Geilo Winter School 2008

Programming Multicore Processors

Session 2

Henrik Löf, Sverker Holmgren, Jarmo Rantakokko

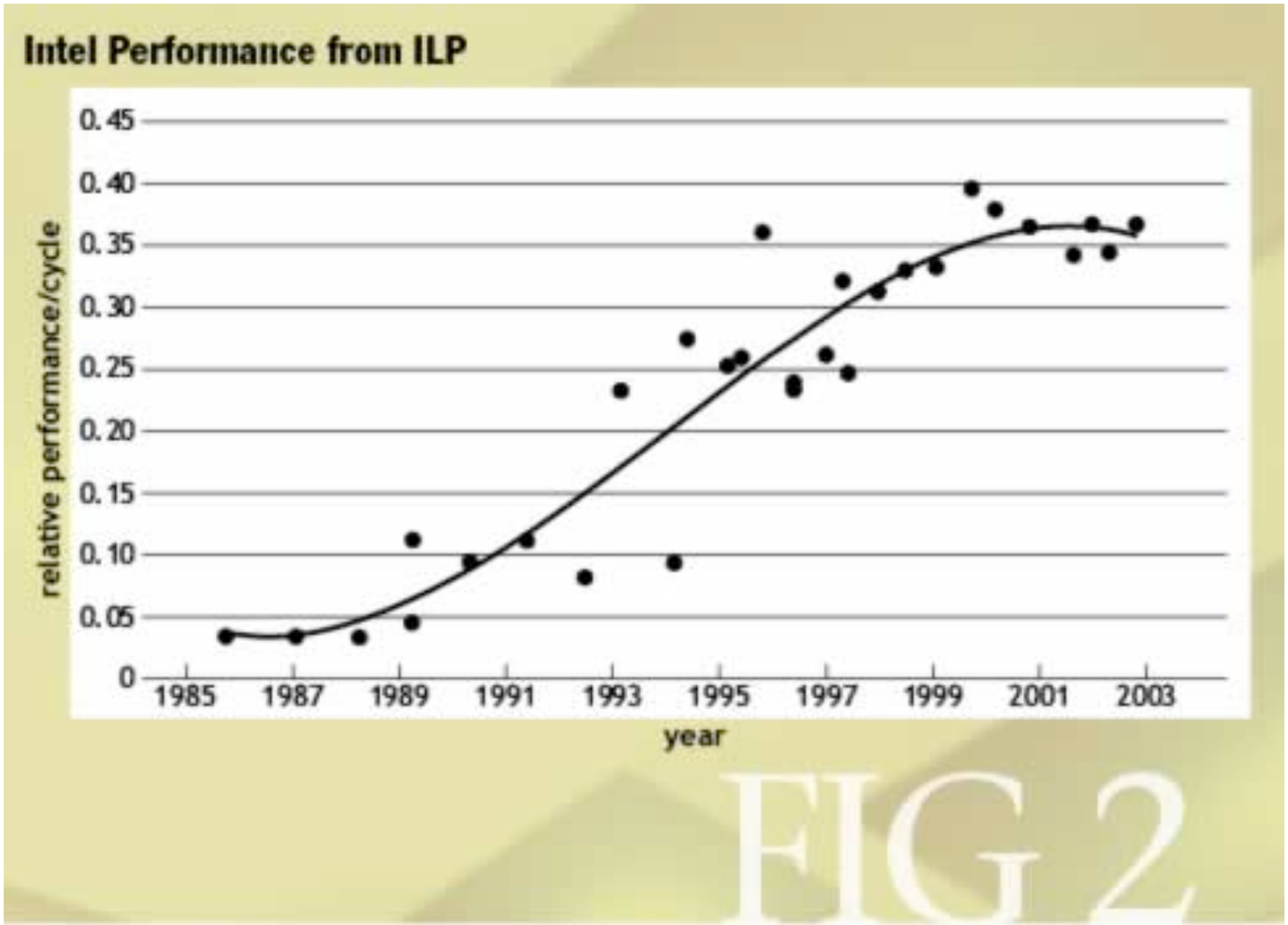
UPPSALA UNIVERSITET

Classic ways to high performance

- Locality
  - Exploiting cache memories
- Instruction Level Parallelism (ILP)
  - Extracted "automatically" by the compiler
  - Unrolling
  - Supported by microprocessor inventions
    - Multiple pipelines, superscalar execution
    - Branch prediction
    - Out-of-order execution

UPPSALA
UNIVERSITET

Limit on ILP



UPPSALA UNIVERSITET

Free lunch is over

- ILP is dead
  - Compilers are stuck
  - Single thread performance will not follow Moore´s law
- Parallelism is king
  - Number of cores and hardware threads will increase
  - From now on, every performance critical component must be parallelized
  - What to do with existing code?
  - Which model/language shall I use for a rewrite?

UPPSALA UNIVERSITET

Ways to Parallelism

- Task Parallelism
- Auto-parallelizing compilers
  - Recompile your code
- Parallelized Libraries
  - Recompile your code
- Data parallelism
  - Rewrite and recompile
- Classic parallel programming
  - First get a PhD
  - Then redesign, rewrite, recompile, re-everything

UPPSALA UNIVERSITET

Task parallelism (GRID)

