Parallel Computing

Introduction and Shared Memory Programming

Dr. Georg Hager, Dr. Gerhard Wellein
Regionales Rechenzentrum Erlangen (RRZE)

Vorlesung „Parallelrechner“
Georg-Simon-Ohm-Fachhochschule Nürnberg
28.02.-05.03.2007

Outline

- **Part 1**
  - Introduction, motivation
  - Understanding parallelism
  - Limitations of parallelism
- **Part 2**
  - Shared Memory architectures
  - Some comments about multi-core
  - Cache coherence
  - Introduction to OpenMP as an example for shared memory programming

Introduction

Parallel Computing

- **Parallelism will substantially increase through the use of dual/multi-core chips in the future!**
  - See later for some comments

- **Parallel computing is entering everyday life**
  - Dual-core based system (Workstation, Laptop, etc…)

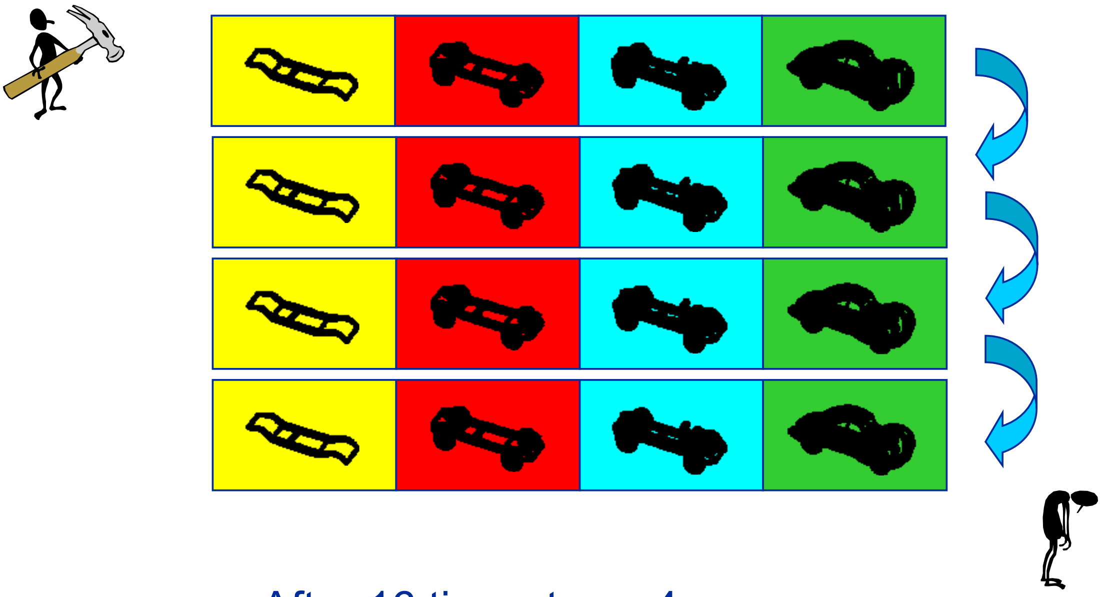
- **Basic design concepts for parallel computers:**
  - ***Shared memory multi-processor systems***: Multiple processors run in parallel but use the same (a single) address space (“shared memory”), e.g.: Dual-core workstations or Xeon/Opteron based servers.

  - ***Distributed memory systems:*** Multiple processors/compute nodes are connected via a network. Each processor has its own address space/ memory, e.g. GBit Clusters with Xeon/Opteron based servers.

Understanding Parallelism and the Limitations of Parallel Computing

Understanding Parallelism:

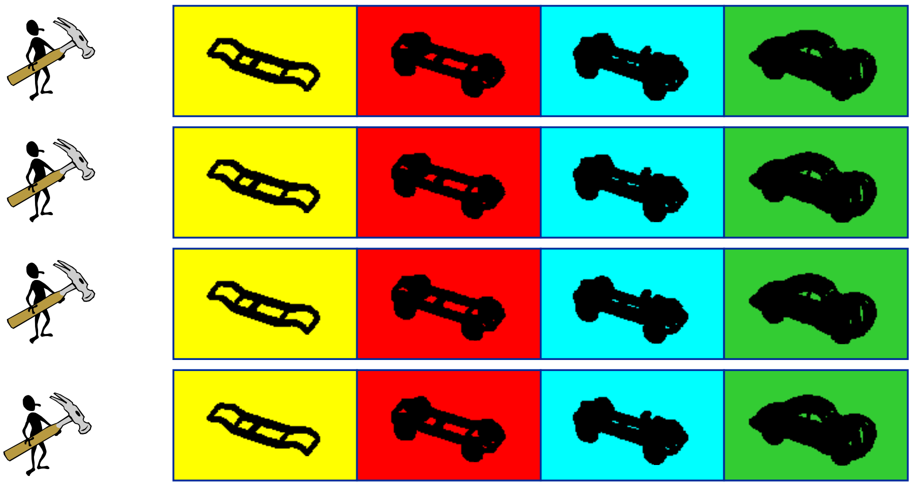
Sequential work



After 16 time steps: 4 cars

Understanding Parallelism:

Parallel work

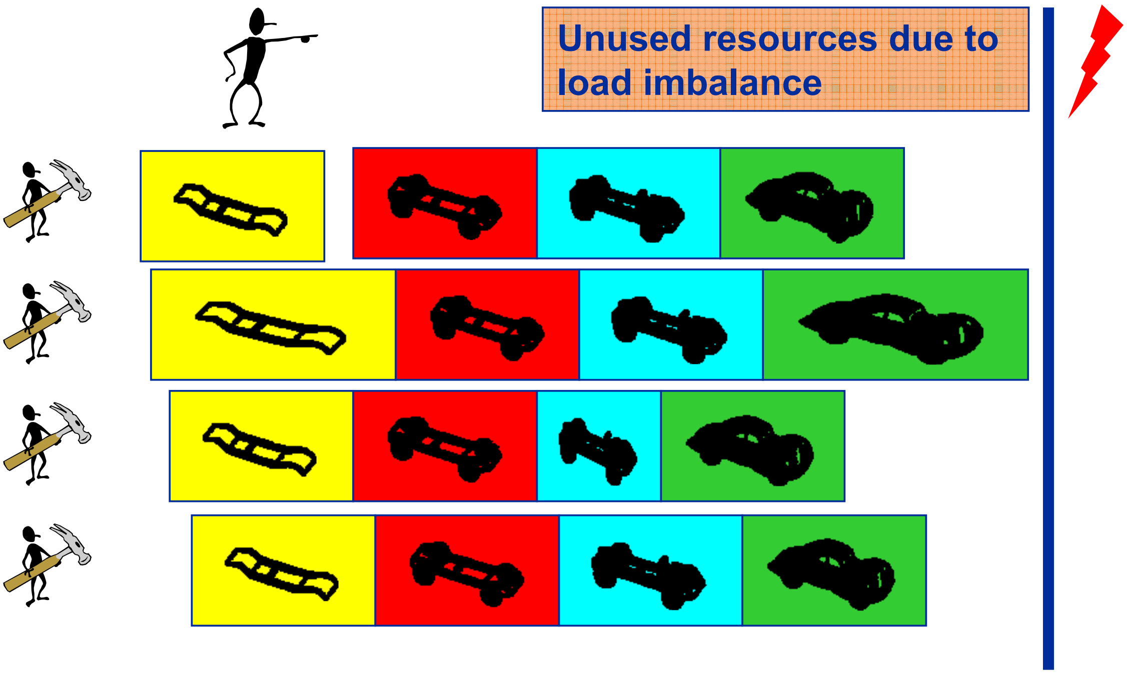


After 4 time steps: 4 cars

"*perfect speedup*"

Understanding Parallelism:

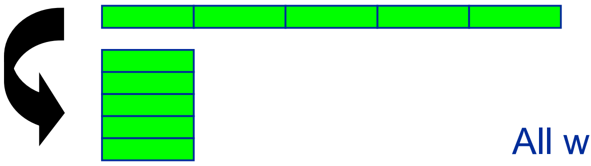
Limits of Scalability



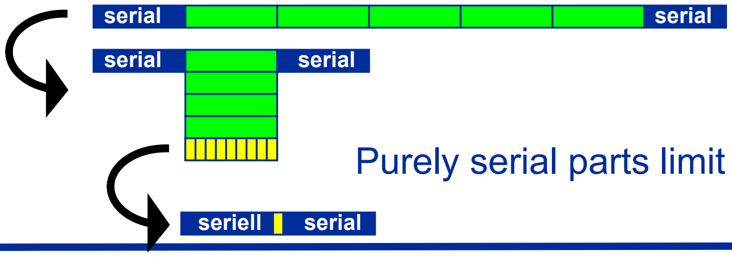
Unused resources due to load imbalance

Limitations of Parallel Computing:

