDDO
      ENDDO

      END
```

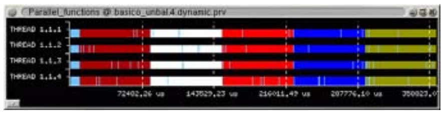


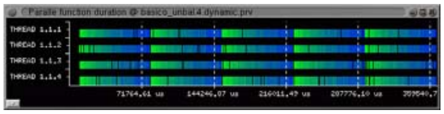


Dynamic scheduling

```
      PROGRAM test
      PARAMETER (N=1024)
      REAL dummy(N), factor
      REAL a(64000)
      INTEGER i,iter,time
      common /varios/a

      factor=1/1.0000001

      DO iter=1,5
C$OMP PARALLELDO SCHEDULE(DYNAMIC)
C$OMP+ SHARED(dummy) PRIVATE(I,time)
         DO i=1,N
            dummy(i)= dummy(i)*factor
            time = i/100
            call delay(time)
         ENDDO
      ENDDO

      END
```

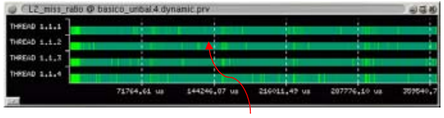






Outlined routine duration

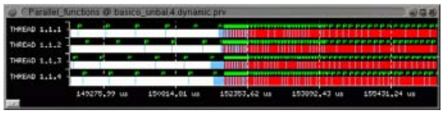
L2 miss ratio

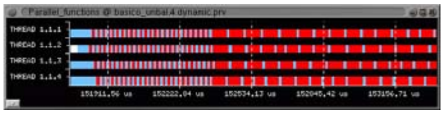


Avg. 2.3%

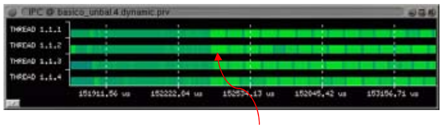
Dynamic scheduling

```
      PROGRAM test
      PARAMETER (N=1024)
      REAL dummy(N), factor, var
      REAL a(64000)
      INTEGER i,iter,time
      common /varios/a

      factor=1/1.0000001

      DO iter=1,5
C$OMP PARALLELDO SCHEDULE(DYNAMIC)
C$OMP+ SHARED(dummy) PRIVATE(i,time)
         DO i=1,N
            dummy(i)= dummy(i)*factor
            time = i/100
            call delay(time)
         ENDDO
      ENDDO

      END
```





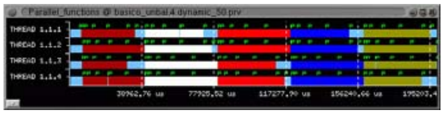
IPC

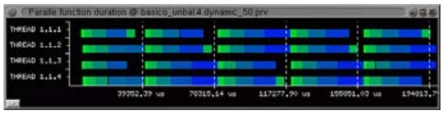


0.25

Coarser Grain Dynamic

```
      PROGRAM test
      PARAMETER (N=1024)
      REAL dummy(N), factor, var
      REAL a(64000)
      INTEGER i,iter,time
      common /varios/a

      factor=1/1.0000001

      DO iter=1,5
C$OMP PARALLELDO SCHEDULE(DYNAMIC,50)
C$OMP+ SHARED(dummy) PRIVATE(i,time)
         DO i=1,N
            dummy(i)= dummy(i)*factor
            time = i/100
            call delay(time)
         ENDDO
      ENDDO

      END
```



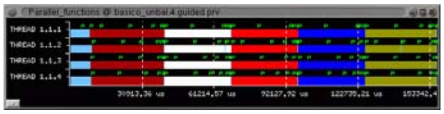


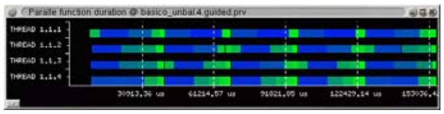
Parallel function duration

- **Less overhead**
- **Some imbalance:**
    - Heavy chunks towards the end

Guided Scheduling

```
      PROGRAM test
      PARAMETER (N=1024)
      REAL dummy(N), factor
      REAL a(64000)
      INTEGER i,iter,time
      common /varios/a

      factor=1/1.0000001

      DO iter=1,5
C$OMP PARALLELDO SCHEDULE(GUIDED)
C$OMP+ SHARED(dummy) PRIVATE(I,time)
         DO i=1,N
            dummy(i)= dummy(i)*factor
            time = i/100
            call delay(time)
         ENDDO
      ENDDO

      END
```



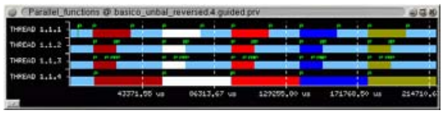


Parallel function duration

- **Less overhead**
- **Good load balance:**
    - Heavy chunks towards the beginning
- **Dynamic:**
    - Non repetitive pattern

Always possible to fool a schedule

```
      PROGRAM test
      PARAMETER (N=1024)
      REAL dummy(N), factor
      REAL a(64000)
      INTEGER i,iter,time
      common /varios/a

      factor=1/1.0000001

      DO iter=1,5
C$OMP PARALLELDO SCHEDULE(GUIDED)
C$OMP+ SHARED(dummy) PRIVATE(I,time)
         DO i=1,N
            dummy(i)= dummy(i)*factor
            time = (N-i)/100
            call delay(time)
         ENDDO
      ENDDO