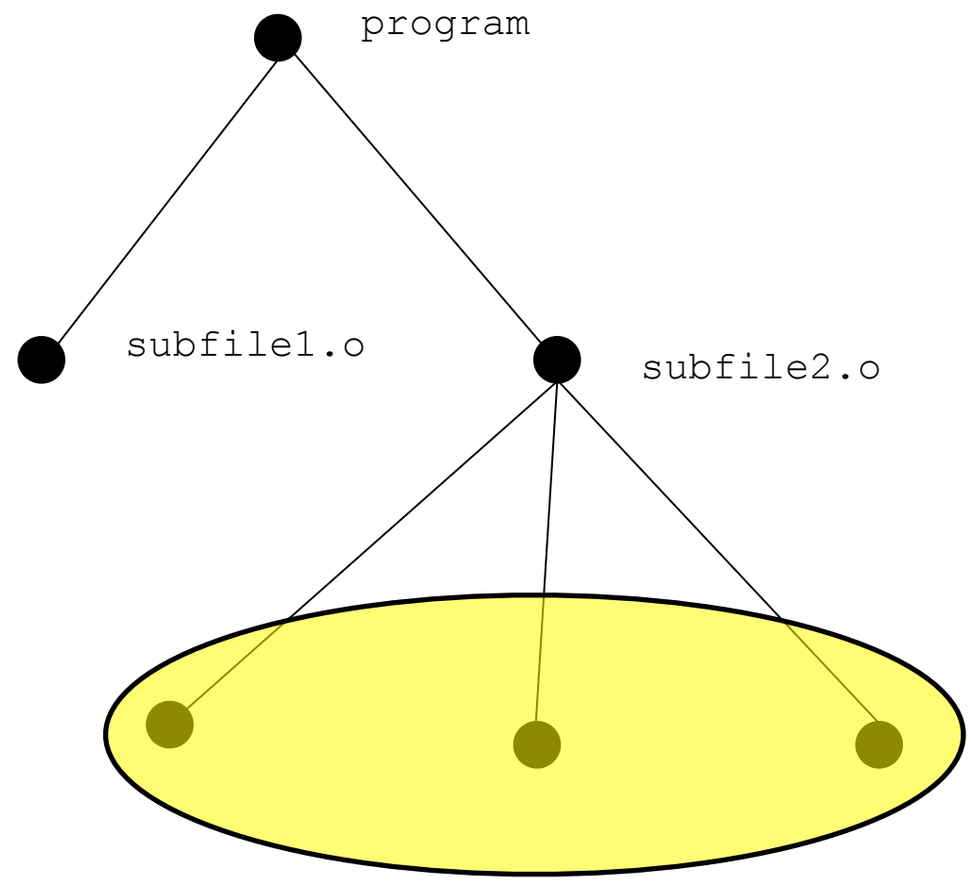
- Coarse grained
  - No fine-grained communication
- Reducing OS time-sharing effects
  - Schedule processes of different progams to multiple program counters
  - Run “Web browser”, “Email client”, and the OS in parallel
- Clusters
  - Schedule multiple serial jobs (matlab runs, or data base requests) to nodes of a cluster
  - Parameter studies
  - Monte-Carlo simulations

UPPSALA UNIVERSITET

Example

- Your makefiles define dependencies between software components
  - Think of it as a graph or tree
- Leaves of the tree can be compiled in parallel
  - The subtasks are synchronized at the parent
- GNU Make: "make -j *n*"

```
program: subfile1.o subfile2.o
          gcc -o program …

subfile1.o: subfile1.c
          gcc -c ……
subfile2.o: subfile2.c
          gcc -c ….
```



UPPSALA
UNIVERSITET

Parallelizing Compilers

- Been around for 30 years
  - Very limited applicability
- Compilers are conservative
  - Will not produce code that fails even if it only happens every full moon on a tuesday
- Compilers have limited "vision"
  - Hard and expensive to do whole program analysis
  - Most interesting things are unknown at compile time
- Typical blockers
  - Data dependencies, pointer aliasing, function calls

UPPSALA UNIVERSITET

Libraries and Components

- Replace all your external library calls with parallelized variants
  - BLAS, LAPACK, MKL, ACML, NAG, ESSL, ..
- Extend your code using predefined skeletons and libraries
  - Intel Thread Building Blocks (TBB), STAPL (STL)
  - PETSc, HYPRE, Trilinos
- Most of this stuff is for MPI
- Perfect fit for some applications
- Remember Amdahl's law
  - Limited speedup if 10% of your code is serial

UPPSALA UNIVERSITET

Data parallelism

- Let multiple processors chew on your data in parallel
  - Fine-grained or coarse-grained
- If you use many processors and your tasks are fine-grained you may be hit by Amdahl's law (again!)
  - Communication costs
  - Synchronization
- Need to work with the parallel overhead
  - Efficient algorithms
  - Careful implementations
  - Choosing the right model/tool

Parallel

UPPSALA UNIVERSITET

Exposing Parallelism

- Sequential machine
  - One program counter
  - ILP
- Parallel machines
  - Multi-processors: many program counters
  - Data parallelism: vector machines, SIMD, GPU (monster ILP)
- Programming models
  - How do we load the program counters?

UPPSALA UNIVERSITET

fork()/clone()

- Classic way to load two program counters
  - One process per program counter
- Creates a child process
  - Child process is identical copy (except PID etc)
- Copy-on-Write (COW)
  - Copy is privatized on the first write

```c
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>
int main()
{
    pid_t childpid;
    int retval, status;

    childpid = fork();

    if (childpid >= 0) /* fork succeeded */
    {
        if (childpid == 0) /* child process */
        {
            printf("CHILD: I am the child process!\n");
            printf("CHILD: Here's my PID: %d\n", getpid());
            sleep(1);
            exit(0);
        }
        else /* parent process */
        {
            printf("PARENT: I am the parent process!\n");
            printf("PARENT: Here's my PID: %d\n", getpid());
            wait(&status); /* wait for child to exit */
            exit(0);  /* parent exits */
        }
    }
    else /* fork returns -1 on failure */
    {
        perror("fork"); /* display error message */
        exit(0);
    }
}
```

UPPSALA UNIVERSITET

Inter-Process Communication (IPC)

- Because of COW processes communicate using OS services
  - Files
  - Shared Memory Segments
  - Sockets
  - CORBA
  - RPC
- Explicit message passing between two private address spaces
- Distributed Memory Programming
  - Can run on a single machine
  - Better name: "local name space" model
  - Examples: MPI, Erlang

UPPSALA UNIVERSITET

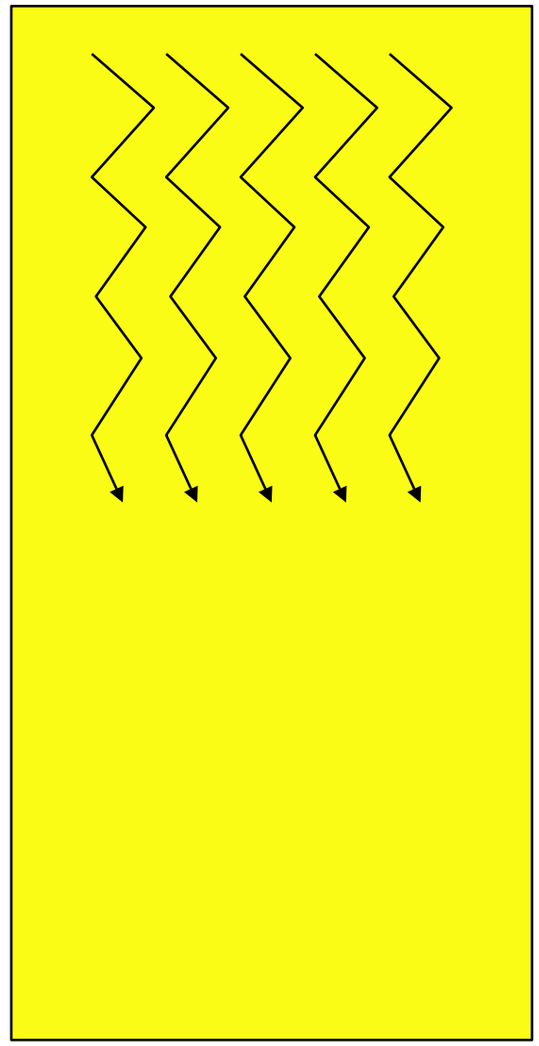
Message Passing Interface (MPI)