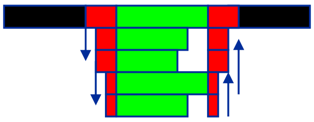
Amdahl's Law



Ideal world:
All work is perfectly parallelizable



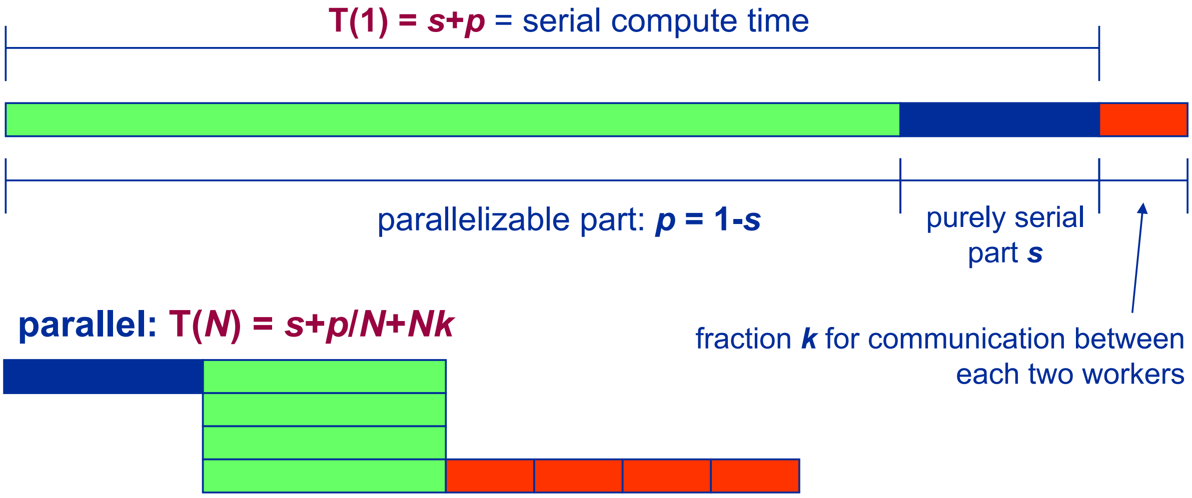
Closer to reality:
Purely serial parts limit maximum speedup



Reality is even worse:
Communication processes hurt
scalability even further

Limitations of Parallel Computing:

Calculating Speedup in a Simple Model (“strong scaling”)



T(1) = *s*+*p* = serial compute time

parallelizable part: ***p* = 1-*s***

purely serial part ***s***

fraction ***k*** for communication between each two workers

parallel: T(*N*) = *s*+*p*/*N*+*Nk*

General formula for speedup (worst case):
***k*=0: Amdahl's Law "strong scaling"**

$$S_p^k = \frac{T(1)}{T(N)} = \frac{1}{s+\frac{1-s}{N}+Nk}$$

Limitations of Parallel Computing:

Amdahl's Law (“strong scaling”)

- **Reality: No task is perfectly parallelizable**
  - **Shared resources have to be used serially**
  - **Task interdependencies must be accounted for**
  - **Communication overhead**
- **Benefit of parallelization is strongly limited**
  - **"Side effect": limited scalability leads to inefficient use of resources**
  - **Metric: Parallel Efficiency**
    (*what percentage of the workers/processors is efficiently used*):

$$\varepsilon_p(N) = \frac{S_p(N)}{N}$$

- **Amdahl case:** $\varepsilon_p = \frac{1}{s(N-1)+1}$

Limitations of Parallel Computing:
Amdahl's Law (“strong scaling”)

- Large N limits
  - **at k=0, Amdahl's Law predicts**

$$\lim_{N\to\infty} S_p^0(N) = \frac{1}{s}$$

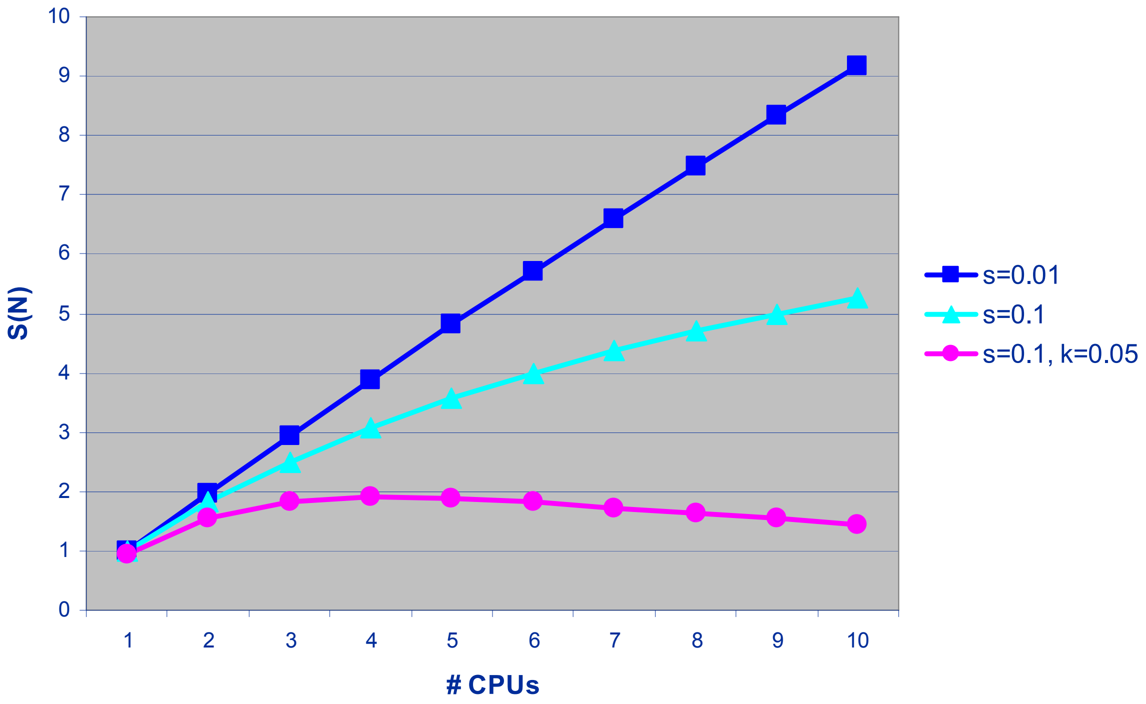
independent of *N !*

  - **at k≠0, our simplified model of communication overhead yields a beaviour of**

$$S_p^k(N) \xrightarrow{N>>1} \frac{1}{Nk}$$

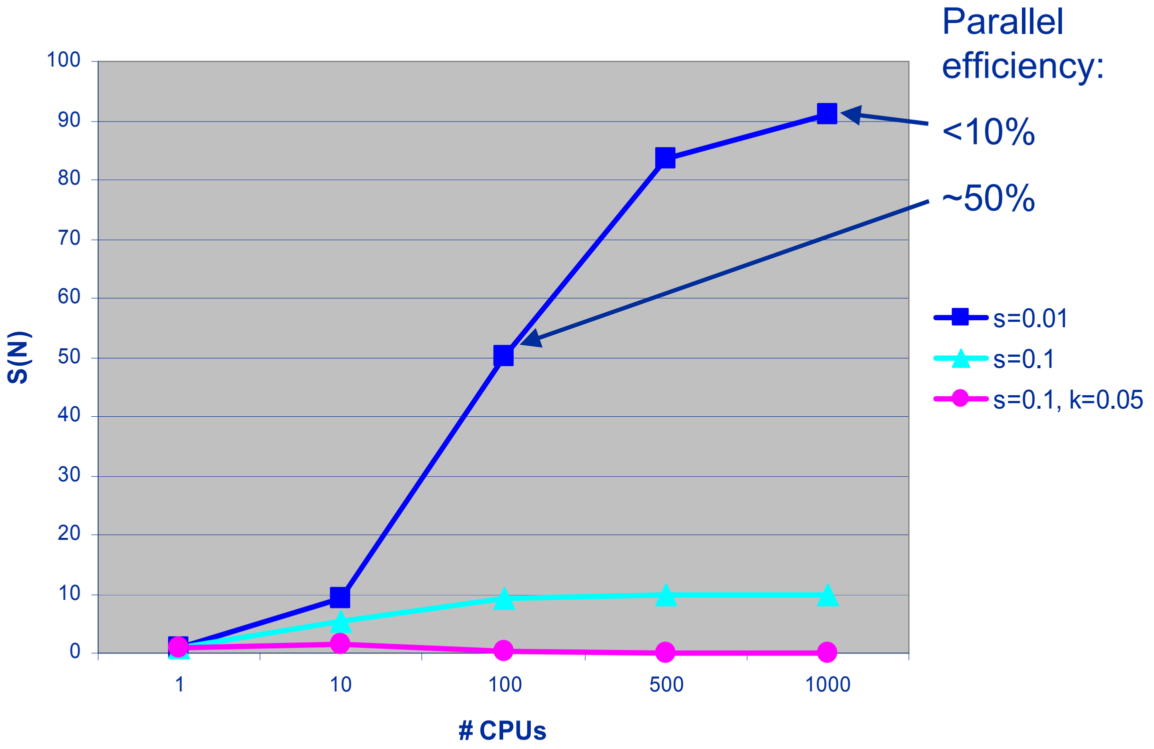
- Problems in real world programming
  - **Load imbalance**
  - **Shared resources have to be used serially (e.g. IO)**
  - **Task interdependencies must be accounted for**
  - **Communication overhead**

Limitations of Parallel Computing:
Amdahl´s Law (“strong scaling”)



Limitations of Parallel Computing:

Amdahl´s Law (“strong scaling”)



Parallel efficiency:

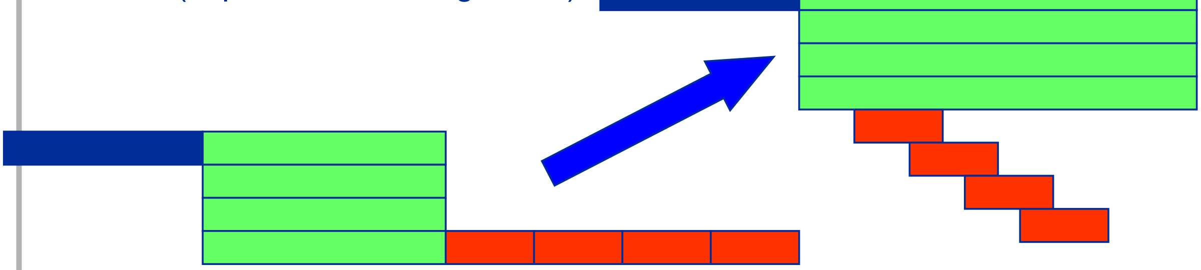
<10%

~50%

Limitations of Parallel Computing:

