Programming Multi-Core Systems with OpenMP

Clemens Grelck

University of Amsterdam

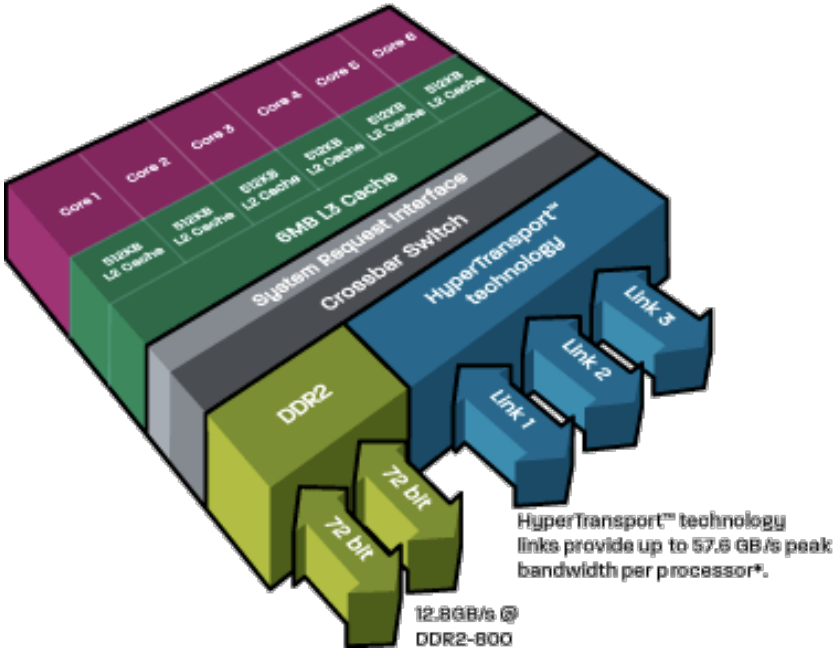
UvA / SURFsara
High Performance Computing and Big Data

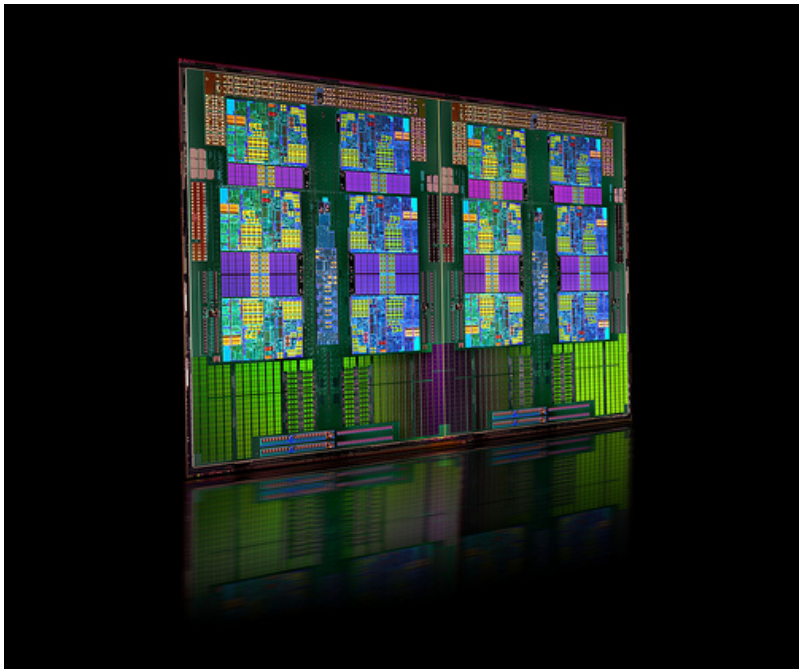
Programming Multi-Core Systems with OpenMP

Target Multi-core Systems

Small-scale general-purpose (x86) multicore processors:

- Intel / AMD commodity processors with 2, 4, 6 or 8 cores
- potentially hyperthreaded

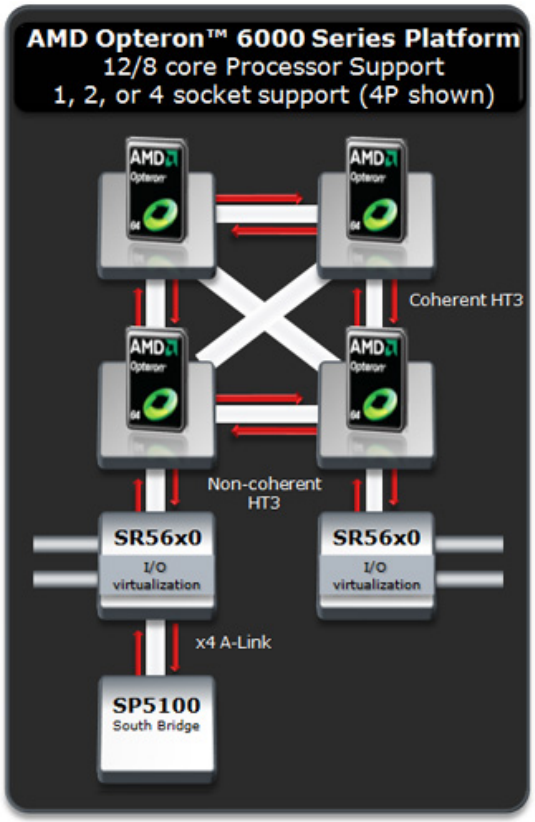




Target Multi-core Systems

Medium-scale server systems:

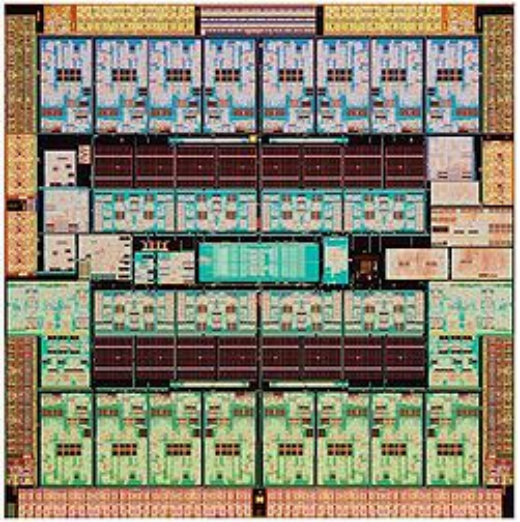
- multiple (2 or 4 in practice) identical processors
- each processor with several cores
- high bandwidth data path between processors