- Every cluster node runs a daemon process
- MPI_Create is a remote fork() of this daemon process (COW)
- MPI_Send is a wrapper to some IPC mechanism
- On clusters you have to use a network interface
- Low overhead interfaces can write directly to the memory of another node using RDMA (one-sided communication in MPI-2)
- On a shared memory machine you can use a shared memory IPC
  - Use “postboxes” to send messages

UPPSALA UNIVERSITET

Threads

- Run multiple PC:s inside a single process
  - "Threads" of computation
- Threads share the entire address space
  - No IPC between threads
  - Communication by loads and stores
  - Shared name space



UPPSALA UNIVERSITET

Creating Threads

- Supply a function pointer (POSIX, Windows)
- Create a new thread object and call the run() method (Java)
- Lightweight context switching
- Used primarily for concurrency not parallelism

```c
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void *thread_func( void *vptr_args );

int main( void ){
  int i, j;
  pthread_t thread;

  pthread_create( &thread, NULL, &thread_func, NULL );

  do_some_work();

  pthread_join( thread, NULL );

  exit( EXIT_SUCCESS );
}

void *thread_func( void *vptr_args ){
  int i, j;

    do_some_work();
  }
  return NULL;
}
```

UPPSALA UNIVERSITET

Local Name Space Programming

- Need to distribute data explicitly
  - May require a complete rewrite of your application
- Need to handle all communication explicitly
  - Opportunities for optimization
  - Lots of code
  - Source of errors (deadlocks etc.)
- Assembly language of parallel programming
- Runs on both distributed memory and shared memory architectures

UPPSALA UNIVERSITET

Shared Name Space Programming

- Decomposition is implicit
  - You can incrementally parallelize your application
  - "Use it only where it matters"
- Communication is implicit
  - Handled by cache coherency
  - Less coding
  - May trigger unnecessary communication
  - Spinlocks is tricky business
- Shared memory systems are hard to understand
  - But they appear to be simple

UPPSALA UNIVERSITET

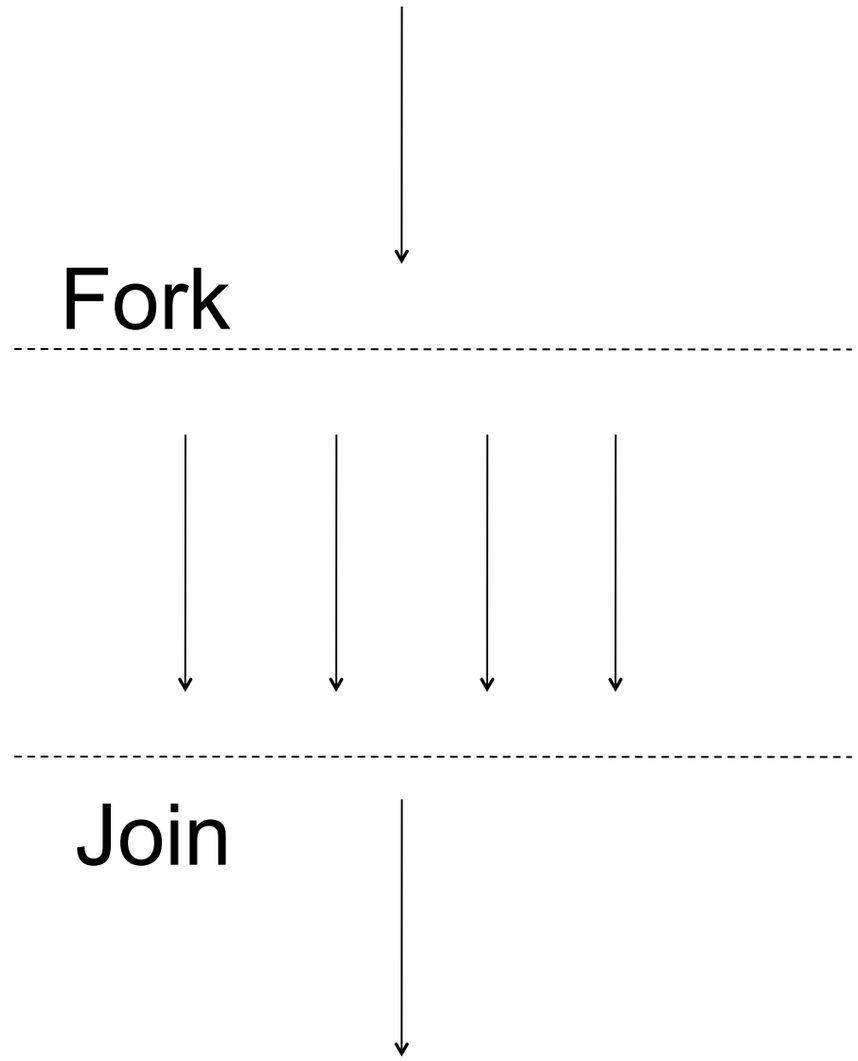
OpenMP

- OpenMP is a set of directives which transforms a serial code into a parallel one
  - Parallelization is triggered by the compiler
  - Source to source translation
- Calls a runtime library that uses POSIX threads
- If your compiler does not support OpenMP your code will be compiled into a serial program
- Supported by most compilers
  - Fortran, C and C++
  - Intel and Sun most influential in the OpenMP ARB
  - GCC 4.2

UPPSALA UNIVERSITET

OpenMP fork/join model

- A fundamental concept in OpenMP is the parallel region
- An OpenMP program starts executing using one master thread
- When it hits a parallel region directive, it spawns a team of slave threads which execute the code in parallel
  - Your program counters are executing code from the parallel region



UPPSALA
UNIVERSITET

Example

```
#include <omp.h>

  int main(void) {

    printf(“This is the master thread with ID %d\n, omp_get_thread_num());

    /* Define a parallel region */

 #pragma omp parallel
 {
     printf(“I am slave thread %d\n”,omp_get_thread_num());
 }

    printf(“This is the master thread again with ID %d\n, omp_get_thread_num());

 return 0;
}
```

UPPSALA UNIVERSITET

Compiling

```
$cc –xopenmp example.c
cc: Warning: Specify a supported level of optimization when using -
 xopenmp, -xopenmp will not set an optimization level in a future
 release. Optimization level changed to 3 to support –xopenmp
```

```
$./a.out
This is the master thread with ID 0
I am slave thread 0
This is the master thread again with ID 0
```