How to Circumvent Amdahl's Law

- **Communication is not necessarily purely serial**
  - Non-blocking crossbar networks can transfer many messages concurrently – factor *Nk* in denominator becomes *k* (technical measure)
  - Sometimes, communication can be overlapped with useful work (implementation, algorithm):

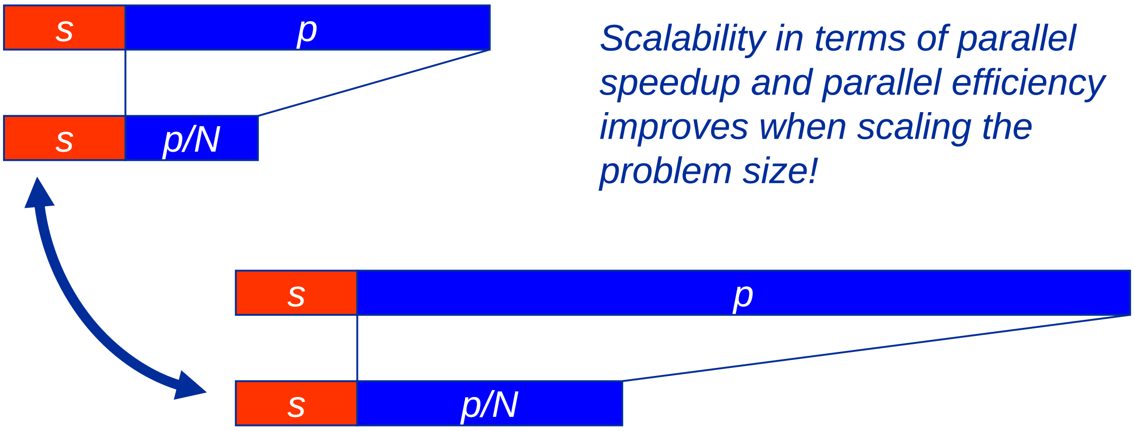


  - Communication overhead may scale with a smaller power than problem size
  - "superlinear speedups“: data size per CPU decreases with increasing CPU count -> may fit into cache at large CPU counts

Limitations of Parallel Computing:

Increasing Parallel Efficiency

- **Increasing problem size often helps to enlarge the parallel fraction *p***
  - Often *p* scales with problem size while s stays constant
  - Fraction of *s* relative to overall runtime decreases



Scalability in terms of parallel speedup and parallel efficiency improves when scaling the problem size!

Limitations of Parallel Computing:

Increasing Parallel Efficiency („weak scaling“)

- **When scaling a problem to more workers, the amount of work will often be scaled as well**
  - **Let *s* and *p* be the serial and parallel fractions so that *s+p*=1**
  - **Perfect situation: runtime stays constant while *N* increases**
  - „Parallel Performance“ **=**

$$\frac{\textbf{work/time for problem size } N \textbf{ with } N \textbf{ workers}}{\textbf{work/time for problem size 1 with 1 worker}}$$

$$\boxed{P_s(N) = \frac{s+pN}{s+p} = s + pN = N + (1-N)s = s + (1-s)N}$$

Gustafsson's Law ("weak scaling")

- Linear in *N* – but closely observe the meaning of the word "work"!

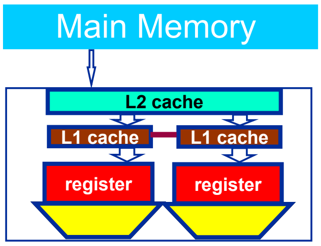
Architecture of shared memory computers

Shared memory computers: Basic concepts

- **Shared Memory Computer provides single, shared address space for all parallel processors**
- **Two basic categories of shared memory systems**
  - Uniform Memory Access (UMA):
    - Flat Memory: Memory is equally accessible to all processors with the same performance (Bandwidth & Latency).
    - A.k.a Symmetric Multi Processor (SMP) system
  - Cache-Coherent Non Uniform Memory Access (ccNUMA):
    - Memory is physically distributed: Performance (Bandwidth & Latency) is different for local and remote memory access.
- **Cache-Coherence protocols and/or hardware provide consistency between data in caches (multiple copies of same data!) and data in memory**

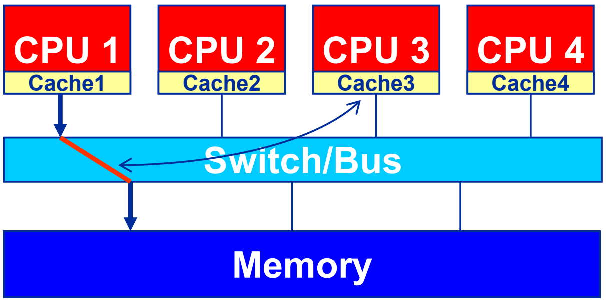
Shared memory computers: UMA

UMA architecture



Simplest implementation: Dual-Core Processor (e.g. AMD Opteron dual-core or Intel Core)

Multi-Processor servers use bus or switch to connect CPUs with main memory



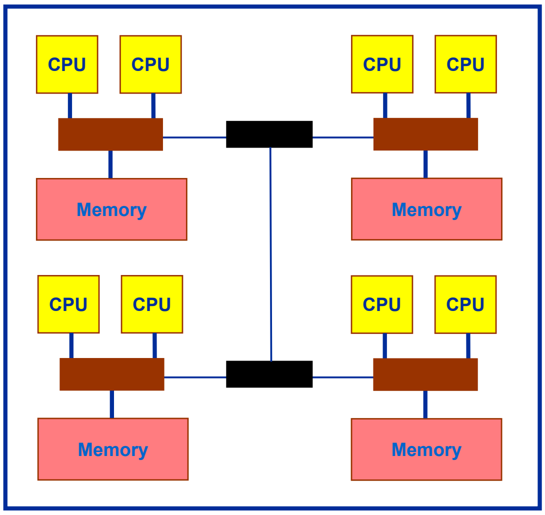
- Bus: Only one processor can access bus at a time!
- Switch: Cache-Coherency traffic can “pollute” switch
- Scalability beyond 2–8 CPUs is a problem
- Dual core chips, small Itanium servers, NEC SX8

Shared memory computers: ccNUMA

ccNUMA architecture

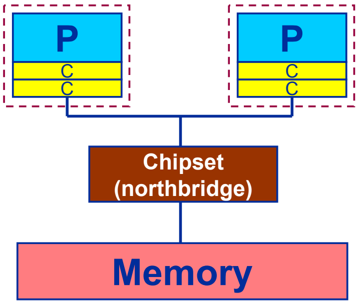
Proprietary hardware concepts (e.g. Hypertransport/Opteron or NUMALink /SGI) provide single address space & cache coherency for physically distributed memory

- **Advantages:**
  - Scalable concept (systems up to 1024 CPUs are available)
- **Disadvantages:**
  - Cache Coherence hard to implement / expensive
  - Performance depends on access to local or remote memory
    (no flat view of memory!)

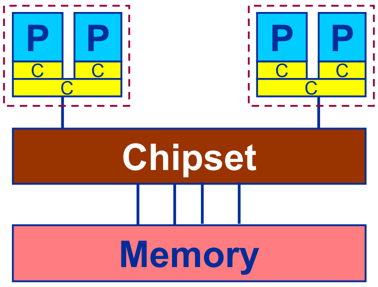


Shared memory computers: Some examples

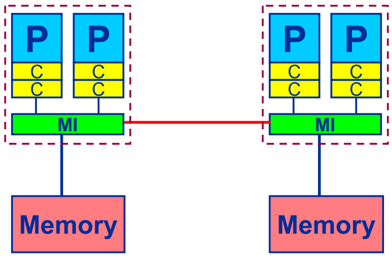
- Dual CPU Intel Xeon node



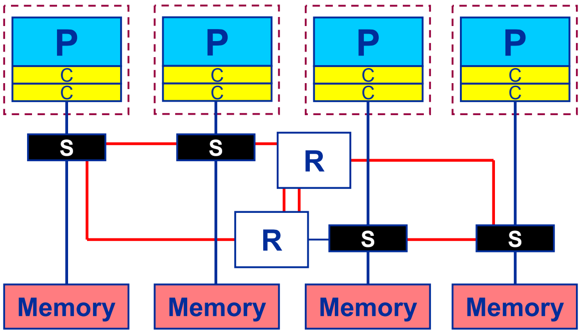
- Dual Intel “Core 2” node



- Dual AMD Opteron node

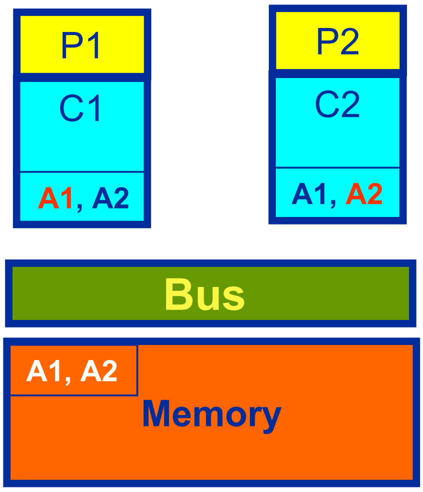


- SGI Altix (HLRB2 @ LRZ)