      END
```

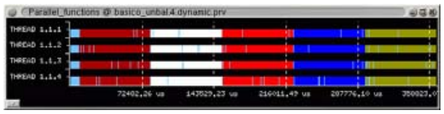


Parallel function duration

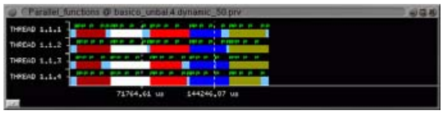
- **Dynamic:**
  - Repetitive pattern (not enforced)

Comparison

- **Dynamic**



- **Dynamic,50**



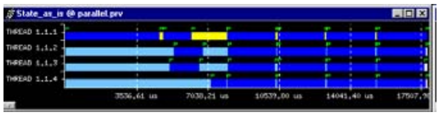
- **Guided**

Same scale

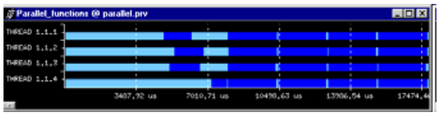
Back to basics: Parallel directive

```
      DO iter=1,5

C$OMP PARALLEL PRIVATE(time)
      time = 10
      call delay(time)
C$OMP ENDPARALLEL

      ENDDO
```

State



Parallel functions



- **Team**
  - Threads to work on a parallel
- **Parallel**
  - All threads execute the body

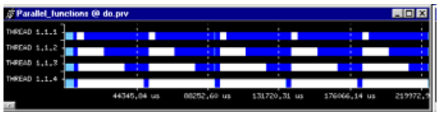
Worksharings

- **Directives within parallel**
- **Switch from replicated work to partitioned work**
- **Work sharing constructs**
  - loops
    - ✓ `C$OMP [END] DO [clause[[,] clause]…]`
    - ✓ `C$OMP [END] PARALLEL DO …`
  - sections
    - ✓ `C$OMP [END] SECTIONS [clause[[,] clause]…]`
    - ✓ `C$OMP SECTION`
    - ✓ `C$OMP [END] PARALLEL SECTIONS …`

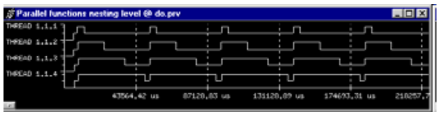
Do

```
      DO iter=1,5

C$OMP PARALLEL PRIVATE(I,time)
      time = 50
      call delay(time)
C$OMP DO SCHEDULE (STATIC)
      DO i=1,N
         time = i/100
         call delay(time)
      ENDDO
C$OMP ENDPARALLEL

      ENDDO
```

Parallel function



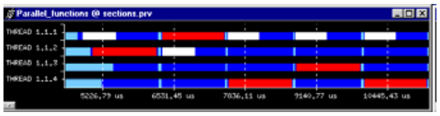
Parallel function nesting level



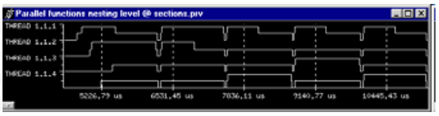
Sections

```
      DO iter=1,5

C$OMP PARALLEL PRIVATE(time)
C$OMP SECTIONS
C$OMP SECTION
      time = 30
      call delay(time)
C$OMP SECTION
      time = 60
      call delay(time)
C$OMP END SECTIONS
C$OMP ENDPARALLEL

      ENDDO
```

Parallel function



Parallel function nesting level



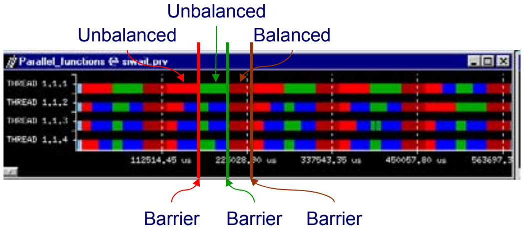
- **Dynamic:**
  - Sections randomly taken by threads

Shortcuts

- **Parallel do**
  - Shortcut for do within parallel
  - Allows more efficient implementation
- **Parallel sections**
  - Shortcut for sections within parallel
  - Allows more efficient implementation

Relaxing synchronizations

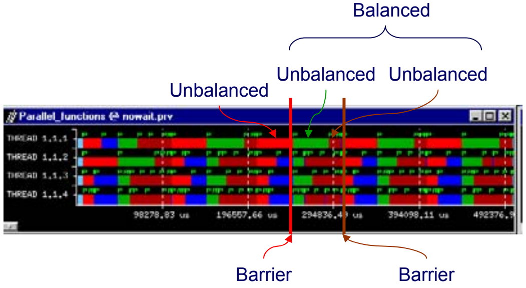
```
      DO iter=1,5
C$OMP PARALLEL SHARED(dummy) PRIVATE(I,J, time)
C$OMP DO SCHEDULE(GUIDED)
      DO i=1,N
         time = (N-i)/100
         call delay(time)
      ENDDO
C$OMP DO SCHEDULE(GUIDED)
      DO i=1,N
         time = (N-i)/100
         call delay(time)
      ENDDO
C$OMP ENDDO
C$OMP DO SCHEDULE(GUIDED)
      DO i=1,N
         time = i/100
         call delay(time)
      ENDDO
C$OMP END PARALLEL
      ENDDO
```



- **Globally**
  - 2 unbalanced loops

Relaxing synchronizations: Nowait