UPPSALA UNIVERSITET

Setting the number of threads

- By default, the team will consist only of the master thread when the parallel region is entered
- To add threads to the default team size use
  1. Environment variable OMP_NUM_THREADS
  2. Library call **omp_set_num_threads(n)**

UPPSALA UNIVERSITET

Setting the number of threads

```
$ export OMP_NUM_THREADS=4
$./a.out
This is the master thread with ID 0
I am slave thread 2
I am slave thread 1
I am slave thread 0
I am slave thread 3
This is the master thread again with ID 0
```

UPPSALA UNIVERSITET

OpenMP directives (spec 2.5):

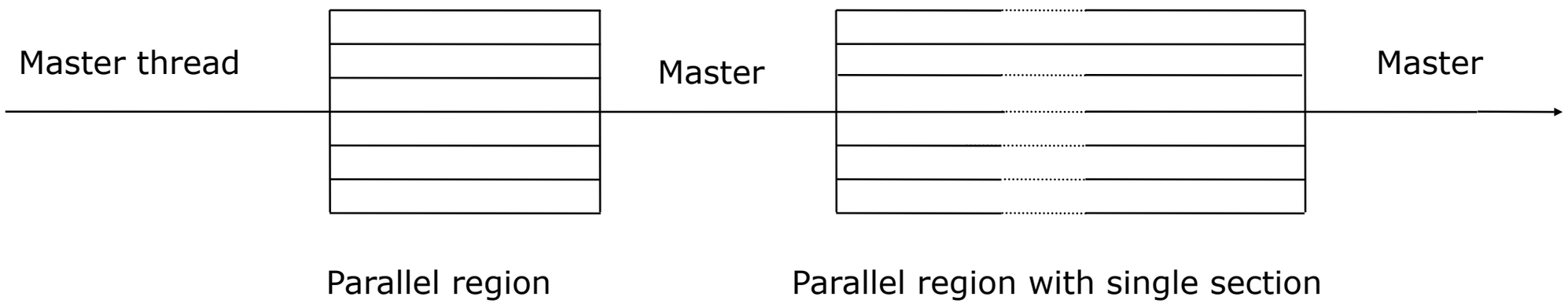
Parallel (main, fork threads)

Data sharing
- shared
- private
- firstprivate
- lastprivate
- threadprivate

Work sharing
- do/for
  - reduction
  - schedule
  - ordered
- sections

Serial sections
- single
- master
- critical
- atomic
- ordered

Synchronization
- barrier
- flush
- nowait
- spinlocks



UPPSALA UNIVERSITET

OpenMP library functions:

- omp_set_num_threads
- omp_get_num_threads
- omp_get_max_threads
- omp_get_thread_num
- omp_set_nested
- and more (e.g. locks)

Allows for more flexible and user controlled (e.g. load balancing) programming than with the standard directives.

Environment variables:

- OMP_NUM_THREADS
- OMP_SCHEDULE
- OMP_NESTED

UPPSALA UNIVERSITET

Directives: (Support only in Fortran/C/C++)

C/C++: #pragma omp *directive*
{ code block }

Fortran: !$omp *directive*
code block
!$omp end *directive*

Note: The directives are ignored by non-supporting compiler or if OpenMP-flag is turned off in compiling.
=> Portable code between single CPU, multi-core, and general parallel computers.

Also, possible to parallelize code incrementally
(start with heaviest routine and continue until sufficient parallelism and performance are achivied)

UPPSALA UNIVERSITET

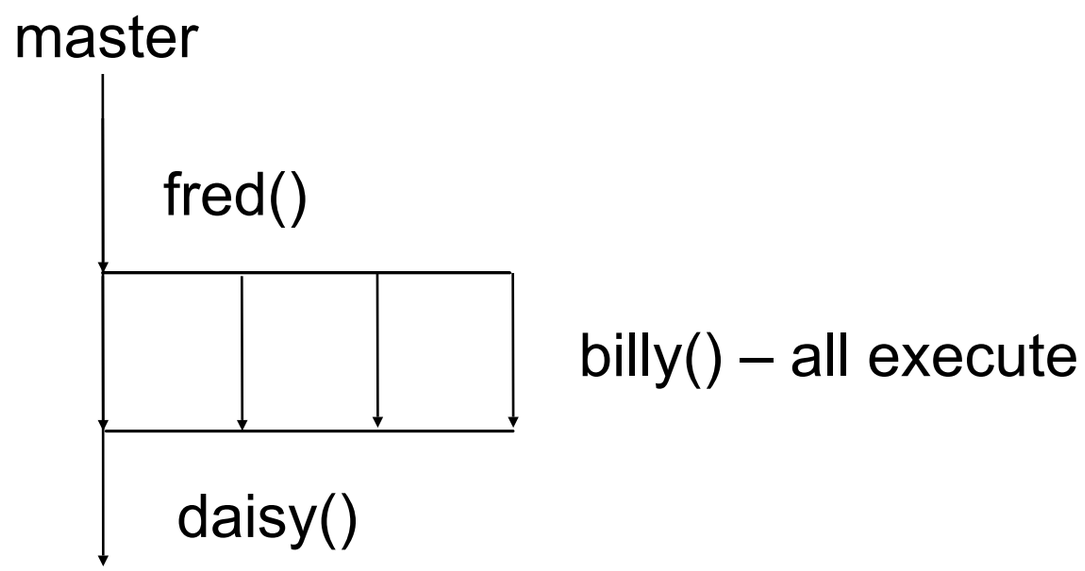
Parallel: (Fork-Join)

```
!$omp parallel [clauses]
   "parallel code"
!$omp end parallel
```

If no clauses, all data shared (global) and all code executed in parallel by all threads. At the end of parallel the threads are synchronized and joined.

Ex: program p1

```
...
call fred()
!$omp parallel
     call billy()
!$omp end parallel
call daisy()
```



UPPSALA UNIVERSITET

Serial sections

!$omp single [clauses]
The code-block within single is executed only by one thread, the others skip and wait at the end of block.
Clauses: - private
- firstprivate

!$omp master
The code-block is executed only by master thread, the other skip and continue (no barrier).

UPPSALA UNIVERSITET

Data sharing:

- **shared**([list of variables]) - default
- **private**([list of variables])

Ex: program p2

```
...
a=10; b=0;
!$omp parallel private(a)
      a=a+10
      b=b+a
!$omp end parallel
write(*,*) a,b
```

What is the result (assume 4 threads)?

UPPSALA
UNIVERSITET

Note: All private variables are allocated on the stack
=> uninitialized at entry and removed at exit,
original a not equal to *private a*!

Note2: Shared variables must be protected from
simultaneous writes by different threads!
(Use a critical section directive or spinlock.)

- **firstprivate**([list of variables])
  As private but the variables are initialized from the original variable (in master) before parallel.