Shared memory computers Cache coherence

- Data in cache is only a copy of data in memory
  - Multiple copies of same data on multiprocessor systems
  - Cache coherence protocol/hardware ensure consistent data view
  - Without cache coherence, shared cache lines can become clobbered:



P1

```
Load A1
Write A1=0
```

P2

```
Load A2
Write A2=0
```

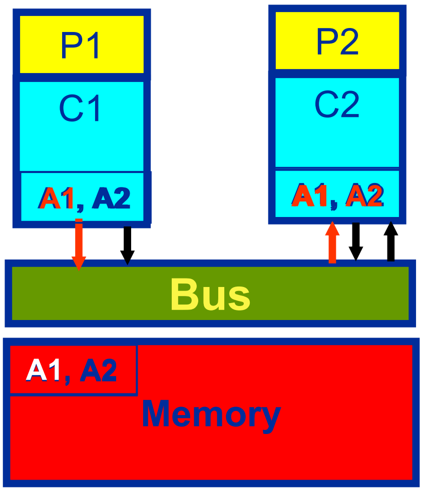
Write-back to memory leads to incoherent data

A1, A2 | A1, A2 | A1, A2

C1 & C2 entry can not be merged to: A1, A2

Shared Memory Computers
Cache coherence

- **Cache coherence protocol must keep track of cache line (CL) status**



P1

`Load A1`

`Write A1=0:`

1. Request exclusive access to CL

2. Invalidate CL in C2

3. Modify A1 in C1

P2

`Load A2`

`Write A2=0:`

1. Request exclusive CL access

2. CL write back+ Invalidate

3. Load CL to C2

4. Modify A2 in C2

t

C2 is exclusive owner of CL

Shared Memory Computers
Cache coherence

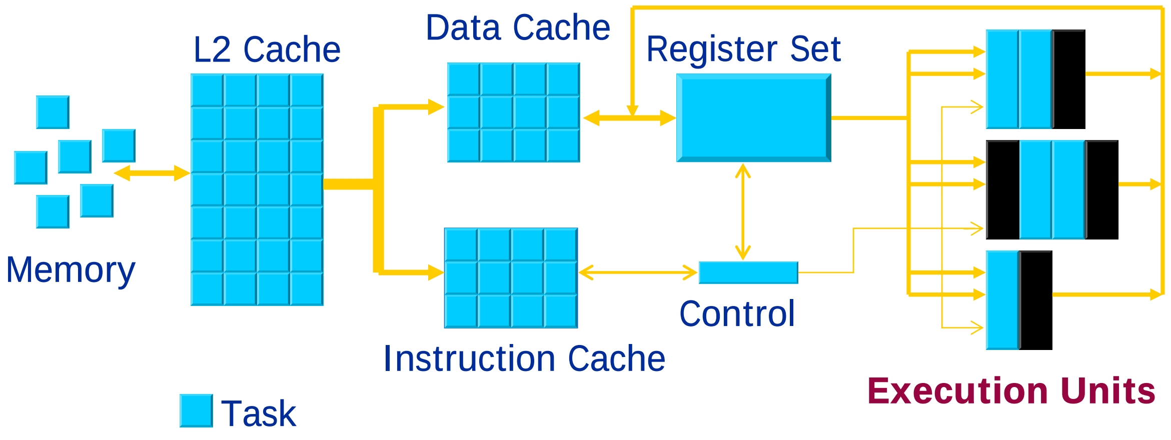
- **Cache coherence can cause substantial overhead**
  - may reduce available bandwidth
- **Different implementations**
  - Snoopy: On modifying a CL, a CPU must broadcast its address to the whole system
  - Directory, “snoop filter”: Chipset (“network”) keeps track of which CLs are where and filters coherence traffic
- **Directory-based ccNUMA can reduce pain of additional coherence traffic**
- **But always take care:**

Multiple processors should never write frequently to the same cache line (“*false sharing*”)!

Why Multi-Core?

Why Multi-Core?

- **Modern processors are highly complex**
- **With each new generation, more transistors are required to achieve a certain performance gain**
  - Even highly optimized software leaves more and more transistors unused



© Intel

- **All those transistors need energy (switching/leakage)**

Power dissipation in VLSI Circuits

- In CMOS VLSIs, power dissipation is proportional to clock frequency: $W \propto f_c$
- Moreover, it is proportional to supply voltage squared: $W \propto V_{cc}^2$
- For reasons of noise immunity, supply voltage has to grow linearly with frequency, so: $\boxed{W \propto f_c^3}$
- Frequency reduction is the key to saving power with modern microprocessors
  - all other factors, e.g. manufacturing technology, unchanged
- This seems to contradict the verdict of ever-growing chip performance

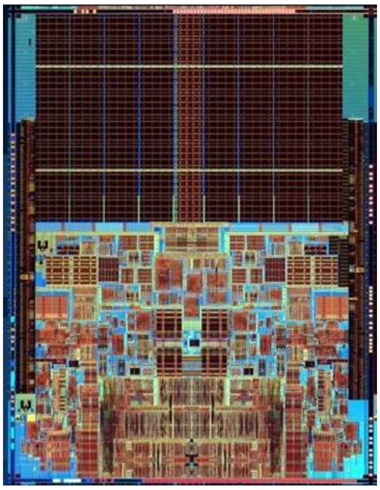
Multi-core processors
The party is over!

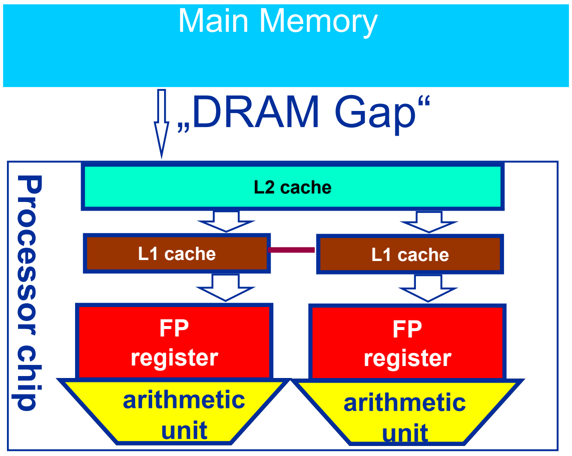
- **Problem:** Moore’s law is still valid but increasing clock speed hits a technical wall (heat)
- **Solution:** Reduce clock speed of processor but put 2 (or more) processors (cores) on a single silicon die

Clock speed of single core will decrease in future!

(Xeon/Netburst: max. 3.73 GHz -> Xeon/Core: max. 3.0 GHz)

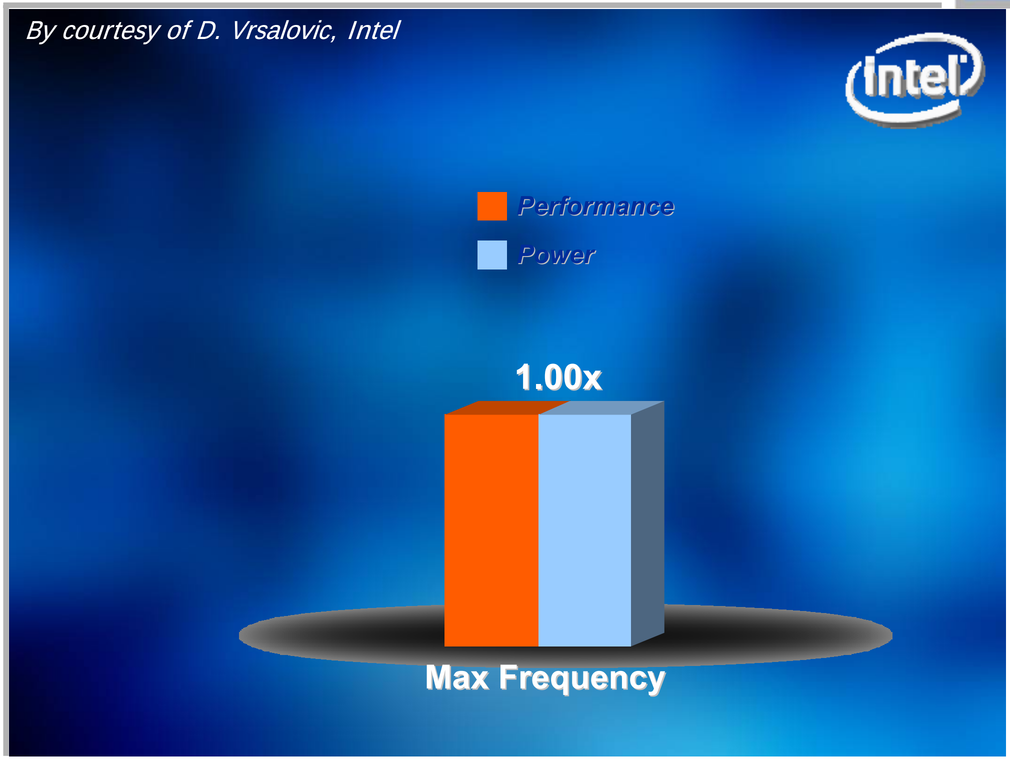
Intel Xeon / Core2 (“Woodcrest”)