```
      DO iter=1,5
C$OMP PARALLEL SHARED(dummy) PRIVATE(I,J, time)
C$OMP DO SCHEDULE(GUIDED)
      DO i=1,N
         time = (N-i)/100
         call delay(time)
      ENDDO
C$OMP DO SCHEDULE(GUIDED)
      DO i=1,N
         time = (N-i)/100
         call delay(time)
      ENDDO
C$OMP ENDDO NOWAIT
C$OMP DO SCHEDULE(GUIDED)
      DO i=1,N
         time = i/100
         call delay(time)
      ENDDO
C$OMP END PARALLEL
      ENDDO
```



- **Globally**
  - Achieved global balance

Other computation patterns

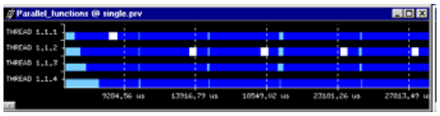
- **Master**
  - `C$OMP [END] MASTER`
  - Only master threads executes. All other wait
- **Single**
  - `C$OMP [END] SINGLE [clause[[,] clause]…]`
  - One thread executes
  - Barrier at end
- **Ordered**
  - `C$OMP [END] ORDERED`
  - Ensures execution in sequential order

Other computation patterns

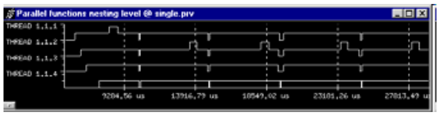
- **Critical**
  - `C$OMP [END] CRITICAL [(name)]`
  - Mutual exclusion
- **Atomic**
  - `C$OMP ATOMIC`
  - Atomicity of single assignment statement
    - ✓ Optimizable implementation
- **Barrier**
  - `C$OMP BARRIER`
  - All threads must reach it before continuing
- **Flush**
  - `C$OMP FLUSH [(list)]`
  - Enforce consistency

Single

Parallel function



Parallel function nesting level



```
      DO iter=1,5

C$OMP PARALLEL PRIVATE(time)
      time = 50
      call delay(time)
C$OMP SINGLE
      time = 25
      call delay(time)
C$OMP END SINGLE
      time = 10
      call delay(time)