- **lastprivate**([list of variables])
  At exit, the original variable gets the value from the thread executing the last iteration in a loop using the do-directive or the last section in the sections-directive.

UPPSALA UNIVERSITET

Data sharing:

A fixed program

```
Ex: program p2
    ...
    a=10; b=0;
    !$omp parallel firstprivate(a)

      a=a+10

    !$omp critical
          b=b+a
    !$omp end critical

     !$omp end parallel

     write(*,*) a,b
```

Add firstprivate

Add critical section. (more on this later)

UPPSALA UNIVERSITET

Barriers

```
#include <stdio.h>
#include <unistd.h>
#include <omp.h>

int main(void) {

#pragma omp parallel
  {

    sleep(omp_get_thread_num()+1);

    printf("I am slave thread %d\n",omp_get_thread_num());

#pragma omp barrier

    printf("I am still slave thread %d\n",omp_get_thread_num());

  }

  return 0;
```

Hack to order the threads

UPPSALA
UNIVERSITET

Barrier,example

With barrier

```
I am slave thread 0
I am slave thread 1
I am slave thread 2
I am slave thread 3
I am still slave thread 3
I am still slave thread 0
I am still slave thread 2
I am still slave thread 1
```

Without barrier

```
I am slave thread 0
I am still slave thread 0
I am slave thread 1
I am still slave thread 1
I am slave thread 2
I am still slave thread 2
I am slave thread 3
I am still slave thread 3
```

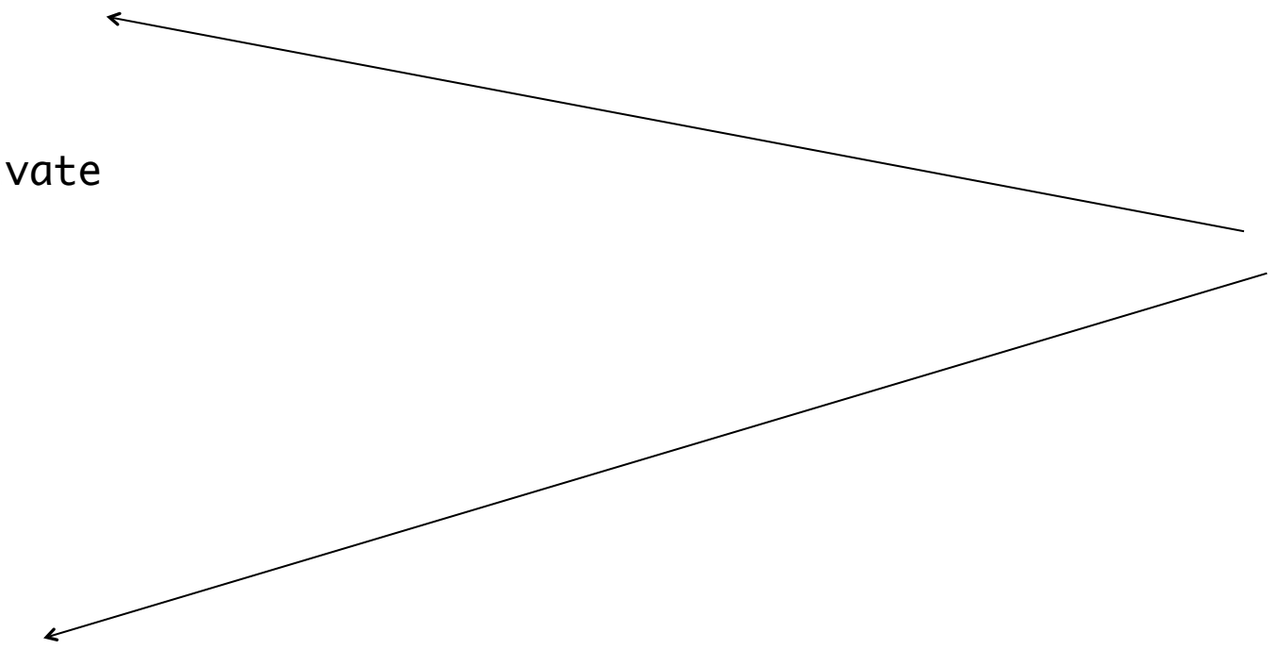
UPPSALA
UNIVERSITET

Private/Shared example

```
 int myid;

#pragma omp parallel shared(myid)
  {
    myid = omp_get_thread_num();
#pragma omp barrier
    printf("I am slave thread %d\n",myid);
  }

  printf("Making the variable private\n");

#pragma omp parallel private(myid)
  {
    myid = omp_get_thread_num();
#pragma omp barrier
    printf("I am slave thread %d\n",myid);
  }
```

Parallel execution

```
 bash-3.1$ ./a.out
I am slave thread 1
I am slave thread 1
I am slave thread 1
I am slave thread 1
Making the variable private
I am slave thread 1
I am slave thread 3
I am slave thread 0
I am slave thread 2
bash-3.1$ ./a.out
I am slave thread 0
I am slave thread 0
I am slave thread 0
I am slave thread 0
Making the variable private
I am slave thread 1
I am slave thread 2
I am slave thread 3
I am slave thread 0
```



Depends on the ordering of the threads!

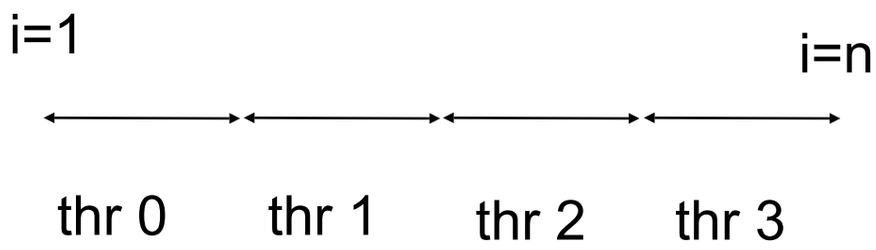
UPPSALA UNIVERSITET

Worksharing (within parallel)

- Loop level parallelism – do/for
- Task parallelism - sections

do/for-directive:

```
!$omp do [clauses]
do i=1,n
    loop-body
end do
[ !$omp end do ]
```



Without clauses, loop counter is private, loop space is divided statically into nthr equal pieces, and run in parallel (different iterations in different threads). Threads are synchronized at end of the for-directive.

Note: We must have a perfectly parallel loop!