Multi-core processors

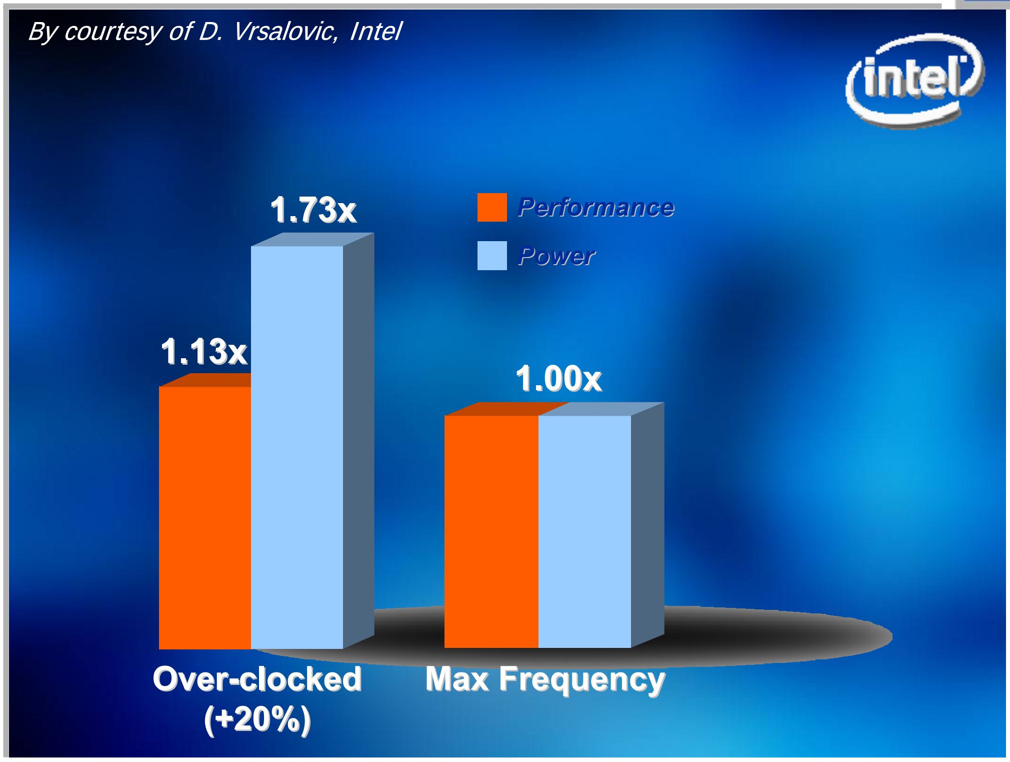
The party is over!



By courtesy of D. Vrsalovic, Intel

Multi-core processors

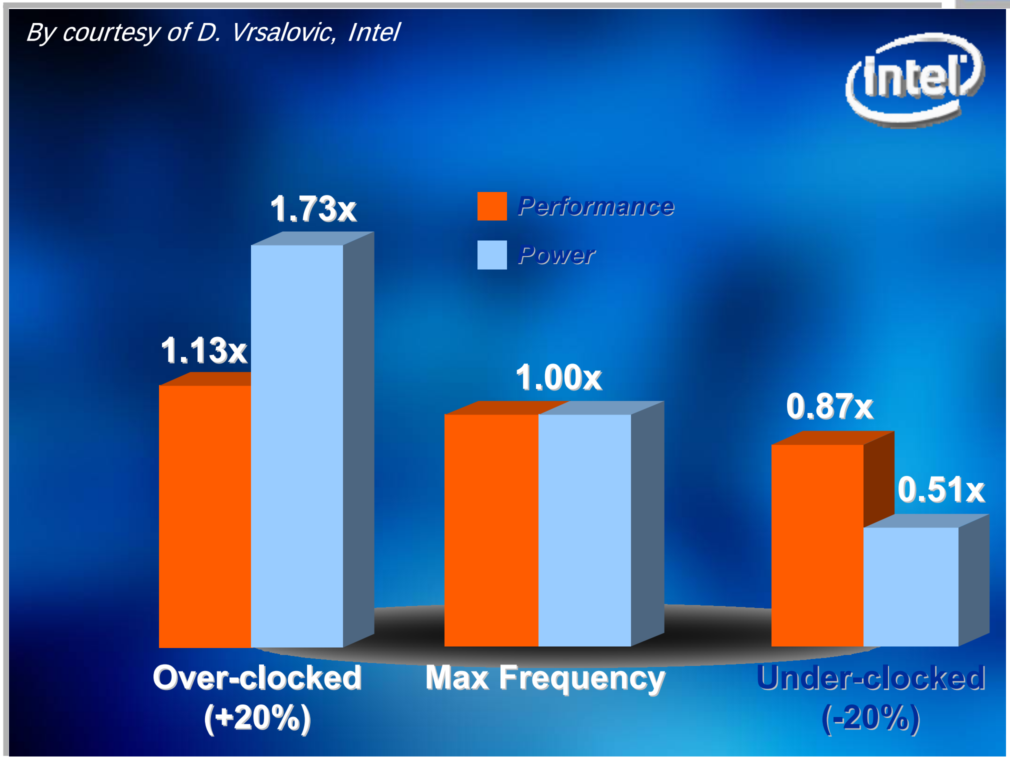
The party is over!



By courtesy of D. Vrsalovic, Intel

Multi-core processors

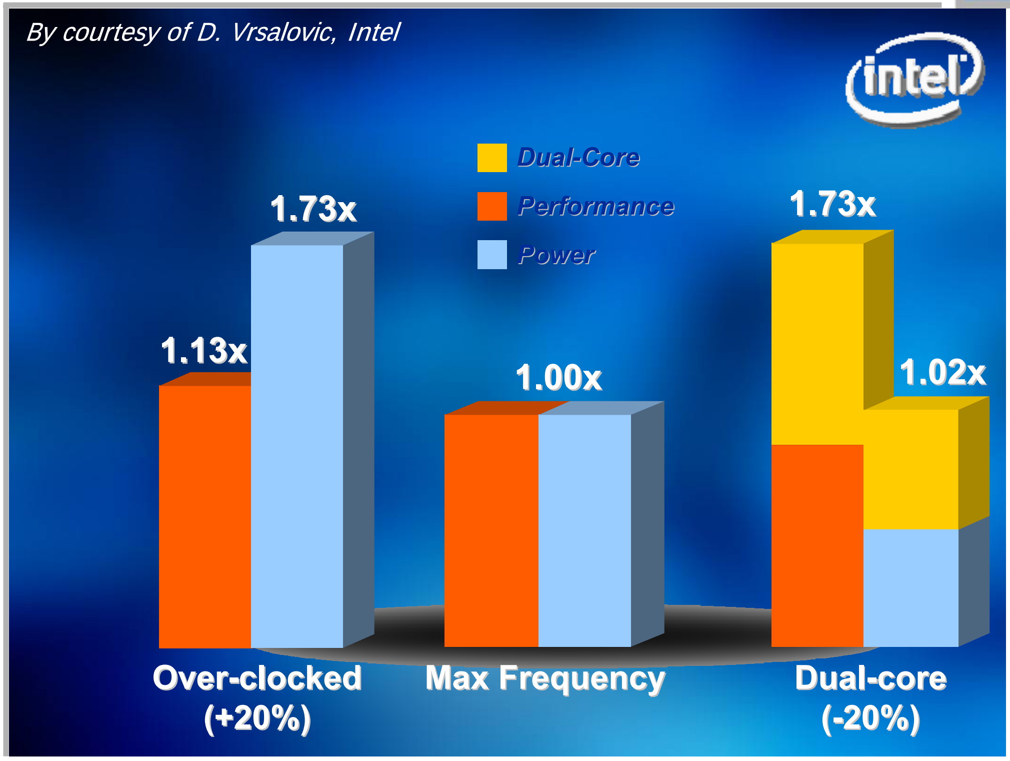
The party is over!



By courtesy of D. Vrsalovic, Intel

Multi-core processors

The party is over!



By courtesy of D. Vrsalovic, Intel

Multi-Core Processors

- **Question: What fraction of performance must be sacrificed per core in order to benefit from *m* cores?**
- **Prerequisite: Overall power dissipation should be unchanged**