C$OMP ENDPARALLEL

      ENDDO
```

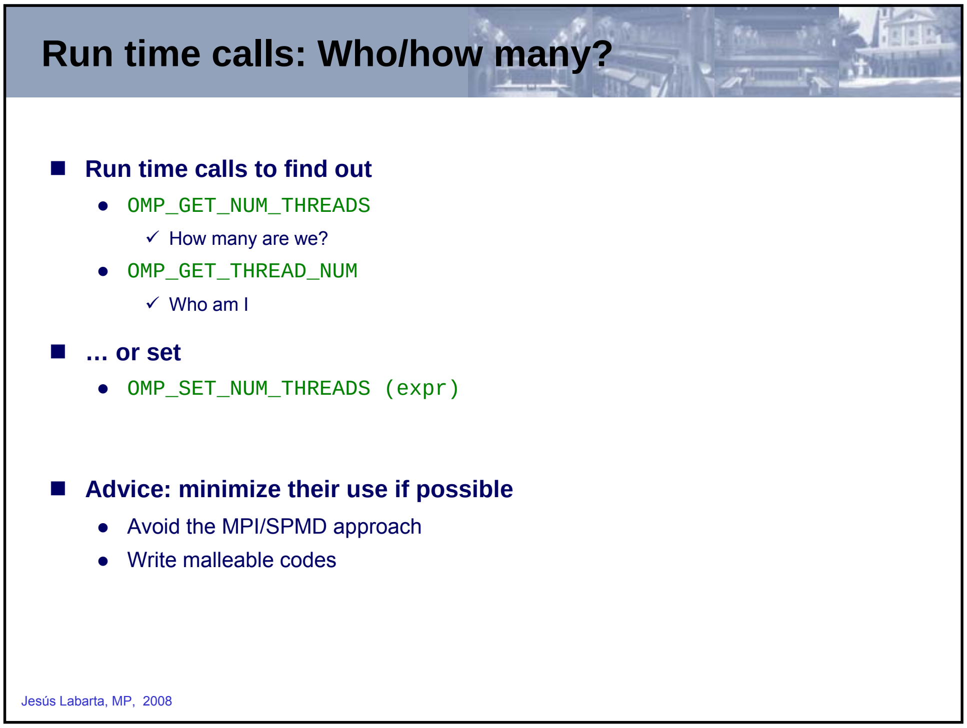
- **Implementation**
  - outlined call
  - Other alternatives possible
- **Duration**
  - Not proportional to time !!!
  - ???

Data scope

- **`PRIVATE(list)`**
  - ≈ allocate local space on entry to outlined routine
  - Scalars / vectors
- **`FIRSTPRIVATE(list)`**
  - Allocate and initialize
- **`LASTPRIVATE(list)`**
  - Copy value of last iteration to global
- **`THREADPRIVATE(list)`**
  - Privatization of common blocks

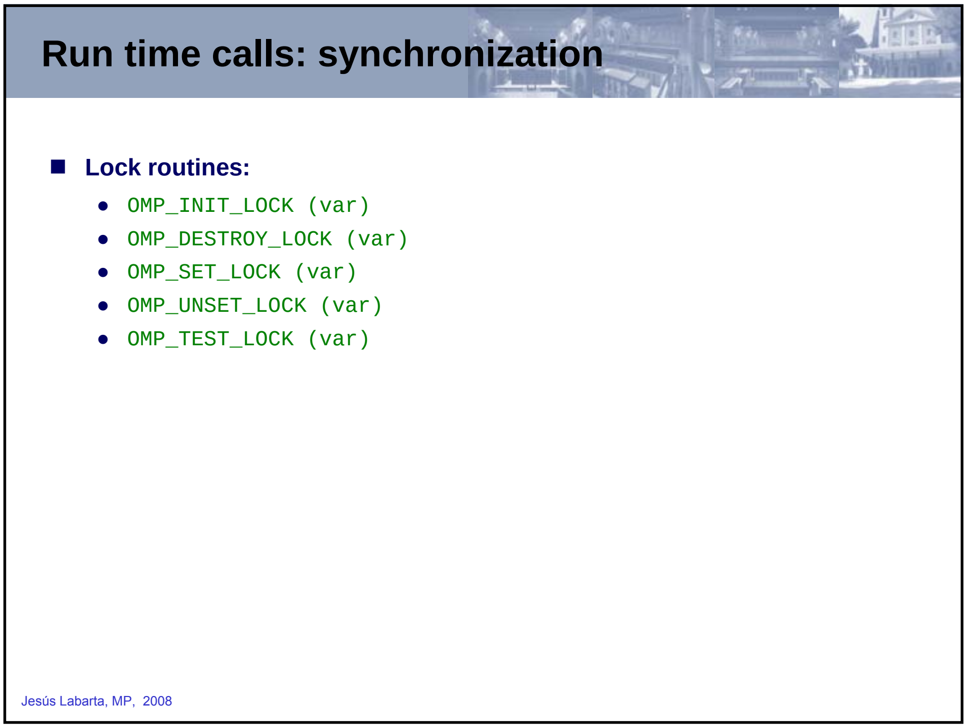
Data scope

- **`REDUCTION(list)`**
  - Perform reduction operation
  - List: op:var
    - ✓ i.e: reduction (+:diff)



Run time calls: Who/how many?

- **Run time calls to find out**
  - `OMP_GET_NUM_THREADS`
    - ✓ How many are we?
  - `OMP_GET_THREAD_NUM`
    - ✓ Who am I
- **… or set**
  - `OMP_SET_NUM_THREADS (expr)`
- **Advice: minimize their use if possible**
  - Avoid the MPI/SPMD approach
  - Write malleable codes



Run time calls: synchronization

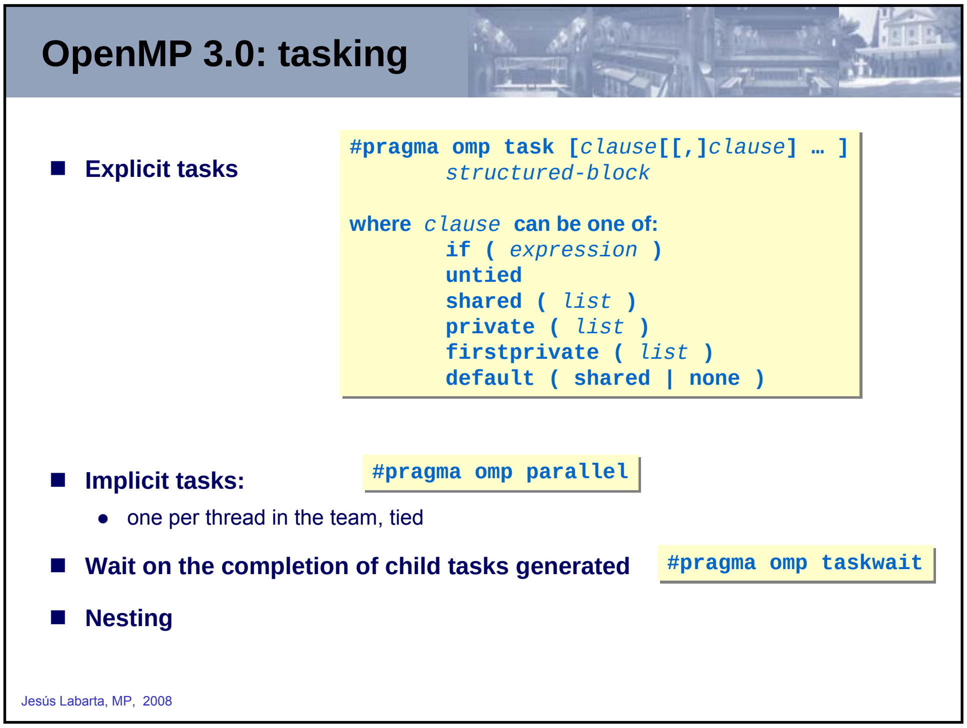
- **Lock routines:**
  - `OMP_INIT_LOCK (var)`
  - `OMP_DESTROY_LOCK (var)`
  - `OMP_SET_LOCK (var)`
  - `OMP_UNSET_LOCK (var)`
  - `OMP_TEST_LOCK (var)`

Environment variables

- **OMP_NUM_THREADS**
- **OMP_SCHEDULE**

Other topics

- **OpenMP2.0**
  - Array worksharings
  - Clarification of consistency model
  - More on nested Parallelism
- **OpenMP 3.0**
  - Nested parallelism
  - Tasks
  - Further schedules
  - Multiprogrammed workload management
- **Future ?**
  - Precedences



OpenMP 3.0: tasking

- **Explicit tasks**