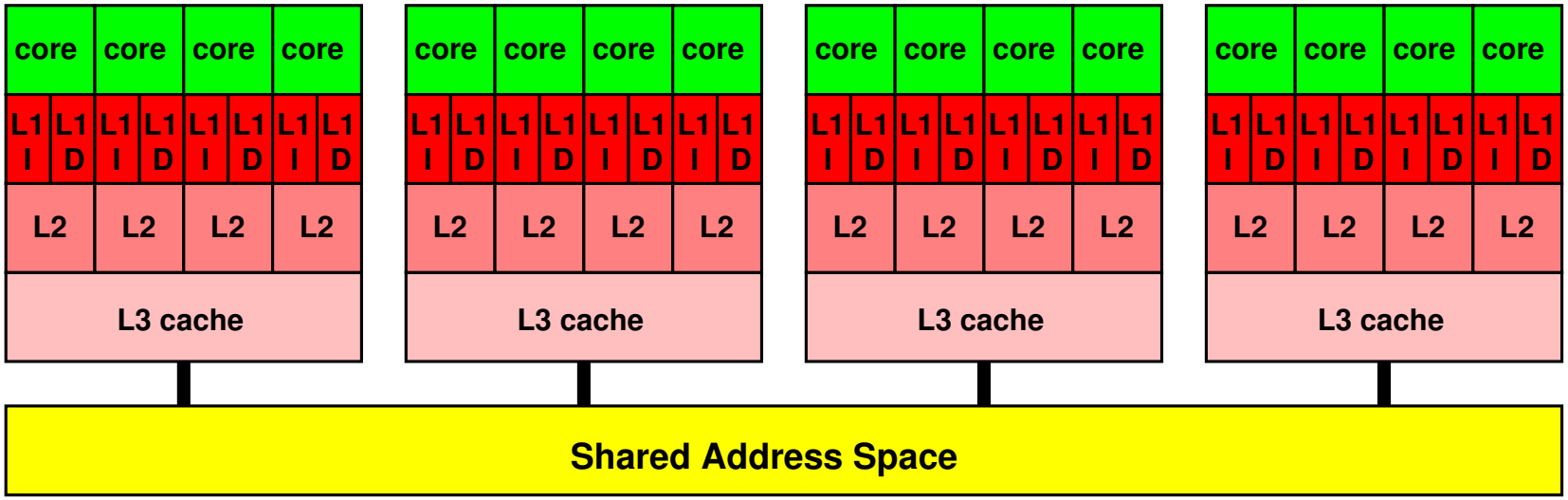
Target Multi-core Systems

Large-scale shared address space compute systems:

- large number of slightly simpler cores
- SUN MicroSystems / Oracle Niagara / UltraSparc T series
- up to 512 hardware threads (T3-4 server)



Symmetric Multiprocessor Architecture Model



Characteristics:

- **Shared address space notion of shared memory**
- Multiple levels of hardware-coherent caches
- Multiple processors
- Each processor has multiple cores
- Each core has multiple hardware threads

Programming Multi-Core Systems with OpenMP

Design Rationale of OpenMP

Ideal:

- Automatic parallelisation of sequential code.
- No additional parallelisation effort for development, debugging, maintenance, etc.

Design Rationale of OpenMP

Ideal:

- Automatic parallelisation of sequential code.
- No additional parallelisation effort for development, debugging, maintenance, etc.

Problem:

- Data dependences are difficult to assess.
- Compilers must be conservative in their assumptions.

Design Rationale of OpenMP

Ideal:

- Automatic parallelisation of sequential code.
- No additional parallelisation effort for development, debugging, maintenance, etc.

Problem:

- Data dependences are difficult to assess.
- Compilers must be conservative in their assumptions.

Way out:

- Take or write ordinary sequential program.
- Add annotations/pragmas/compiler directives that guide parallelisation.
- Let the compiler generate the corresponding code.

OpenMP at a Glance

OpenMP as a programming interface:

- Compiler directives
- Library functions
- Environment variables

OpenMP at a Glance

OpenMP as a programming interface:

- Compiler directives
- Library functions
- Environment variables

C/C++ version:

```
#pragma omp name [clause]*
structured block
```

Fortran version:

```
!$ OMP name [ clause [, clause]*]
code block
!$ OMP END name
```

Hello World with OpenMP

Example program:

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

int main()
{
  printf( "Starting execution with %d threads.\n",
          omp_get_num_threads());

  printf( "Hello world says thread %d of %d.\n",
          omp_get_thread_num(),
          omp_get_num_threads());

  printf( "Execution of %d threads terminated.\n",
          omp_get_num_threads());

  return 0;
}
```

Hello World with OpenMP

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

int main()
{
  printf( "Starting execution with %d threads.\n",
          omp_get_num_threads());

  printf( "Hello world says thread %d of %d.\n",
          omp_get_thread_num(),
          omp_get_num_threads());

  printf( "Execution of %d threads terminated.\n",
          omp_get_num_threads());

  return 0;
}
```

Compiling the code:

```
gcc -fopenmp hello_world.c
```

Hello World with OpenMP

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

int main()
{
  printf( "Starting execution with %d threads.\n",
          omp_get_num_threads());

  printf( "Hello world says thread %d of %d.\n",
          omp_get_thread_num(),
          omp_get_num_threads());

  printf( "Execution of %d threads terminated.\n",
          omp_get_num_threads());

  return 0;
}
```

Running the compiled code:

```
Starting execution with 1 threads.
Hello world says thread 0 of 1.
Execution of 1 threads terminated.
```

Hello World with OpenMP — now in parallel

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

int main()
{
  printf( "Starting execution with %d threads.\n",
          omp_get_num_threads());

  #pragma omp parallel
  {
    printf( "Hello world says thread %d of %d.\n",
            omp_get_thread_num(),
            omp_get_num_threads());
  }

  printf( "Execution of %d threads terminated.\n",
          omp_get_num_threads());

  return 0;
}
```

Hello World with OpenMP — now in parallel

Running the code with 4 threads:

```
Starting execution with 1 threads.
Hello world says thread 2 of 4.
Hello world says thread 3 of 4.
Hello world says thread 1 of 4.
Hello world says thread 0 of 4.
Execution of 1 threads terminated.
```

Hello World with OpenMP — now in parallel

Running the code with 4 threads:

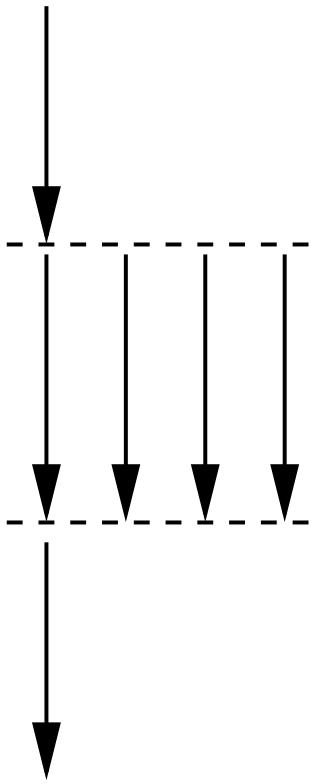
```
Starting execution with 1 threads.
Hello world says thread 2 of 4.
Hello world says thread 3 of 4.
Hello world says thread 1 of 4.
Hello world says thread 0 of 4.
Execution of 1 threads terminated.
```

Who determines number of threads ?

- Environment variable: `export OMP_NUM_THREADS=4`
- Library function: `void omp_set_num_threads( int)`

OpenMP Execution Model

Classical Fork/Join:



Master thread executes serial code.

Master thread encounters parallel directive.

Master and slave threads concurrently execute parallel block.

Implicit barrier, wait for all threads to finish.

Master thread resumes serial execution.

Programming Multi-Core Systems with OpenMP

Simple Loop Parallelisation

Example: element-wise vector product:

```
void elem_prod( double *c, double *a, double *b, int len)
{
  int i;

  for (i=0; i<len; i++)
  {
    c[i] = a[i] * b[i];
  }
}
```

Simple Loop Parallelisation

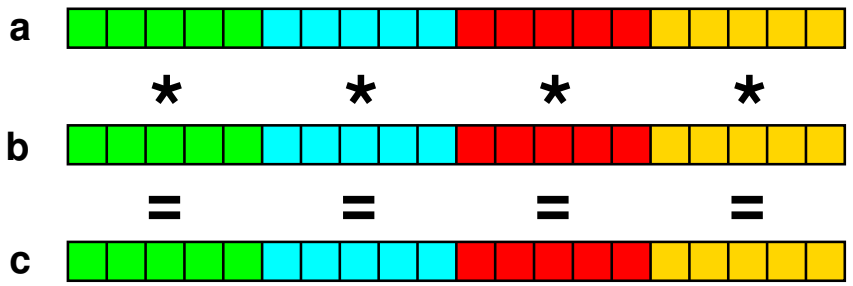
Example: element-wise vector product:

```
void elem_prod( double *c, double *a, double *b, int len)
{
  int i;

  for (i=0; i<len; i++)
  {
    c[i] = a[i] * b[i];
  }
}
```

Idea of parallelisation:

- Have each thread compute some disjoint part of the vectors:



Simple Loop Parallelisation

Example: parallelised element-wise vector product:

```
void elem_prod( double *c, double *a, double *b, int len)
{
  int i;

  #pragma omp parallel for
  for (i=0; i<len; i++)
  {
    c[i] = a[i] * b[i];
  }
}
```

Simple Loop Parallelisation

Example: parallelised element-wise vector product:

```
void elem_prod( double *c, double *a, double *b, int len)
{
  int i;

  #pragma omp parallel for
  for (i=0; i<len; i++)
  {
    c[i] = a[i] * b[i];
  }
}
```

Prerequisite:

- No data dependence between any two iterations.

Simple Loop Parallelisation

Example: parallelised element-wise vector product:

```
void elem_prod( double *c, double *a, double *b, int len)
{
  int i;

  #pragma omp parallel for
  for (i=0; i<len; i++)
  {
    c[i] = a[i] * b[i];
  }
}
```

Prerequisite:

- No data dependence between any two iterations.
- **Caution: YOU claim this property !!**

Directive #pragma omp parallel for

What the compiler directive does for you:

- It starts additional worker threads depending on `OMP_NUM_THREADS` or `omp_set_num_threads()`.
- It divides the iteration space among all threads.
- It lets all threads execute loop restricted to their mutually disjoint subsets.
- It synchronizes all threads at an implicit barrier.
- It terminates the worker threads.

Directive #pragma omp parallel for

What the compiler directive does for you:

- It starts additional worker threads depending on `OMP_NUM_THREADS` or `omp_set_num_threads()`.
- It divides the iteration space among all threads.
- It lets all threads execute loop restricted to their mutually disjoint subsets.
- It synchronizes all threads at an implicit barrier.
- It terminates the worker threads.

Restrictions:

- The directive must directly precede for-loop.
- The for-loop must match a constrained pattern.
- The trip-count of the for-loop must be known in advance.

Shared and Private Variables

Example:

```
#pragma omp parallel for
for (i=0; i<len; i++)
{
  c[i] = a[i] * b[i];
}
```

Private variables:

- One *private* instance for each thread
- No communication between threads within parallel section
- No communication between parallel and sequential sections

Shared variables:

- One *shared* instance for all threads
- Allows communication between threads within parallel section
- Allows communication between parallel and sequential sections

Shared and Private Variables

Example:

```
#pragma omp parallel for
for (i=0; i<len; i++)
{
  c[i] = a[i] * b[i];
}
```

Who decides that a, b, c and len are shared variables, whereas i is private ??