$$W+\Delta W=(1+\varepsilon_f)^3 W$$

$$(1+\varepsilon_f)^3 m=1$$

$$\boxed{\varepsilon_f=m^{-1/3}-1}$$

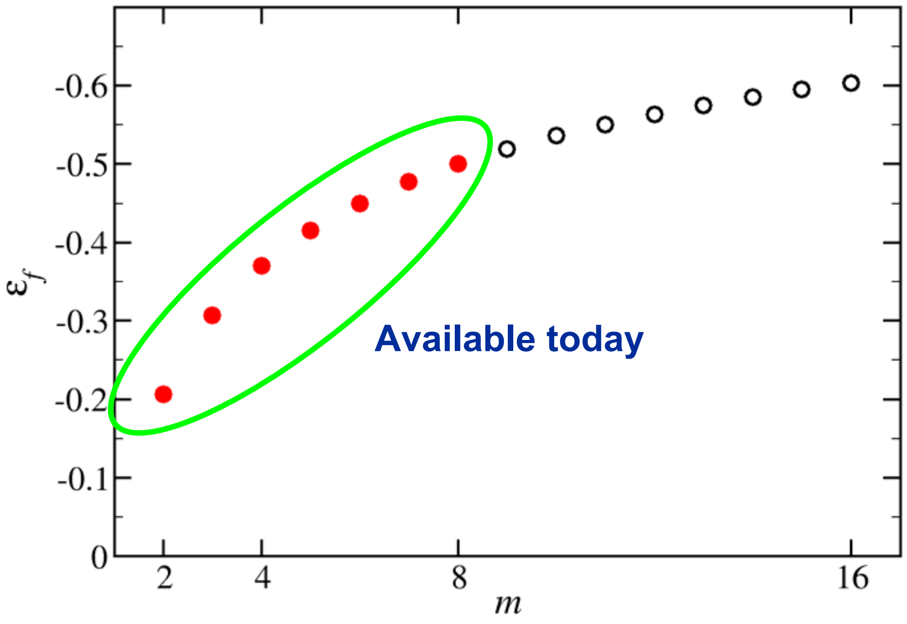
$$p_m=(1+\varepsilon_p)\,pm$$

$$\boxed{p_m\ge p \;\Rightarrow\; \varepsilon_p\ge\frac{1}{m}-1}$$

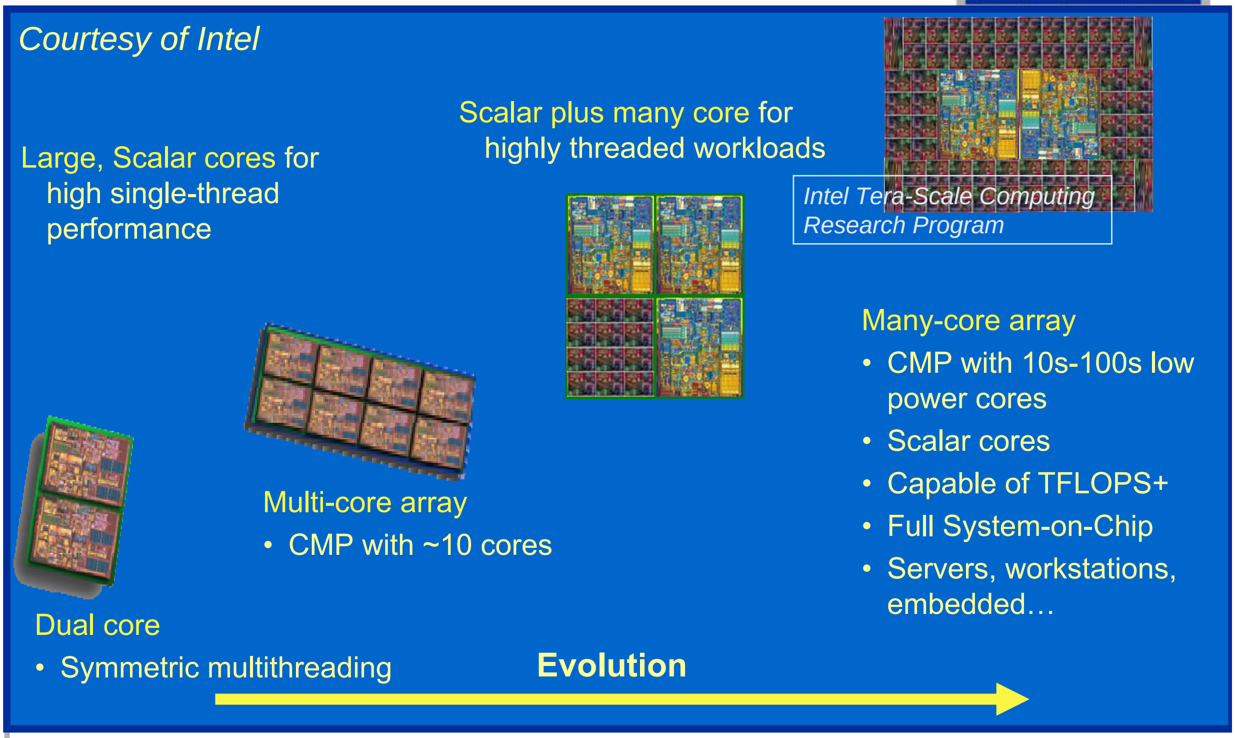
W	power dissipation
p	performance (1 core)
p_m	performance (*m* cores)
ε_f	rel. frequency change $\Delta f_c/f_c$
ε_p	rel. performance change $\Delta p/p$
m	number of cores

Why Multi-Core?

- **Required relative frequency reduction vs. core count**



Multi-core processors
A challenging future ahead?



Courtesy of Intel

Large, Scalar cores for high single-thread performance

Scalar plus many core for highly threaded workloads

Intel Tera-Scale Computing Research Program

Many-core array
- CMP with 10s-100s low power cores
- Scalar cores
- Capable of TFLOPS+
- Full System-on-Chip
- Servers, workstations, embedded…

Multi-core array
- CMP with ~10 cores

Dual core
- Symmetric multithreading

Evolution

Parallelization will be mandatory in the future !

Multi-Core
A Challenging Future or the Programmer‘s Waterloo?

- **Multi-core does not come for free**
  - i.e., frequency reduction is not enough
- **Putting two cores on the same die requires either**
  - changes in manufacturing technology (smaller structures), or
  - simplification of the core
- **Moore‘s Law is still valid, so multi-core must put the transistors to good use**
  - Simplify the core (better utilization of functional units)
  - Increase the cache sizes using the saved transistors
- **Are we giving up the „general-purpose“ processor for more and more specialized solutions?**

- **Caveat: While multi-core enhances chip performance, it makes the DRAM gap more severe**
  - Shared path to memory

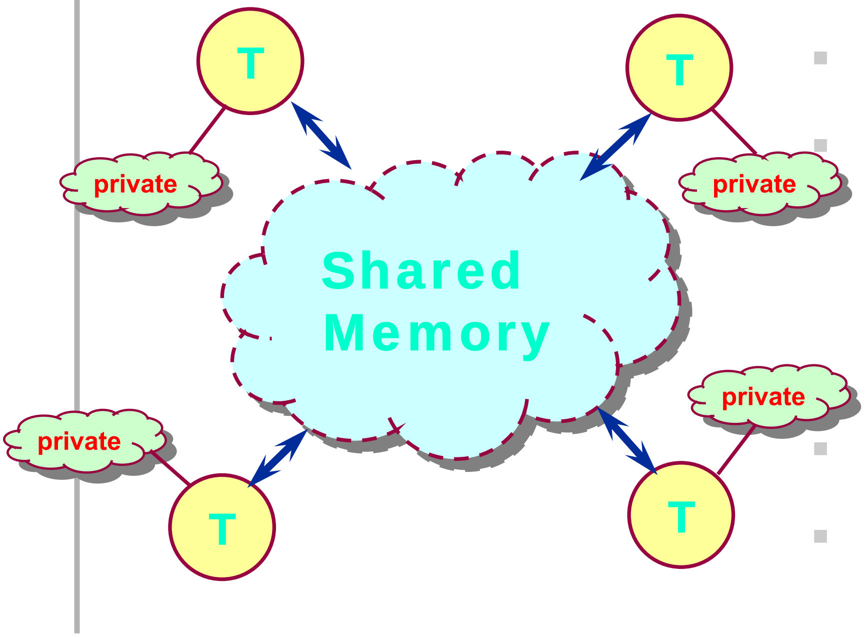
Shared-Memory Parallelization with OpenMP

Parallel Programming with OpenMP

- **“Easy” and portable parallel programming of shared memory computers: OpenMP**
- **Standardized set of compiler directives & library functions:** `http://www.openmp.org/`
  - FORTRAN, C and C++ interfaces are defined
  - Supported by most/all commercial compilers, GNU starting with 4.2
  - Few free tools are available
- **OpenMP program can be written to compile and execute on a single-processor machine just by ignoring the directives**
  - API calls must be masked out though
  - Supports data parallelism
- **R.Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon: Parallel programming in OpenMP. Academic Press, San Diego, USA, 2000, ISBN 1-55860-671-8**

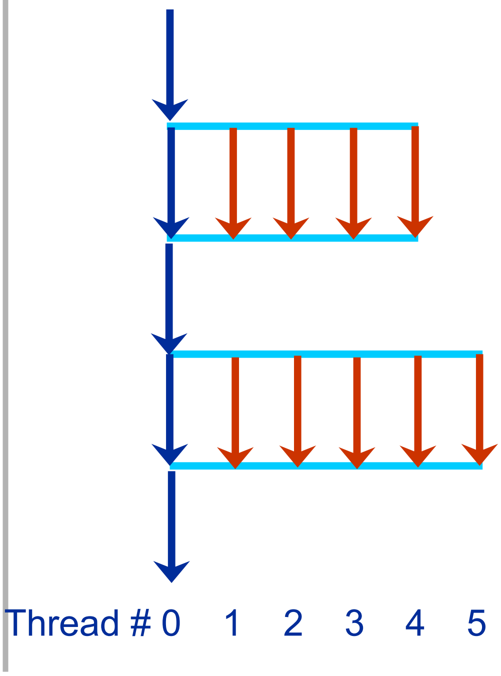
Shared Memory Model used by OpenMP

- **Central concept of OpenMP programming: Threads**



- **Threads access globally shared memory**
- **Data can be shared or private**
  - shared data available to all threads (in principle)
  - private data only to thread that owns it
- **Data transfer transparent to programmer**
- **Synchronization takes place, is mostly implicit**

OpenMP Program Execution
Fork and Join



- **Program start: only master thread runs**
- **Parallel region: team of worker threads is generated (“fork”)**
- **synchronize when leaving parallel region (“join”)**
- **Only master executes sequential part**
  - worker threads persist, but are inactive
- **task and data distribution possible via directives**
- **Usually optimal:**
  1 Thread per Processor