```
#pragma omp task [clause[[,]clause] … ]
        structured-block

where clause can be one of:
        if ( expression )
        untied
        shared ( list )
        private ( list )
        firstprivate ( list )
        default ( shared | none )
```

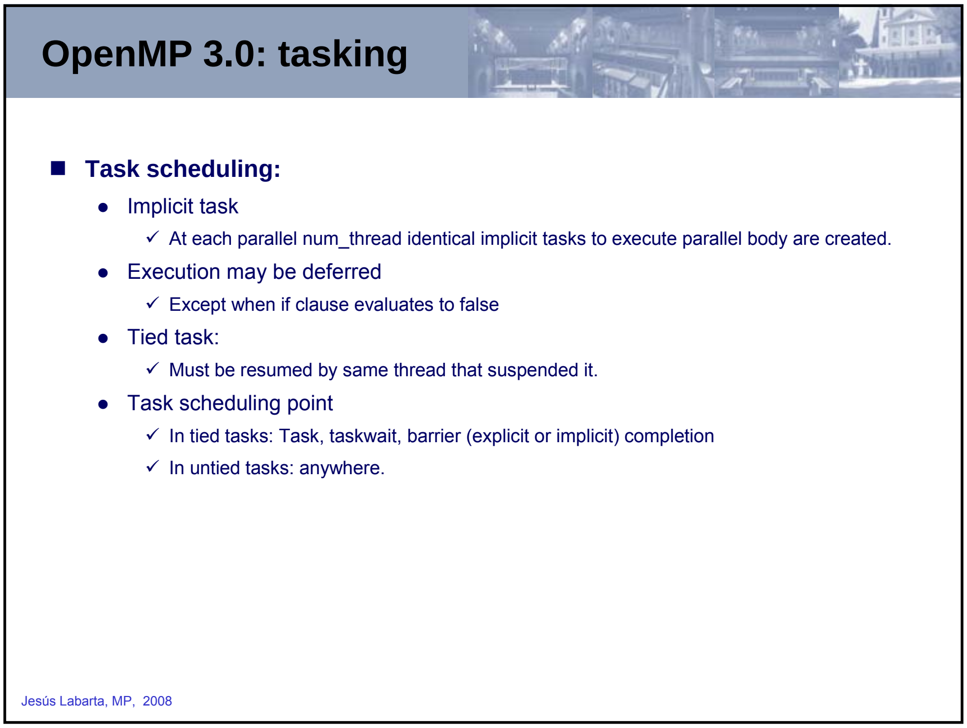
- **Implicit tasks:**
  - one per thread in the team, tied

```
#pragma omp parallel
```

- **Wait on the completion of child tasks generated**