UPPSALA UNIVERSITET

Worksharing Clauses:

- Shared
- Private
- Firstprivate
- Lastprivate
- Reduction
- Schedule
- Ordered

UPPSALA UNIVERSITET

Scalar product example

```
sum = 0.0;
for(i=0;i<N;i++)
  {
     sum += (a [i] + b[i] );
  }

printf("Scalar product is %f\n",sum);
```

UPPSALA UNIVERSITET

Scalar product example

```
sum = 0.0;
for(i=0;i<N;i++)
  {
     sum += (a [i] + b[i] );
  }

printf("Scalar product is %f\n",sum);
```

```
for(i=0;i<N;i++)
  {
     a[i]+=b[i];
  }
sum = 0.0;
for(i=0;i<N;i++)
     sum += a[i];

printf("Scalar product is %f\n",sum);
```

UPPSALA UNIVERSITET

Scalar product example

```
sum = 0.0;
for(i=0;i<N;i++)
  {
     sum += (a [i] + b[i] );
  }

printf("Scalar product is %f\n",sum);
```

Perfectly Parallel

```
for(i=0;i<N;i++)
  {
     a[i]+=b[i];
  }
sum = 0.0;
for(i=0;i<N;i++)
     sum += a[i];

printf("Scalar product is %f\n",sum);
```

UPPSALA UNIVERSITET

Worksharing example

```
printf("This is the master thread with ID %d\n",omp_get_thread_num());

#pragma omp parallel
  {
    printf("I am slave thread %d\n",omp_get_thread_num());
#pragma omp single
    {
      printf("Summing two vectors of size %d in parallel\n",N);
    }
#pragma omp for
    for(i=0;i<N;i++) {
      a[i]+=b[i];
    }
 } /* End of parallel region */

  sum = 0.0;
  for(i=0;i<N;i++)
    sum+=a[i];
  printf("This is the master thread again with ID %d\n",omp_get_thread_num());
  printf("Array sum is %f\n",sum);

  return 0;
}
```

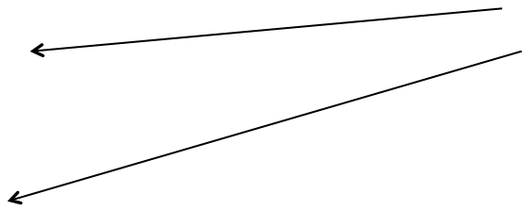
Automatic or implicit barrier

UPPSALA UNIVERSITET

Parallel execution

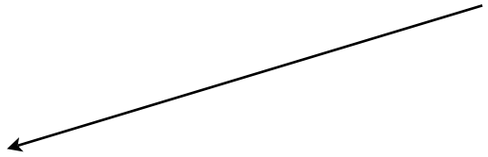
Init a,b to
a[:] = 0.0
b[:] = 1..1000

```
$./a.out
This is the master thread with ID 0
I am slave thread 2
Summing two vectors of size 1000 in parallel
I am slave thread 0
I am slave thread 3
I am slave thread 1
This is the master thread again with ID 0
Array sum is 5.005000e+05
$
```



Notice the ordering of the printouts

Worksharing example

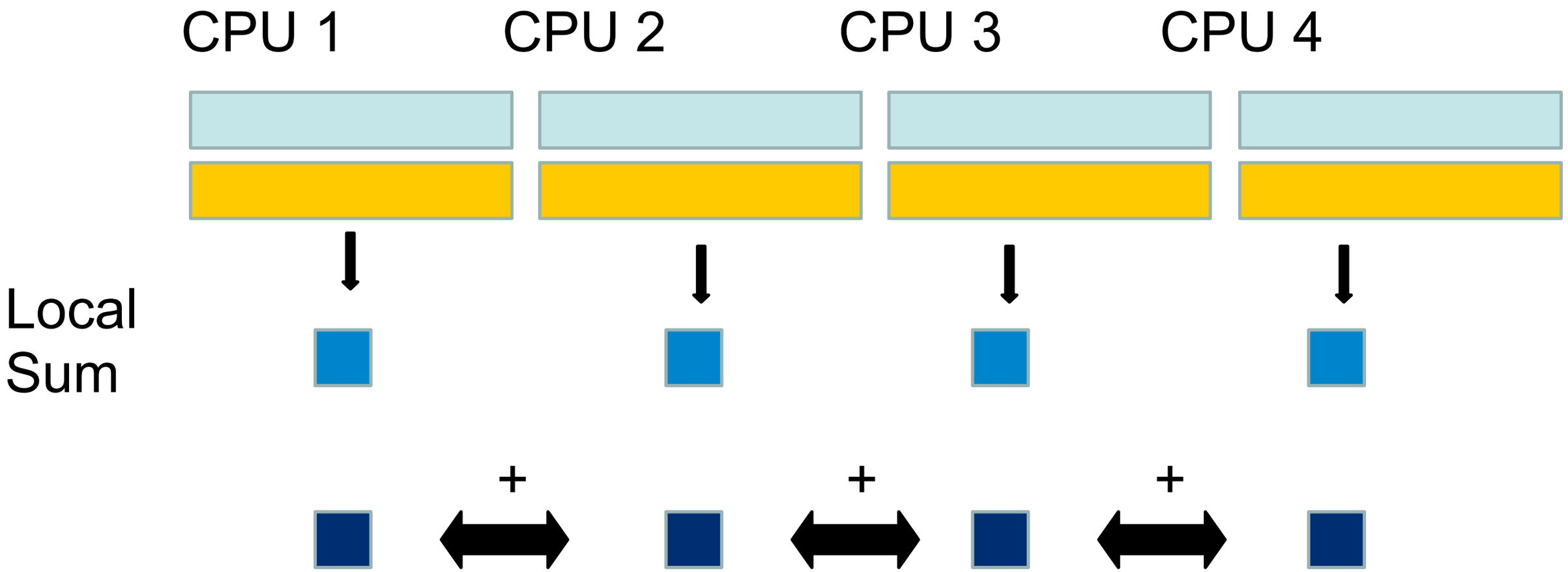


Parallel region and worksharing

```
#pragma omp parallel for shared(a,b) private(i)
    for(i=0;i<N;i++) {
      a[i]+=b[i];
    }
 } /* End of parallel region */

  sum = 0.0;
  for(i=0;i<N;i++)
    sum+=a[i];

  printf("Array sum is %f\n",sum);

  return 0;
}
```

UPPSALA UNIVERSITET

Example: parallel scalar product



Communicate and sum to form p copies of scalar product

UPPSALA UNIVERSITET

Parallel scalar product

- Create a local sum per thread and the sum them together
- How do we create a local variable?
- If its local how can we sum them together. The other threads does know about it?
- Answer: use a **reduction clause**