Basic OpenMP functionality

About Directives and Clauses

About Data

About Parallel Regions and Work Sharing

First example: Numerical integration

Approximate by a discrete sum

$$\int_0^1 f(t)\,dt \approx \frac{1}{n}\sum_{i=1}^{n} f(x_i)$$

where

$$x_i = \frac{i-0.5}{n} \quad (i=1,\ldots,n)$$

We want

$$\int_0^1 \frac{4\,dx}{1+x^2} = \pi$$

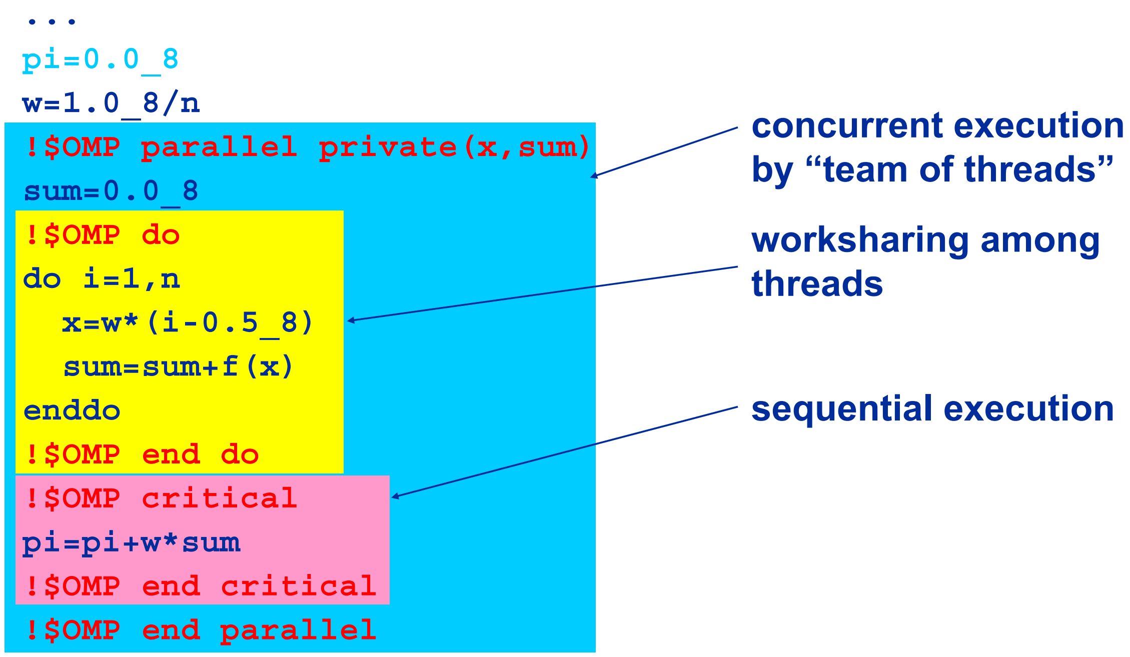
→ **solve this in OpenMP**

```
program compute_pi
...   (declarations omitted)

! function to integrate
f(a)=4.0_8/(1.0_8+a*a)
```

```
w=1.0_8/n
sum=0.0_8

do i=1,n
   x=w*(i-0.5_8)
   sum=sum+f(x)
enddo
pi=w*sum
```

```
...    (printout omitted)
end program compute_pi
```

First example: Numerical integration



```
...
pi=0.0_8
w=1.0_8/n
!$OMP parallel private(x,sum)
sum=0.0_8
!$OMP do
do i=1,n
  x=w*(i-0.5_8)
  sum=sum+f(x)
enddo
!$OMP end do
!$OMP critical
pi=pi+w*sum
!$OMP end critical
!$OMP end parallel
```

concurrent execution by “team of threads”

worksharing among threads

sequential execution

OpenMP Directives Syntax in Fortran

- **Each directive starts with sentinel in column 1:**
  - fixed source: `!$OMP` or `C$OMP` or `*$OMP`
  - free source: `!$OMP`

  followed by a directive and, optionally, clauses.
- **For function calls:**
  - conditional compilation of lines starting with `!$` or `C$` or `*$`

  Example:

```
   myid = 0
!$ myid = omp_get_thread_num()
```

  - use include file for API call prototypes (or Fortran 90 module `omp_lib` if available)

OpenMP Directives
Syntax in C/C++

- **Include file**

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

- **pragma preprocessor directive:**

```
#pragma omp [directive [clause ...]]
  structured block
```

- **Conditional compilation: Compiler’s OpenMP switch sets preprocessor macro**

```
#ifdef _OPENMP

... do something

#endif
```

OpenMP Syntax:
Clauses

- **Many (but not all) OpenMP directives support clauses**
- **Clauses specify additional information with the directive**
- **Integration example:**
  - `private(x,sum)` appears as clause to the `parallel` directive
- **The specific clause(s) that can be used depend on the directive**
- **Another example: `schedule(…)` clause**
  - `static[,chunksize]`: round-robin distribution of chunks across threads (no chunksize: max. chunk size – default!)
  - `dynamic[,chunksize]`: threads get assigned work chunks dynamically; used for load balancing
  - `guided[,chunksize]`: like dynamic, but with decreasing chunk size (minimal size = chunksize); used for load balancing when dynamic induces too much overhead
  - `runtime`: determine by `OMP_SCHEDULE` shell variable

OpenMP parallel regions
How to generate a team of threads

- `!$OMP PARALLEL` **and** `!$OMP END PARALLEL`
  - Encloses a parallel region: All code executed between start and end of this region is executed by all threads.
  - This includes subroutine calls within the region (unless explicitly sequentialized)
  - Both directives must appear in the same routine.

- **C/C++:**

```
#pragma omp parallel
structured block
```

No `END PARALLEL` **directive since block structure defines boundaries of parallel region**

OpenMP work sharing for loops

Requires thread distribution directive

`!$OMP DO / !$OMP END DO` **encloses a loop which is to be divided up if within a parallel region (“sliced”).**

- all threads synchronize at the end of the loop body
- this default behaviour can be changed ...

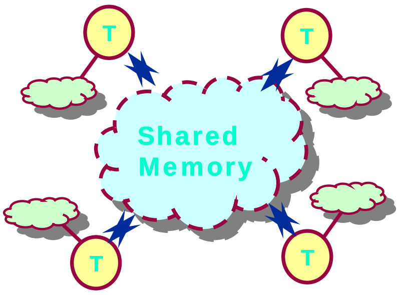
- **Only loop immediately following the directive is sliced**
- **C/C++:**

```
#pragma omp for [clause]
for ( ... )  {
          ...
     }
```

- **restrictions on parallel loops (especially in C/C++)**
  - trip count must be computable (no `do while`)
  - loop body with single entry and single exit point
  - Use integers, not iterators als loop variables

Directives for data scoping: shared and private

- **Remember the OpenMP memory model?**
  Within a parallel region, data can either be
- **private to each executing thread**
  → each thread has its own local copy of data
  or be
- **shared between threads**
  → there is only one instance of data available to all threads
  → this does not mean that the instance is always visible to all threads!



- **Integration example:**
  - shared scope not desirable for x and sum since values computed on one thread must not be interfered with by another thread.
  - Hence:

```
!$OMP parallel private(x,sum)
```

Defaults for data scoping

- **All data in parallel region is shared**
- **This includes global data (Module, COMMON)**
- **Exceptions:**
  1. Local data within enclosed subroutine calls are private (Note: Inlining must be treated correctly by compiler!) unless declared with `SAVE` attribute (`static` in C)
  2. Loop variables of parallel ("sliced") loops are private

- **Due to stack size limits it may be necessary to make large arrays static**
  - This presupposes it is safe to do so!
  - If not: make data dynamically allocated
  - For Intel Compilers: `KMP_STACKSIZE` may be set at run time (increase thread-specific stack size)

Changing the scoping defaults

- **Default value for data scoping can be changed by using the `default` clause on a parallel region:**

```
!$OMP parallel default(private)
```

Not in C/C++

- **Beware side effects of data scoping:**

  Incorrect `shared` attribute may lead to race conditions and/or performance issues ("false sharing").
  - Use verification tools.
- **Scoping of local subroutine data and global data**
  - is not (hereby) changed
  - compiler cannot be assumed to have knowledge
- **Recommendation: Use**

```
!$OMP parallel default(none)
```

to not overlook anything

Compiling and running an OpenMP program

- **Compiler must be instructed to recognize OpenMP directives (Intel compiler: `-openmp`)**
- **Number of threads: Determined by shell variable `OMP_NUM_THREADS`**
- **Loop scheduling: Determined by shell variable `OMP_SCHEDULE`**
- **Some implementation-specific environment variables exist (here for Intel):**
  - `KMP_STACKSIZE`: configure thread-local stack size
  - `KMP_LIBRARY`: specify the strategy for releasing threads that have nothing to do
- **… and then: just type `./a.out`**

Some Details About OpenMP

OpenMP Runtime Library

- `omp_get_num_threads` **Function**
  Returns the number of threads currently in the team executing the parallel region from which it is called
  - Fortran:
    `integer function omp_get_num_threads()`
  - C/C++:
    `int omp_get_num_threads(void);`
- `omp_get_thread_num` **Function**
  Returns the thread number, within the team, that lies between `0` **and** `omp_get_num_threads()-1`**, inclusive. The master thread of the team is thread** `0`
  - Fortran:
    `integer function omp_get_thread_num()`
  - C/C++:
    `int omp_get_thread_num(void);`

OpenMP Example: Hello World Program

```
program hello
!$     integer OMP_GET_THREAD_NUM
       i = -1
!$OMP PARALLEL PRIVATE(i)
!$     i = OMP_GET_THREAD_NUM()
       print *, 'hello world',i
!$OMP END PARALLEL
       stop
       end
```

Work Sharing and Synchronization

- **Which thread executes which statement or operation?**
  - … and in which sequence?
- **i.e., how is parallel work organized/scheduled?**
  - Work-sharing constructs
  - Master and synchronization constructs

OpenMP Work Sharing Constructs

- **Distribute the execution of the enclosed code region among the members of the team**
  - Must be enclosed dynamically within a parallel region
  - Threads do not (usually) launch new threads
  - No implied barrier on entry
- **Directives**
  - `section(s)` directives
  - `do` directive (Fortran)
  - `for` directive (C/C++)