Shared and Private Variables

Example:

```
#pragma omp parallel for
for (i=0; i<len; i++)
{
  c[i] = a[i] * b[i];
}
```

Who decides that a, b, c and len are shared variables, whereas i is private ??

Default rules:

- All variables are **shared**.
- Only loop variables of parallel loops are **private**.

Shared and Private Variables

The default rule is not what you want ?

- The shared clause determines shared variables
- The private clause determines private variables

Example with explicit clauses:

```
#pragma omp parallel for private( i) shared( c, a, b, len)
for (i=0; i<len; i++)
{
  c[i] = a[i] * b[i];
}
```

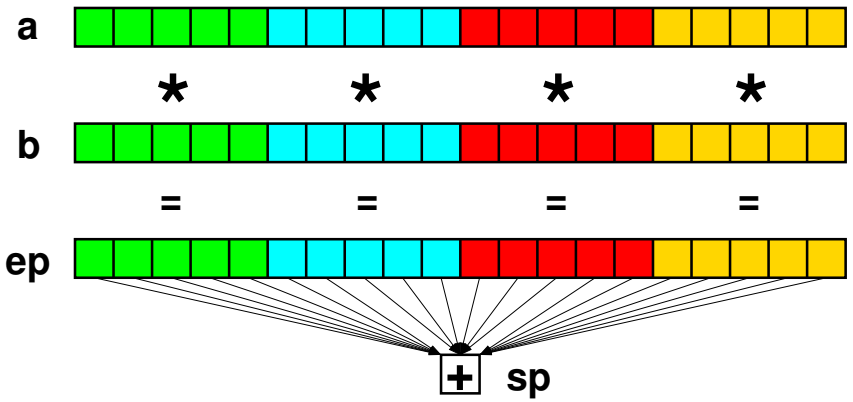
From Vector Product to Scalar Product

Scalar product:

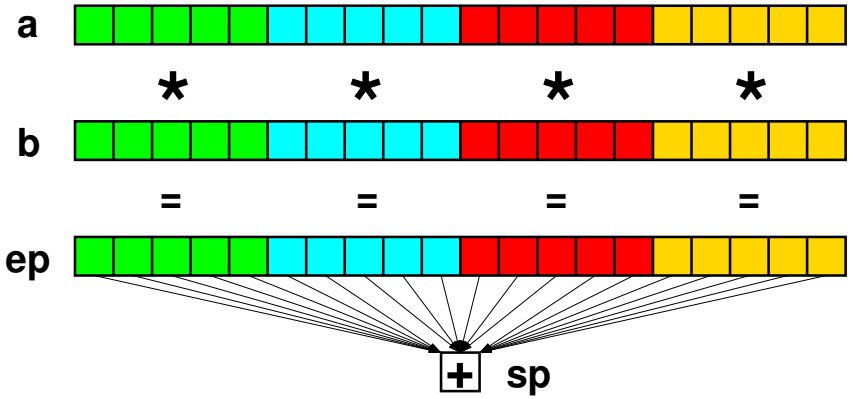
```
double scalar_prod( double *a, double *b, int len)
{
  int i; double ep, sp=0.0;

  for (i=0; i<len; i++)
  {
    ep = a[i] * b[i];
    sp = sp + ep;
  }

  return sp;
}
```



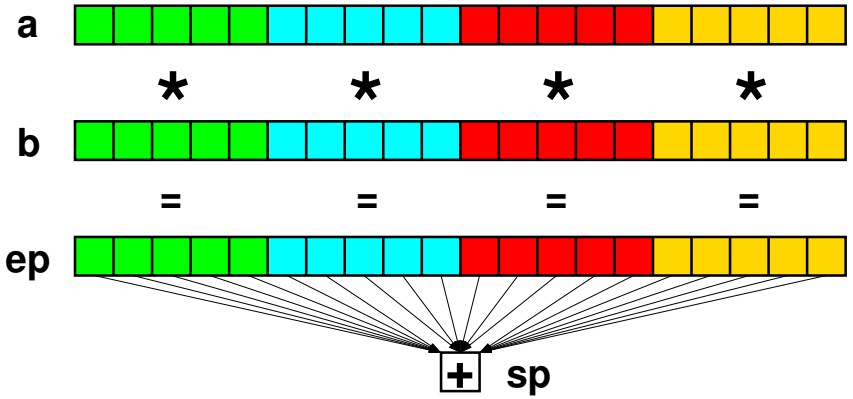
From Vector Product to Scalar Product



Reduction operations:

- Reductions *reduce* a set of elements to one
- Using some function
- Examples: sum or product of a set of numbers

From Vector Product to Scalar Product



Reduction operations:

- Reductions *reduce* a set of elements to one
- Using some function
- Examples: sum or product of a set of numbers

Can we parallelise reductions ?

Parallel Reduction

Scalar product:

```
double scalar_prod( double *a, double *b, int len)
{
  int i; double ep, sp=0.0;

  for (i=0; i<len; i++)
  {
    ep = a[i] * b[i];
    sp = sp + ep;
  }

  return sp;
}
```

Properties:

- Reduction introduces a *loop-carried dependence*.

Parallel Reduction

Scalar product:

```
double scalar_prod( double *a, double *b, int len)
{
  int i; double ep, sp=0.0;

  for (i=0; i<len; i++)
  {
    ep = a[i] * b[i];
    sp = sp + ep;
  }

  return sp;
}
```

Properties:

- Reduction introduces a *loop-carried dependence*.
- Good news: operation is associative

Parallel Reduction

Scalar product:

```
double scalar_prod( double *a, double *b, int len)
{
  int i; double ep, sp=0.0;

  for (i=0; i<len; i++)
  {
    ep = a[i] * b[i];
    sp = sp + ep;
  }

  return sp;
}
```

Properties:

- Reduction introduces a *loop-carried dependence*.
- Good news: operation is associative
- Order of reductions irrelevant for final result

Parallel Reduction

Scalar product:

```
double scalar_prod( double *a, double *b, int len)
{
  int i; double ep, sp=0.0;

  for (i=0; i<len; i++)
  {
    ep = a[i] * b[i];
    sp = sp + ep;
  }

  return sp;
}
```

Properties:

- Reduction introduces a *loop-carried dependence*.
- Good news: operation is associative
- Order of reductions irrelevant for final result
- (If we ignore deficiencies of computer arithmetic...)