UPPSALA UNIVERSITET

Adding a reduction clause

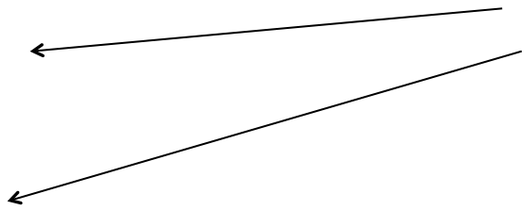
```
sum = 0.0;
#pragma omp parallel shared(sum)
  {
    printf("I am slave thread %d\n",omp_get_thread_num());

#pragma omp single
    {
      printf("Summing two vectors of size %N in parallel\n",N);
    }

#pragma omp for reduction(+:sum)
    for(i=0;i<N;i++) {
      sum += a[i] * b[i];
    }

  } /* end of parallel region */
```

Creates a local copy of sum, and combines the local copies using the “+” operator ar the end of the loop

UPPSALA UNIVERSITET

Parallel execution

```
$./a.out
This is the master thread with ID 0
I am slave thread 2
I am slave thread 1
I am slave thread 0
I am slave thread 3
Summing two vectors of size 1000 in parallel
This is the master thread again with ID 0
Array sum is 5.005000e+05
$
```



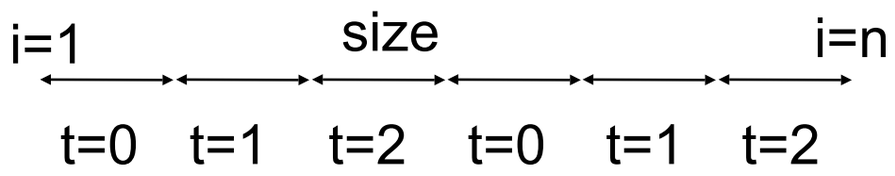
Notice the ordering of the printouts

UPPSALA UNIVERSITET

Schedule(type, [size])

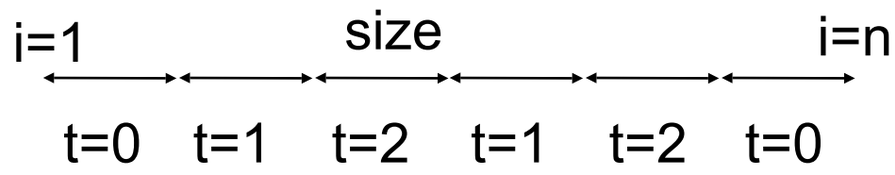
Divides the iteration space into chunks=size and schedules the chunks to threads according to type. (size=n/nthr by default)

type=static:



Assign the chunks cyclicly to threads

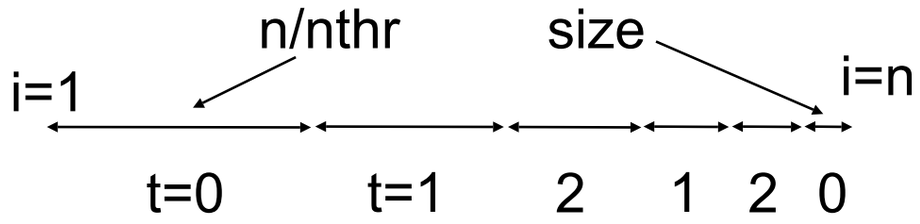
type=dynamic:



Dynamic scheduling, as soon as a thread is ready it gets a new chunk

UPPSALA UNIVERSITET

type=guided:



As dynamic but the chunk size is decreasing exponentially. Minimizes synchronization time.

type=runtime:

Decide at runtime using the environment variable export *schedule=type* (where *type* is some above) or let the compiler decide (don't set *schedule*).

Note: Static scheduling is good for data locality (cache) while dynamic/guided good for load balance.

UPPSALA UNIVERSITET

Explicit user supplied load balancing:

Ex:

```
!$OMP PARALLEL DO SCHEDULE
DO J=1,N
      A(J)=WORK(A(J))
END DO
```

Assume irregular work load, using

static => load imbalance

dynamic,1 => cache misses

Explicit load balancing:

```
CALL COMPUTE(LB,UB,NUM_THREADS)
!$OMP PARALLEL PRIVATE(J,ID)
      ID = OMP_GET_THREAD_NUM()
      DO J=LB(ID),UB(ID)
            A(J)=WORK(A(J))
      END DO
!$OMP END PARALLEL
```

UPPSALA UNIVERSITET

Sections

```
!$omp sections [subdirectives]

      !$omp section
            task 1
      !$omp end section

      !$omp section
            task 2
      !$omp end section

      etc.
!$omp end sections
```

Subdirectives:

- Private
- Firstprivate
- Lastprivate
- Reduction

The sections/tasks are scheduled (statically) to the threads and run in parallel. At end of sections the threads are synchronized. (No load balancing).

UPPSALA UNIVERSITET

Nested parallelism (load balancing of sections)

```
CALL OMP_SET_NESTED(.TRUE.)
!$OMP PARALLEL SECTIONS OMP_NUM_THREADS(2)

      !$OMP SECTION
      !$OMP PARALLEL DO OMP_NUM_THREADS(P1)
      DO K=1,N
            call WORK1(A(K))
      END DO
      !$OMP END SECTION

      !$OMP SECTION
      !$OMP PARALLEL DO OMP_NUM_THREADS(P2)
      DO K=1,N
            call WORK2(A(K))
      END DO
      !$OMP END SECTION

!$OMP END PARALLEL SECTIONS
```

Assign appropriate number of threads to each section.

Synchronization

Done implicitly at end of:

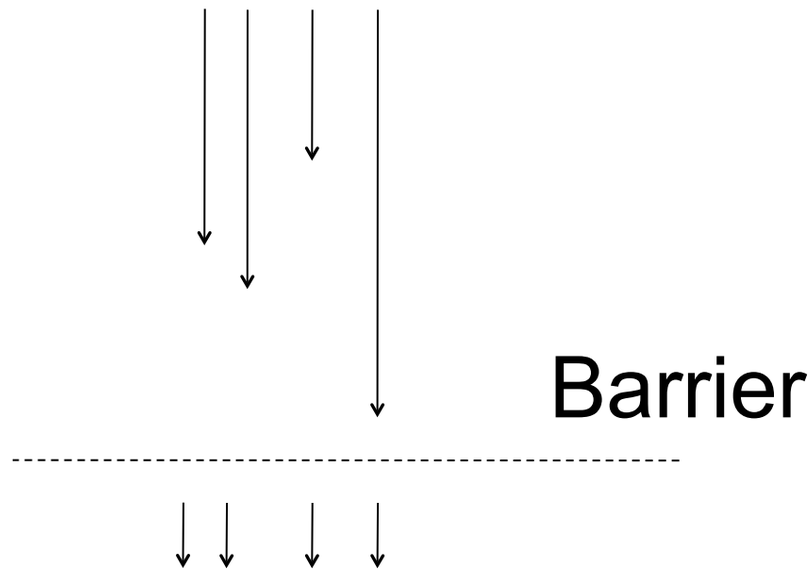
- parallel
- do/for
- sections
- single

Can override with *nowait:*

```
!$omp do
do i=1,n
      code
end do
!$omp end do nowait
```

Explicit barrier:

```
!$omp barrier
```



UPPSALA UNIVERSITET

!$omp critical [name]

The code-block is executed by one thread at a time.
As ordered but no predefined order.
If no name all critical sections have the same name.
Only one critical section with the same name can be executed by one thread at a time.