```
#pragma omp taskwait
```

- **Nesting**



OpenMP 3.0: tasking

- **Task scheduling:**
  - Implicit task
    - ✓ At each parallel num_thread identical implicit tasks to execute parallel body are created.
  - Execution may be deferred
    - ✓ Except when if clause evaluates to false
  - Tied task:
    - ✓ Must be resumed by same thread that suspended it.
  - Task scheduling point
    - ✓ In tied tasks: Task, taskwait, barrier (explicit or implicit) completion
    - ✓ In untied tasks: anywhere.

OpenMP 3.0: tasking

- **Synchronization:**
  - Barrier
    - ✓ All tasks generates within parallel must have completed.
  - Taskwait
    - ✓ Suspend current task until completion of all child tasks it has generated.
  - Lock ownership
    - ✓ by tasks, not threads

OpenMP 3.0

- **Loop schedules**
  - Static, dynamic, guided, auto, runtime
  - collapse
- **Environment variables**
  - OMP_DYNAMIC: true/false
    - ✓ Number of threads
  - OMP_NESTED

  - OMP_WAIT_POLICY: active/passive
  - OMP_THREAD_LIMIT
  - OMP_STACKSIZE

OpenMP 3.0: tasking examples

- **Recursivity**

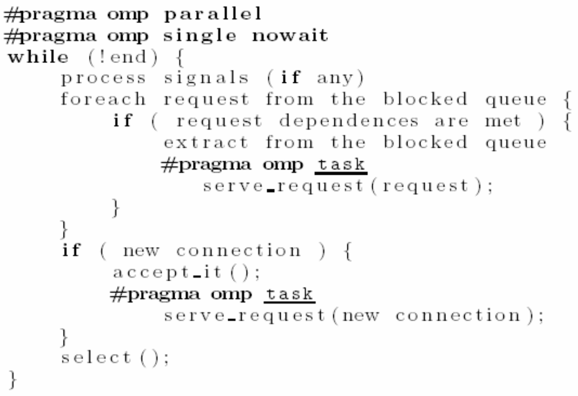
```
void traverse(binarytree *p, bool postorder) {
  #pragma omp task
    if (p->left) traverse(p->left, postorder);
  #pragma omp task
    if (p->right) traverse(p->right, postorder);
  if (postorder) {
    #pragma omp taskwait
  }
  process(p);
}
```

- **Pointer chasing**

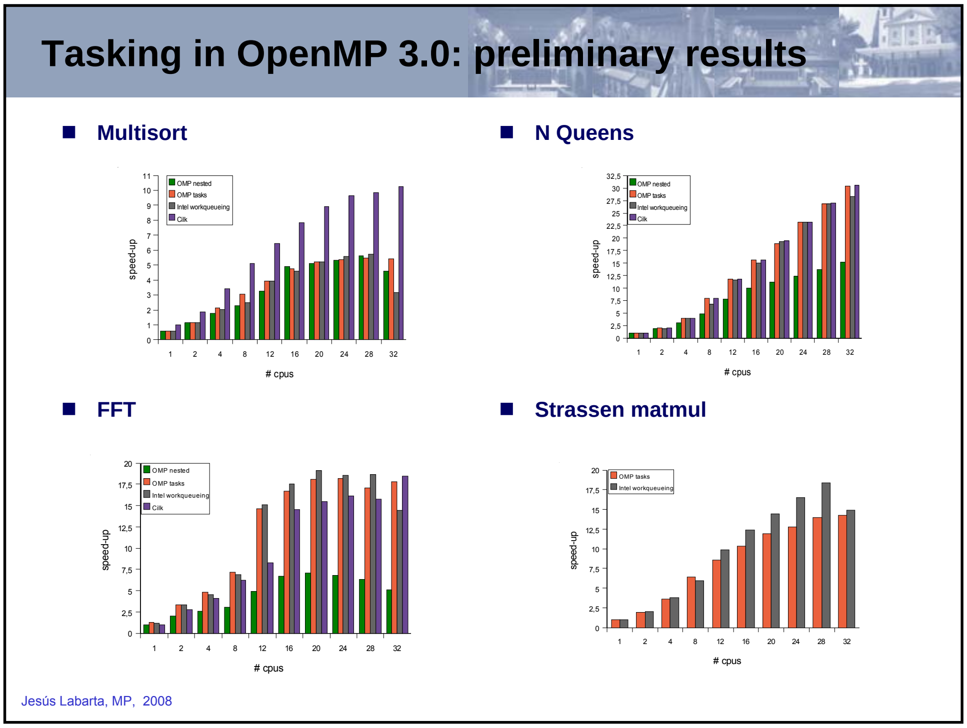
```
 #pragma omp parallel
  {
    #pragma omp single
    { p = listhead;
      while(p) {
        #pragma omp task
           process(p)
        p=next(p);
      }
    }
  }
```

Tasking in OpenMP 3.0: more examples

- **Asynchronous tasks (e.g web server)**