Parallel Reduction

Scalar product:

```
double scalar_prod( double *a, double *b, int len)
{
  int i; double ep, sp=0.0;

  for (i=0; i<len; i++)
  {
    ep = a[i] * b[i];
    sp = sp + ep;
  }

  return sp;
}
```

Private variables ?

Shared variables ?

Properties:

- Reduction introduces a *loop-carried dependence*.
- Good news: operation is associative
- Order of reductions irrelevant for final result
- (If we ignore deficiencies of computer arithmetic...)

Parallel Reduction

Parallel scalar product:

```
double scalar_prod( double *a, double *b, int len)
{
  int i; double ep, sp=0.0;

  #pragma omp parallel for shared( a, b, len, sp) \
                           private( i, ep)
  for (i=0; i<len; i++)
  {
    ep = a[i] * b[i];
    sp = sp + ep;
  }

  return sp;
}
```

Parallel Reduction

Parallel scalar product:

```
double scalar_prod( double *a, double *b, int len)
{
  int i; double ep, sp=0.0;

  #pragma omp parallel for shared( a, b, len, sp) \
                           private( i, ep)
  for (i=0; i<len; i++)
  {
    ep = a[i] * b[i];
    sp = sp + ep;
  }

  return sp;
}
```

Is this really correct ?

A Look into Assembly

The troublemaker in C:

```
sp = sp + ep;
```

The troublemaker in (pseudo) assembly:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

A Look into Assembly

The troublemaker in C:

```
sp = sp + ep;
```

The troublemaker in (pseudo) assembly:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Problem:

- Same code executed by multiple threads simultaneously
- Any interleaving of assembly instructions possible

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	29	0	29	5	0	0

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	29	0	29	5	0	0
8	29	8	29	5	0	0

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	29	0	29	5	0	0
8	29	8	29	5	0	0
8	37	8	29	5	0	0

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	29	0	29	5	0	0
8	29	8	29	5	0	0
8	37	8	29	5	0	0
8	37	8	37	5	0	0

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	29	0	29	5	0	0
8	29	8	29	5	0	0
8	37	8	29	5	0	0
8	37	8	37	5	0	0
8	37	8	37	5	37	0

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	29	0	29	5	0	0
8	29	8	29	5	0	0
8	37	8	29	5	0	0
8	37	8	37	5	0	0
8	37	8	37	5	37	0
8	37	8	37	5	37	5

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	29	0	29	5	0	0
8	29	8	29	5	0	0
8	37	8	29	5	0	0
8	37	8	37	5	0	0
8	37	8	37	5	37	0
8	37	8	37	5	37	5
8	37	8	37	5	42	5

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	29	0	29	5	0	0
8	29	8	29	5	0	0
8	37	8	29	5	0	0
8	37	8	37	5	0	0
8	37	8	37	5	37	0
8	37	8	37	5	37	5
8	37	8	37	5	42	5
8	37	8	**42**	5	42	5

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	0	0	29	5	29	0

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	0	0	29	5	29	0
8	0	0	29	5	29	5

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	0	0	29	5	29	0
8	0	0	29	5	29	5
8	0	0	29	5	34	5

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	0	0	29	5	29	0
8	0	0	29	5	29	5
8	0	0	29	5	34	5
8	0	0	34	5	34	5

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	0	0	29	5	29	0
8	0	0	29	5	29	5
8	0	0	29	5	34	5
8	0	0	34	5	34	5
8	34	0	34	5	34	5

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	0	0	29	5	29	0
8	0	0	29	5	29	5
8	0	0	29	5	34	5
8	0	0	34	5	34	5
8	34	0	34	5	34	5
8	34	8	34	5	34	5

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	0	0	29	5	29	0
8	0	0	29	5	29	5
8	0	0	29	5	34	5
8	0	0	34	5	34	5
8	34	0	34	5	34	5
8	34	8	34	5	34	5
8	42	8	34	5	34	5

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	0	0	29	5	29	0
8	0	0	29	5	29	5
8	0	0	29	5	34	5
8	0	0	34	5	34	5
8	34	0	34	5	34	5
8	34	8	34	5	34	5
8	42	8	34	5	34	5
8	42	8	**42**	5	34	5

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	29	0	29	5	0	0

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	29	0	29	5	0	0
8	29	0	29	5	29	0

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	29	0	29	5	0	0
8	29	0	29	5	29	0
8	29	8	29	5	29	0

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	29	0	29	5	0	0
8	29	0	29	5	29	0
8	29	8	29	5	29	0
8	29	8	29	5	29	5

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	29	0	29	5	0	0
8	29	0	29	5	29	0
8	29	8	29	5	29	0
8	29	8	29	5	29	5
8	37	8	29	5	29	5

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	29	0	29	5	0	0
8	29	0	29	5	29	0
8	29	8	29	5	29	0
8	29	8	29	5	29	5
8	37	8	29	5	29	5
8	37	8	29	5	34	5

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	29	0	29	5	0	0
8	29	0	29	5	29	0
8	29	8	29	5	29	0
8	29	8	29	5	29	5
8	37	8	29	5	29	5
8	37	8	29	5	34	5
8	37	8	37	5	34	5

Interleaved Execution of Threads

Thread 1:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 2:

```
load sp -> reg1
load ep -> reg2
add reg1, reg2 -> reg1
store reg1 -> sp
```

Thread 1 private			Shared	Thread 2 private		
ep	reg1	reg2	sp	ep	reg1	reg2
8	0	0	29	5	0	0
8	29	0	29	5	0	0
8	29	0	29	5	29	0
8	29	8	29	5	29	0
8	29	8	29	5	29	5
8	37	8	29	5	29	5
8	37	8	29	5	34	5
8	37	8	37	5	34	5
8	37	8	**34**	5	34	5

Race Condition / Data Race

Definition:

- A *race condition / data race* exists if the behaviour (the meaning) of a program depends on the execution order of program parts (threads) whose temporal behaviour is beyond control.