!$omp atomic
Atomic update by one thread at a time. As critical but applies only for a one line expression.

UPPSALA UNIVERSITET

Matrix-Multiply example

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define NRA 8 /* number of rows in matrix A */
#define NCA 552 /* number of columns in matrix A */
#define NCB 23 /* number of columns in matrix B */
int main (int argc, char *argv[]) {
  int tid, nthreads, i, j, k, chunk;
  double a[NRA][NCA], /* matrix A to be multiplied */
         b[NCA][NCB], /* matrix B to be multiplied */
         c[NRA][NCB]; /* result matrix C */

/*** Spawn a parallel region explicitly scoping all variables ***/
#pragma omp parallel shared(a,b,c,nthreads) private(tid,i,j,k)
 {
    tid = omp_get_thread_num();
    if (tid == 0) {
       nthreads = omp_get_num_threads();
       printf("Starting matrix multiple example with %d threads\n",nthreads);
       printf("Initializing matrices...\n");
    }
```

UPPSALA UNIVERSITET

Matrix-Multiply example

```
/*** Initialize matrices ***/

#pragma omp for
  for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
      a[i][j]= i+j;

#pragma omp for
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
      b[i][j]= i*j;

#pragma omp for
  for (i=0; i<NRA; i++)
    for (j=0; j<NCB; j++)
      c[i][j]= 0;
```

UPPSALA UNIVERSITET

Matrix-Multiply example

```
/*** Do matrix multiply sharing iterations on outer loop ***/
/*** Display who does which iterations for demonstration purposes ***/

printf("Thread %d starting matrix multiply...\n",tid);
#pragma omp for
  for (i=0; i<NRA; i++) {
     printf("Thread=%d did row=%d\n",tid,i);
     for(j=0; j<NCB; j++)
        for (k=0; k<NCA; k++)
           c[i][j] += a[i][k] * b[k][j];
  }
} /*** End of parallel region ***/
/*** Print results ***/
printf("Result Matrix:\n");
for (i=0; i<NRA; i++) {
   for (j=0; j<NCB; j++)
     printf("%6.2f ", c[i][j]);
     printf("\n");
}
printf ("Done.\n");
```

Variables j and k must be private

UPPSALA UNIVERSITET

Matrix-Multiply execution

```
Starting matrix multiple example with 4 threads
Initializing matrices...
Thread 0 starting matrix multiply...
Thread=0 did row=0
Thread 1 starting matrix multiply...
Thread=1 did row=2
Thread 2 starting matrix multiply...
Thread=2 did row=4
Thread 3 starting matrix multiply...
Thread=3 did row=6
Thread=0 did row=1
Thread=1 did row=3
Thread=2 did row=5
Thread=3 did row=7
```

Outer loop is 8 elements. Two columns per thread.

UPPSALA UNIVERSITET

Jacobi example

```
do while (k <= maxit .and. error > tol)
   error = 0.0
!$omp parallel
!$omp workshare
  uold(i,j) = u(i,j) ! Copy new solution into old

! Compute stencil, residual, & update
!$omp do private(i,j,residual) reduction(+:error)
   do j = 2,m-1
     do i = 2,n-1
       residual = (ax*(uold(i-1,j) + uold(i+1,j)) + ay*(uold(i,j-1) + uold(i,j+1)) &
            + b * uold(i,j) - f(i,j))/b
! Update solution
       u(i,j) = uold(i,j) - residual
       error = error + residual*residual
     end do
  end do
!$omp enddo nowait
!$omp end parallel
  k = k + 1
  error = sqrt(error)/dble(n*m)
enddo ! End iteration loop
```

Automatically creates loop and parallelizes it

UPPSALA UNIVERSITET

OpenMP 3.0

- New standard
  - Implementations are probably a year or so away
- Biggest revision
  - Parallel tasks
  - Break away from loop-centered parallelisation
- Lots of attention from the gaming industry
  - Microsoft Visual Studio
  - Xbox
  - PC games

UPPSALA UNIVERSITET

Cluster of CMP nodes

- MPI only
  - Use network between the cluster nodes
  - Use shared memory between the processes within the node
  - Processes within the node are wasting cycles waiting for intra-node communication to complete
  - Processes are heavyweight, can consume system resources
- OpenMP only
  - You can create a shared memory architecture in software
  - Called Software Distributed Shared Memory Systems
  - Intel sells a Cluster-OpenMP

UPPSALA UNIVERSITET

Hybrid Parallelization

- Decompose your problem in coarse-grained pieces
- Map these to nodes using MPI
- Parallelize the operations within each process using OpenMP
  - Or calling a parallelized library
- Issues
  - Thread-safe MPI implementations
  - Load Balancing (setting the number of threads)
  - Algorithmic issues (mapping your algorithm to your system)

UPPSALA UNIVERSITET

Hybrid Programming Models

- One-sided communication in MPI-2
  - Read and write directly to the memory of another node
- Unified Parallel C (UPC)
  - Divide the addess space into a shared and private part
  - Partitioned Global Address Space (PGAS)
  - Annotate your code to tell what is shared and what is private
  - Other variants: Co-Array Fortran, Titanium (Java)
- Still a research topic
  - Used at the american labs (Livermore, Sandia..)
  - Needs special hardware support

UPPSALA
UNIVERSITET

New Languages

- Fortress (Sun)
  - Designed to be parallel (hidden from the programmer)
  - Designed to support mathematical notation
  - Partially available today
- X10 (IBM)
  - Java-like PGAS
  - Uses a virtual machine
- Chapel (Cray)
  - Descendant of High Performance Fortan (HPF)
- Who knows?

Programming Multicores

- Given existing code
  - Use parallelized libraries and components
  - Add a little OpenMP
  - Limited to a single CMP
- Starting a new project
  - Use libraries and components (may not match)
  - Use a hybrid model and algorithms taking the architecture of the system into account
  - MPI is never wrong but maybe not the most efficient

UPPSALA UNIVERSITET

Performance obstacles in OpenMP:

- Non-parallelized regions, serial sections
  Amdahl's law, Speedup < 1/s
- Synchronization
  Explicit/implicit barriers
- Load imbalance
  Trivial or naïve load balancing with OpenMP directives
- Cache misses => “communication”
  True/false sharing
- Non-optimal data placement on NUMA
  Costly remote memory accesses