```
#pragma omp parallel
#pragma omp single nowait
while (!end) {
    process signals (if any)
    foreach request from the blocked queue {
        if ( request dependences are met ) {
            extract from the blocked queue
            #pragma omp task
               serve_request(request);
        }
    }
    if ( new connection ) {
        accept_it();
        #pragma omp task
            serve_request(new connection);
    }
    select();
}
```



Tasking in OpenMP 3.0: preliminary results

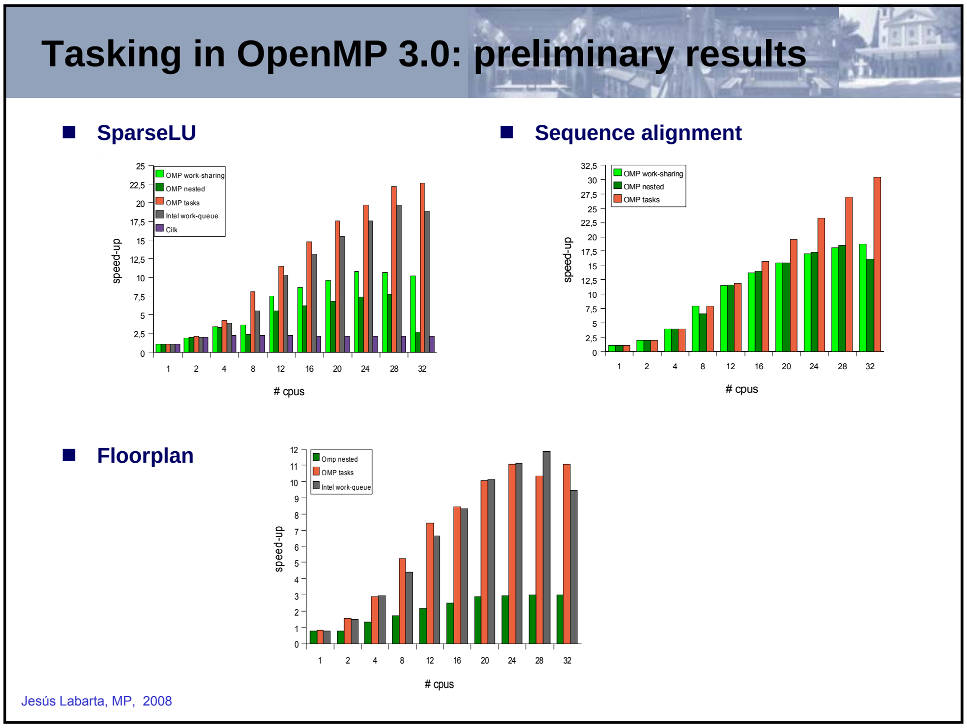
- **Multisort**

- **N Queens**

- **FFT**

- **Strassen matmul**

Jesús Labarta, MP, 2008



Tasking in OpenMP 3.0: preliminary results

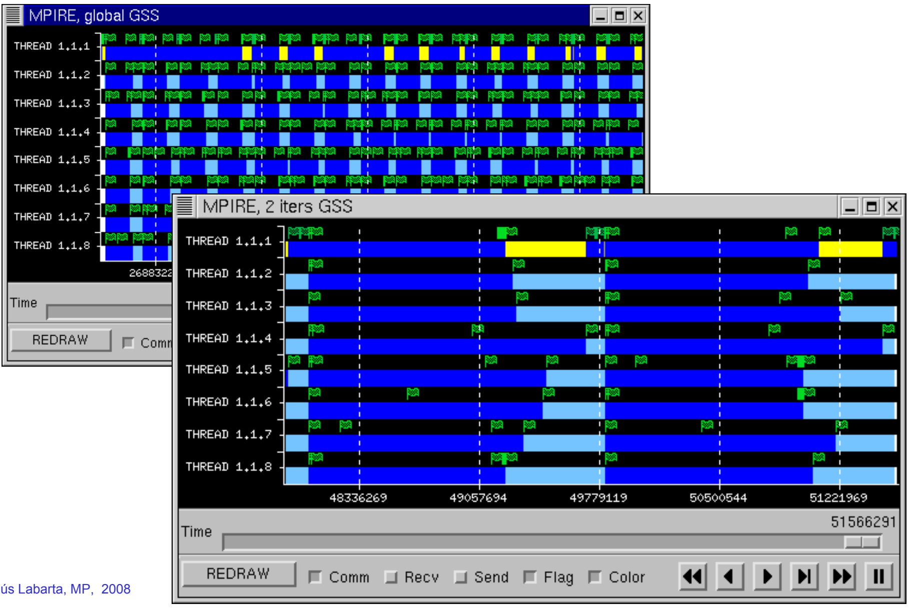
- **SparseLU**

- **Sequence alignment**

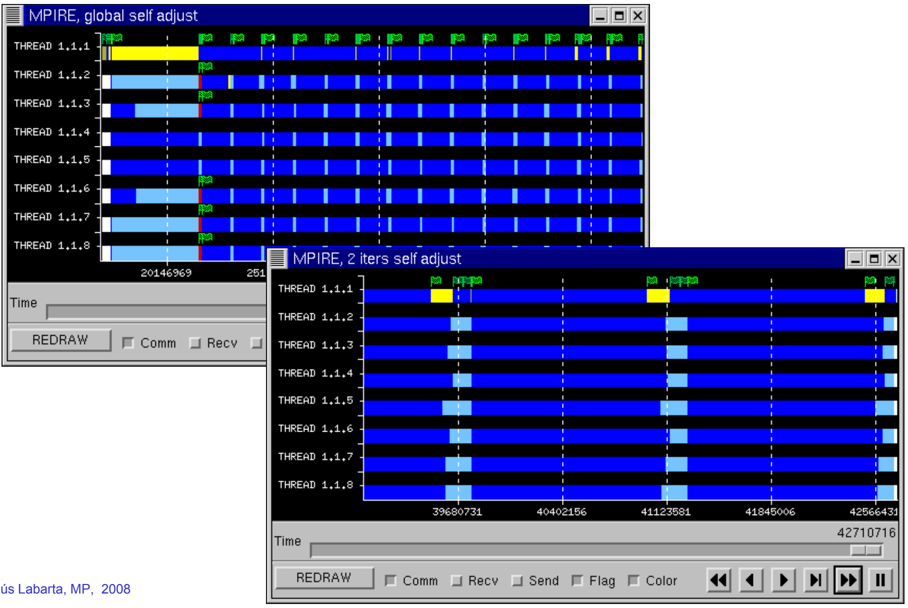
- **Floorplan**

Jesús Labarta, MP, 2008

Loop scheduling: MPIRE



Loop scheduling: MPIRE



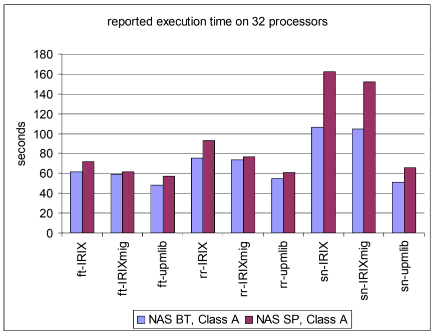
Platform related issues

- **Architectural**
  - False sharing
  - Heterogeneity → load imbalance
- **OS**
  - Page and process placement considerations
  - Malleability
- **In general**
  - Variability, Dynamicity
  - In workload/application characteristics and resource availability

---

NUMA considerations

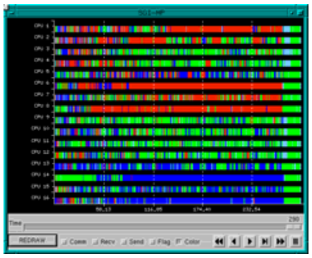
- **Page placement**
  - Non interleaved
  - Round robin
  - First touch
- **Page migration**
  - Counters
  - Policies
  - Correlation to process placement
    - ✓ Pinning
    - ✓ migration



Dimitris et all. “Is data distrubution needed in OpenMP?”

CPU scheduling

- **3 instances of an application at the same time**
  - Asking for 16 processors each
  - In a 16 way SMP
- **Process management impact**
  - Resource utilization
  - Overhead
  - Locality
- **Desirable**
  - Possibility to change
  - minimize changes



Malleability

- **Flexible (dynamic) parallelization structure of an application**
  - Allows responsiveness to dynamic characteristics of a computation and resource availability
- **Malleability requires**
  - Separation between
    - ✓ Algorithm: Problem logic. Programmer responsibility
    - ✓ Scheduling: → efficiency. System responsibility (hints may help)
  - Frequent control points
- **Issue of:**
  - Programming model support
  - Programming practices

Malleability

Scheduling decisions: Once for all

Explicit code only related to parallelism