Race Condition / Data Race

Definition:

- A *race condition / data race* exists if the behaviour (the meaning) of a program depends on the execution order of program parts (threads) whose temporal behaviour is beyond control.

Origin of term:

- Electronics
- Two electric signals race against each other.
- Arrival order of input signals at gate determines output signal.

Race Condition / Data Race

Definition:

- A *race condition / data race* exists if the behaviour (the meaning) of a program depends on the execution order of program parts (threads) whose temporal behaviour is beyond control.

Origin of term:

- Electronics
- Two electric signals race against each other.
- Arrival order of input signals at gate determines output signal.

Big question: how can we avoid data races ?

How can we Avoid Data Races ?

Solution: critical sections

- Once a thread enters a critical section, it must leave it before any other thread can enter; no interleaving.
- Critical sections must be made explicit throughout program.

Example:

```
double scalar_prod( double *a, double *b, int len)
{
  int i; double ep, sp=0.0;

  #pragma omp parallel for shared( a, b, len, sp) \
                           private( i, ep)
  for (i=0; i<len; i++) {
    ep = a[i] * b[i];

    #pragma omp critical
    {
      sp = sp + ep;
  } }
  return sp;
}
```

Critical Sections

Parallel scalar product:

```
double scalar_prod( double *a, double *b, int len)
{
  int i; double ep, sp=0.0;

  #pragma omp parallel for shared( a, b, len, sp) \
                           private( i, ep)
  for (i=0; i<len; i++) {
    ep = a[i] * b[i];

    #pragma omp critical
    {
      sp = sp + ep;
  } }
  return sp;
}
```

The critical directive:

- Directive must immediately precede new statement block.
- Statement block is executed without interleaving.
- Directive implements critical section.

Named Critical Sections

Disadvantage:

- All critical sections in entire program are synchronised.
- Many might be unrelated.
- Unnecessary synchronisation overhead.

Named Critical Sections

Disadvantage:

- All critical sections in entire program are synchronised.
- Many might be unrelated.
- Unnecessary synchronisation overhead.

Solution: named critical directive

- Critical sections may be associated with names.
- Critical sections with identical names are synchronised.
- Critical sections with different names are executed concurrently.

Named Critical Section

Scalar product:

```
double scalar_prod( double *a, double *b, int len)
{
  int i; double ep, sp=0.0;

  #pragma omp parallel for shared( a, b, len, sp) \
                           private( i, ep)
  for (i=0; i<len; i++) {
    ep = a[i] * b[i];

    #pragma omp critical (scalar_prod)
    {
      sp = sp + ep;
  } }
  return sp;
}
```

Named Critical Section

Scalar product:

```
double scalar_prod( double *a, double *b, int len)
{
  int i; double ep, sp=0.0;

  #pragma omp parallel for shared( a, b, len, sp) \
                           private( i, ep)
  for (i=0; i<len; i++) {
    ep = a[i] * b[i];

    #pragma omp critical (scalar_prod)
    {
      sp = sp + ep;
  } }
  return sp;
}
```

BUT: Is this really efficient ?

Reduction Clause

Scalar product:

```
double scalar_prod( double *a, double *b, int len)
{
  int i; double ep, sp=0.0;

  #pragma omp parallel for shared( a, b, len) \
                           private( i, ep) \
                           reduction( +: sp)
  for (i=0; i<len; i++) {
    ep = a[i] * b[i];
    sp = sp + ep;
  }

  return sp;
}
```

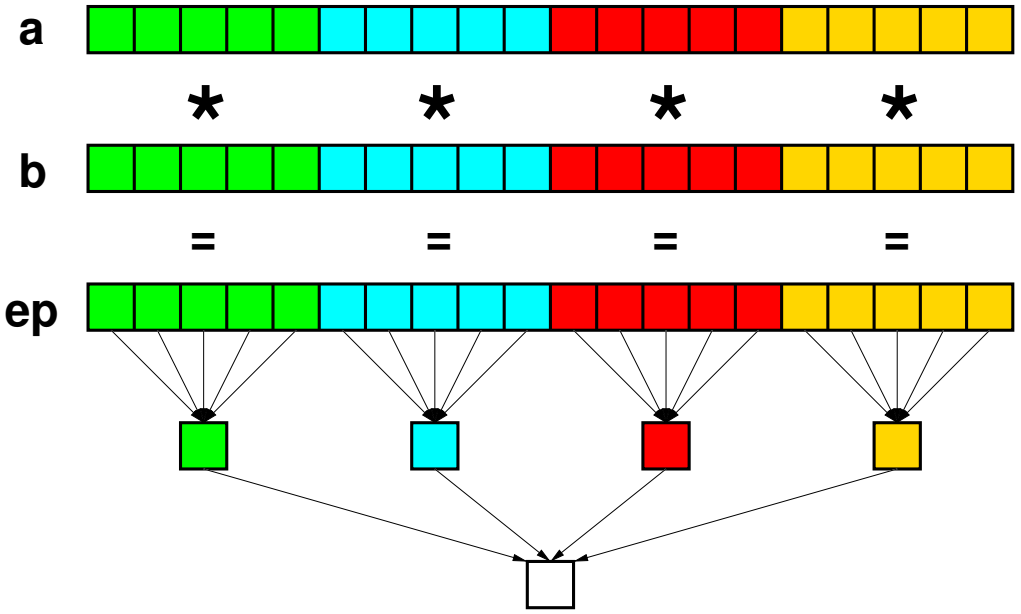
Reduction Clause

Scalar product:

```
double scalar_prod( double *a, double *b, int len)
{
  int i; double ep, sp=0.0;

  #pragma omp parallel for shared( a, b, len) \
                           private( i, ep) \
                           reduction( +: sp)
  for (i=0; i<len; i++) {
    ep = a[i] * b[i];
    sp = sp + ep;
  }

  return sp;
}
```



Reduction Clause

Scalar product:

```
double scalar_prod( double *a, double *b, int len)
{
  int i; double ep, sp=0.0;

  #pragma omp parallel for shared( a, b, len) \
                           private( i, ep) \
                           reduction( +: sp)
  for (i=0; i<len; i++) {
    ep = a[i] * b[i];
    sp = sp + ep;
  }

  return sp;
}
```

Properties:

- Reduction clause only supports built-in reduction operations: `+, *, ^, &, |, &&, ||, min, max`.
- Bit accuracy not guaranteed.

Conditional Parallelisation

Problem:

- Parallel execution of a loop incurs overhead:
  - creation of worker threads
  - scheduling
  - synchronisation barrier
- This overhead must be outweighed by sufficient workload.
- Workload depends on
  - loop body,
  - trip count.

Conditional Parallelisation

Problem:

- Parallel execution of a loop incurs overhead:
  - creation of worker threads
  - scheduling
  - synchronisation barrier
- This overhead must be outweighed by sufficient workload.
- Workload depends on
  - loop body,
  - trip count.

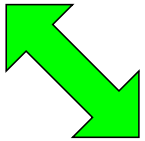
Example:

```
if (len < 1000) {
  for (i=0; i<len; i++)
  {
    res[i] = a[i] * b[i];
  }
}
else {
  #pragma omp parallel for
  for (i=0; i<len; i++)
  {
    res[i] = a[i] * b[i];
  }
}
```

Conditional Parallelisation

Introducing the `if`-clause:

```
if (len < 1000) {
  for (i=0; i<len; i++) {
    res[i] = a[i] * b[i];
  }
}
else {
  #pragma omp parallel for
  for (i=0; i<len; i++) {
    res[i] = a[i] * b[i];
  }
}
```



```
#pragma omp parallel for if (len >= 1000)
for (i=0; i<len; i++) {
  res[i] = a[i] * b[i];
}
```

Conditional Parallelisation

If-clause:

```
#pragma omp parallel for if (len >= 1000)
for (i=0; i<len; i++) {
  res[i] = a[i] * b[i];
}
```

Some facts:

- If-clause can contain any kind of C expression
- C expression may refer to all identifiers in scope
- C expression is evaluated first:
  - false: sequential execution
  - true: parallel execution

Programming Multi-Core Systems with OpenMP

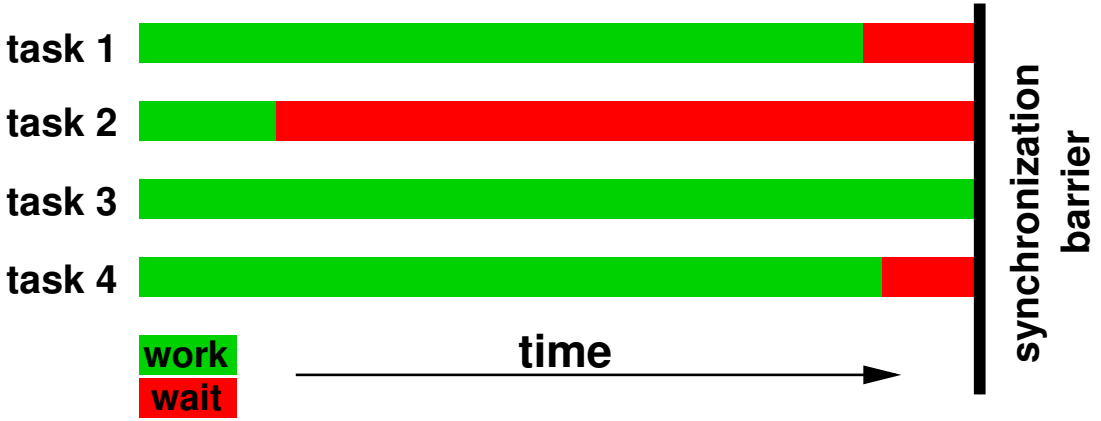
Loop Scheduling

Definition:

- Loop scheduling determines which iterations are executed by which thread.

Aim:

- Equal workload distribution



Loop Scheduling

Problem:

- Different situations require different techniques

The `schedule` clause:

```
#pragma omp parallel for schedule( <type> [, <chunk>])
for (...)
{
   ...
}
```

Properties:

- Clause selects one out of a set of scheduling techniques.
- Optionally, a chunk size can be specified.
- Default chunk size depends on scheduling technique.

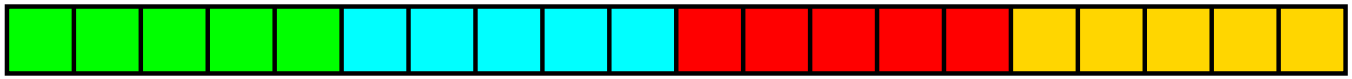
Loop Scheduling

Static scheduling:

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

- Loop is subdivided into as many chunks as threads exist.
- Often called block scheduling.

Illustration:



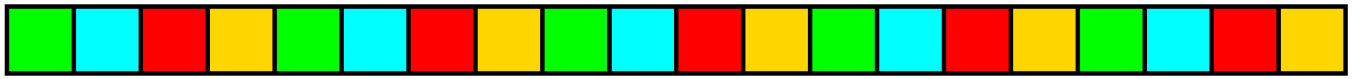
Loop Scheduling

Static scheduling with chunk size 1:

```
#pragma omp parallel for schedule( static, 1)
```

- Iterations are assigned to threads in a round-robin fashion.
- Also called cyclic scheduling.

Illustration:



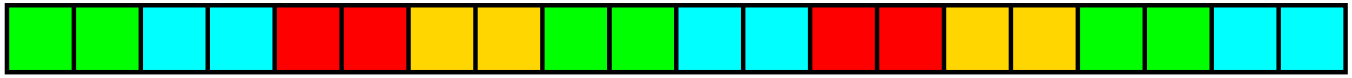
Loop Scheduling

Static scheduling with chunk size ¡n¿:

```
#pragma omp parallel for schedule( static, <n>)
```

- Loop is subdivided into chunks of *n* iterations.
- Chunks are assigned to threads in a round-robin fashion.
- Also called block-cyclic scheduling.