```
C$OMP PARALLEL
WhoAmI=RunTimeCall()
myBlock=f(WhoAmI)
…
Call Compute1(myBlock)
…
DO iters=1, #iters
   Call Compute2(myBlock)
END DO
…
C$OMP END PARALLEL
```

```
C$OMP PARALLEL DO
DO Block=1, #blocks
 Call Compute1(Block)
ENDO
C$OMP END PARALLEL
…
DO iter=1,#iters
   C$OMP PARALLEL DO
   DO Block=1, #blocks
      Call Compute1(Block)
   ENDO
   C$OMP END PARALLEL
END DO
…
C$OMP END PARALLEL
```

WhoamI
#threads in the code
☹

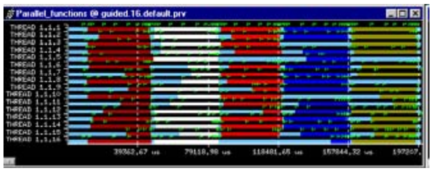
myBlock ∈ [1,#processors] ← f(resources)
Block ∈ [1,#blocks] ← f(algorithm)

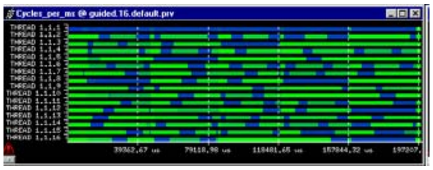
Run time library

- Compute schedules
- Wait modes
  - Overhead
  - Behavior under overcommitted multiprogrammed workload
  - Busy wait, yield, block
    - ✓ When to change mode?
- Locks
- Synchronizations

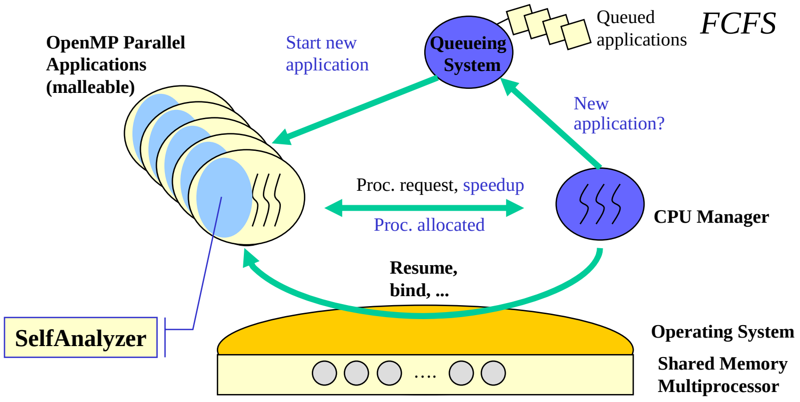
Run time library

- **Multiprogramming effects**
  - OMP_NUM_THREADS=16
  - Guided
  - Machine
    - ✓ 16 way SMP
    - ✓ load ≅ 9


Parallel_functions @ guided.16.default.prv


Cycles_per_ms @ guided.16.default.prv

NANOS OS scheduling environment


OpenMP Parallel Applications (malleable)

Start new application

Queueing System

Queued applications

FCFS

New application?

Proc. request, speedup

Proc. allocated

CPU Manager

Resume, bind, ...

SelfAnalyzer

Operating System

Shared Memory Multiprocessor

….