Illustration:

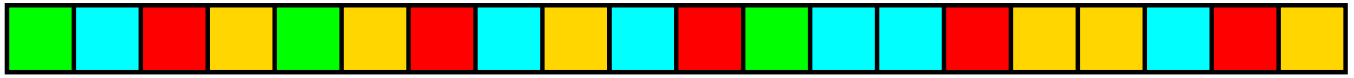


Loop Scheduling

Dynamic scheduling:

```
#pragma omp parallel for schedule( dynamic, <n>)
```

- Loop is subdivided into chunks of *n* iterations.
- Chunks are dynamically assigned to threads on their demand.
- Also called self scheduling.
- Default chunk size: 1 iteration.

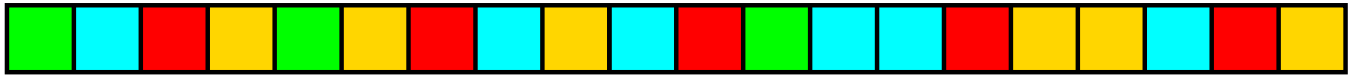


Loop Scheduling

Dynamic scheduling:

```
#pragma omp parallel for schedule( dynamic, <n>)
```

- Loop is subdivided into chunks of n iterations.
- Chunks are dynamically assigned to threads on their demand.
- Also called self scheduling.
- Default chunk size: 1 iteration.



Properties to keep in mind:

- Allows for dynamic load distribution and adjustment.
- Requires additional synchronization.
- Generates more overhead than static scheduling.

Loop Scheduling

Dilemma of chunk size selection:

- Small chunk sizes mean good load balancing, but high synchronisation overhead.
- Large chunk sizes reduce synchronisation overhead, but result in poor load balancing.

Loop Scheduling

Dilemma of chunk size selection:

- Small chunk sizes mean good load balancing, but high synchronisation overhead.
- Large chunk sizes reduce synchronisation overhead, but result in poor load balancing.

Rationale of guided scheduling:

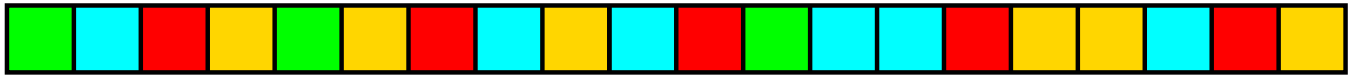
- In the beginning, large chunks keep synchronisation overhead small.
- When approaching the final barrier, small chunks balance workload.

Loop Scheduling

Guided scheduling:

```
#pragma omp parallel for schedule( guided, <n>)
```

- Chunks are dynamically assigned to threads on their demand.
- Initial chunk size is implementation dependent.
- Chunk size decreases exponentially with every assignment.
- Also called guided self scheduling.
- Minimum chunk size: n (default: 1)

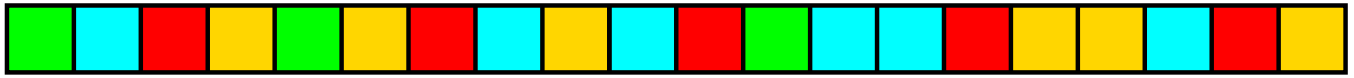


Loop Scheduling

Guided scheduling:

```
#pragma omp parallel for schedule( guided, <n>)
```

- Chunks are dynamically assigned to threads on their demand.
- Initial chunk size is implementation dependent.
- Chunk size decreases exponentially with every assignment.
- Also called guided self scheduling.
- Minimum chunk size: n (default: 1)



Example:

- Total number of iterations: 250
- Initial / minimal chunk size: 50 / 5
- Current chunk size: 80% of last chunk size:
  50 – 40 – 32 – 26 – 21 – 17 – 14 – 12 – 10 – 8 – 6 – 5 – 5 – 4
- All properties are *implementation-dependent* !

Choice of Scheduling Technique

Which scheduling to choose when ?

- Depends on your code !
- Crucial question: is the amount of computational work per iteration (roughly) the same for each iteration or not?

Choice of Scheduling Technique

Which scheduling to choose when ?

- Depends on your code !
- Crucial question: is the amount of computational work per iteration (roughly) the same for each iteration or not?

Static scheduling techniques:

- Preferable for uniform workload distributions
- Minimal overhead
- (Block-)Cyclic schedulings may be useful for regular uneven workload distributions
- (Block-)Cyclic schedulings may run into cache issues

Choice of Scheduling Technique

Which scheduling to choose when ?

- Depends on your code !
- Crucial question: is the amount of computational work per iteration (roughly) the same for each iteration or not?

Static scheduling techniques:

- Preferable for uniform workload distributions
- Minimal overhead
- (Block-)Cyclic schedulings may be useful for regular uneven workload distributions
- (Block-)Cyclic schedulings may run into cache issues

Dynamic scheduling techniques:

- Preferable for irregular workload distributions
- Additional synchronisation overhead needs compensation
- Guided self-scheduling usually superior

Programming Multi-Core Systems with OpenMP

What's More ?

More in OpenMP-2:

- Decouple parallel regions from work sharing
- Control synchronisation barriers
- Task parallel sections
- Low-level locks and condition variables
- ...

What's More ?

More in OpenMP-2:

- Decouple parallel regions from work sharing
- Control synchronisation barriers
- Task parallel sections
- Low-level locks and condition variables
- ...

More in OpenMP-3:

- Nested parallel regions
- Spawning and synchronisation of tasks
- ...

What's More ?

More in OpenMP-2:

- Decouple parallel regions from work sharing
- Control synchronisation barriers
- Task parallel sections
- Low-level locks and condition variables
- ...

More in OpenMP-3:

- Nested parallel regions
- Spawning and synchronisation of tasks
- ...

More information:

- www.openmp.org

The End: Questions ?