OpenMP `sections` Directives (1)

- **Several *blocks* are executed in parallel**
- **Fortran:**

```
 !$OMP SECTIONS  [ clause [ [ , ] clause ] ... ]
[!$OMP SECTION ]
   block1
[!$OMP SECTION ]
   block2
 ...
 !$OMP END SECTIONS  [ nowait ]
```

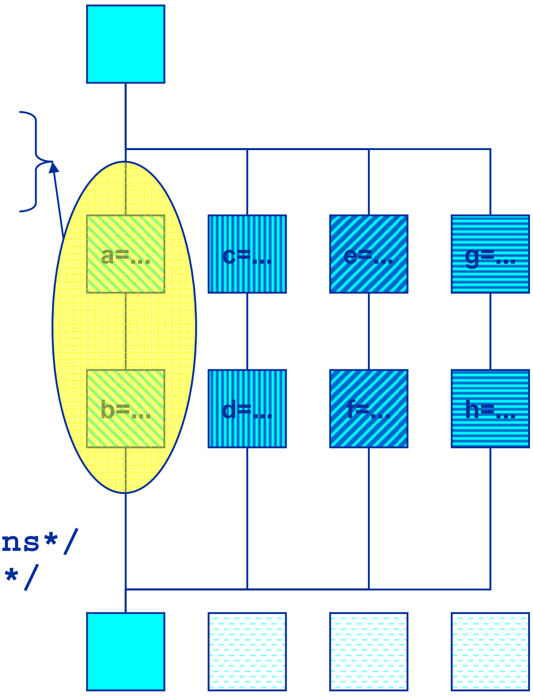
- **C/C++:**

```
#pragma omp sections  [ clause [ clause ] ... ] new-line
  {
    [#pragma omp section new-line ]
       structured-block1
    [#pragma omp section new-line ]
       structured-block2
     ...
  }
```

OpenMP `sections` Directives (2)

C / C++:

```
#pragma omp parallel
{
#pragma omp sections
  {{  a=...;
    b=...; }
#pragma omp section
     { c=...;
    d=...; }
#pragma omp section
     { e=...;
    f=...; }
#pragma omp section
     { g=...;
    h=...; }
  } /*omp end sections*/
} /*omp end parallel*/
```



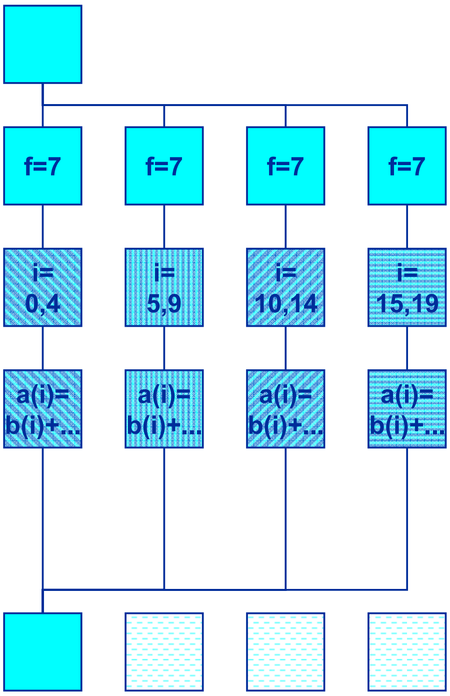
OpenMP `do`/`for` Directives (1)

- Immediately following loop is executed in parallel
- Fortran:
  `!$OMP do` [*clause* [[,] *clause*] ...]
     do_loop
  [`!$OMP end do` [`nowait`]]
- If used, the `end do` directive must appear immediately after the end of the loop
- C/C++:
  `#pragma omp for` [*clause* [*clause*] ...] *new-line*
     for-loop
- The corresponding `for` loop must have "canonical shape":
  `for (i=start; i<=end; i++) { … }`

OpenMP do/for Directives (2)

C / C++:

```
#pragma omp parallel private(f)
{
    f=7;

#pragma omp for
    for (i=0; i<20; i++)
      a[i] = b[i] + f * (i+1);



} /* omp end parallel */
```



OpenMP do/for Directives (3)

- ***clause* can be one of the following:**
  - `private(`*list*`)` **[see later: Data Model]**
  - `reduction(`*operator*`:`*list*`)` **[see later: Data Model]**
  - `schedule(` *type* [, *chunk*] `)`
  - `nowait` (C/C++: on `#pragma omp for`)
(Fortran: on `$!OMP END DO`)
  - ...
- **Implicit barrier at the end of `do/for` unless `nowait` is specified**
- **If `nowait` is specified, threads do not synchronize at the end of the parallel loop**
- **`schedule` clause specifies how iterations of the loop are distributed among the threads of the team.**
  - Default is implementation-dependent

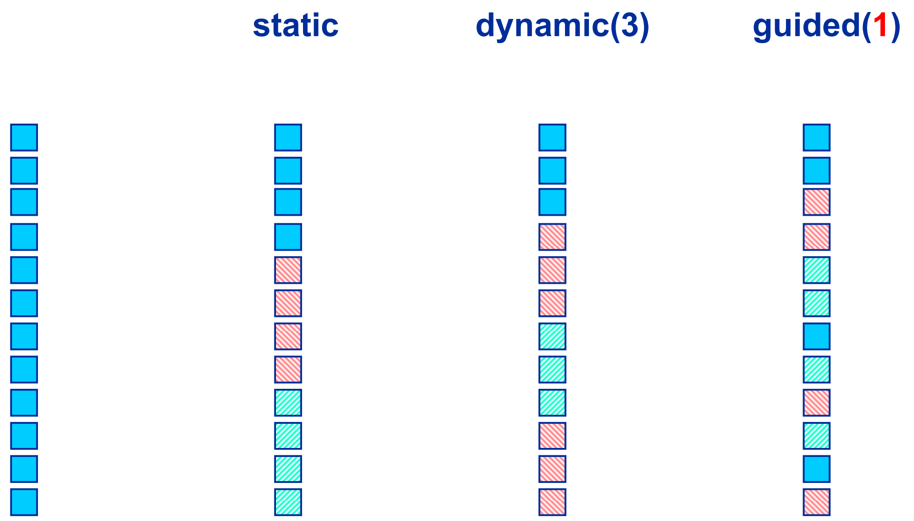
OpenMP `schedule` Clause

Within `schedule(` *type* **[, *chunk* **]** `)` *type* can be one of the following:**

- **`static`: Iterations are divided into pieces of a size specified by *chunk*. The pieces are statically assigned to threads in the team in a round-robin fashion in the order of the thread number.**
  Default chunk size: one contiguous piece for each thread.
- **`dynamic`: Iterations are broken into pieces of a size specified by *chunk*. As each thread finishes a piece of the iteration space, it dynamically obtains the next set of iterations. *Default chunk size: 1.***
- **`guided`: The chunk size is reduced in an exponentially decreasing manner with each dispatched piece of the iteration space.**
  ***chunk* specifies the smallest piece (except possibly the last).**
  ***Default chunk size: 1.* Initial chunk size is implementation dependent.**
- **`runtime`: The decision regarding scheduling is deferred until run time. The schedule type and chunk size can be chosen at run time by setting the `OMP_SCHEDULE` environment variable.**

Default `schedule`: implementation dependent.

Loop scheduling

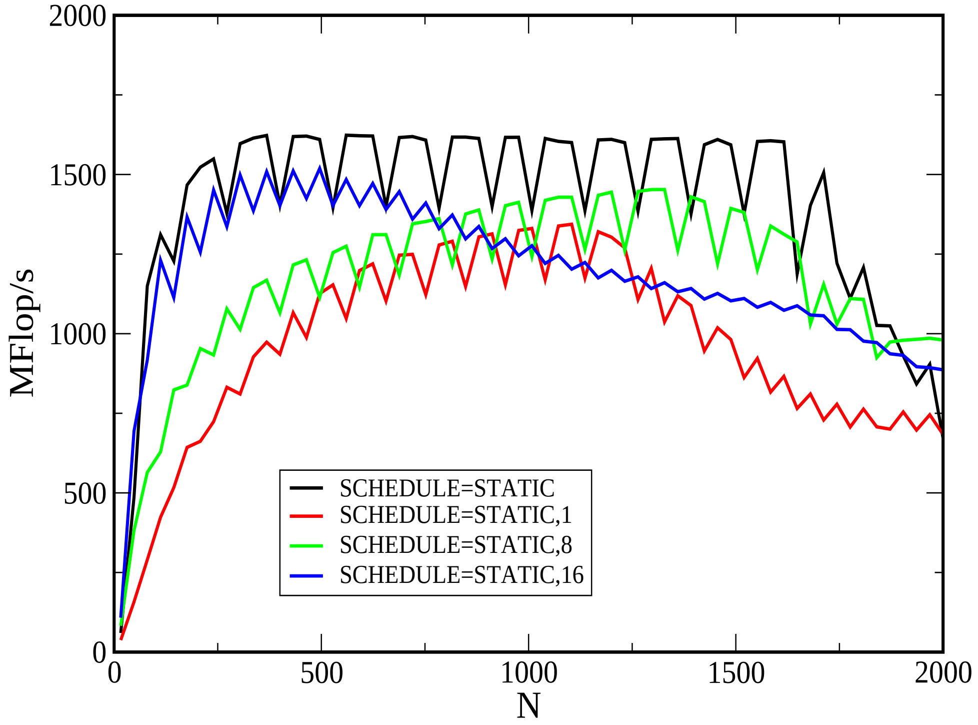


static dynamic(3) guided(1)

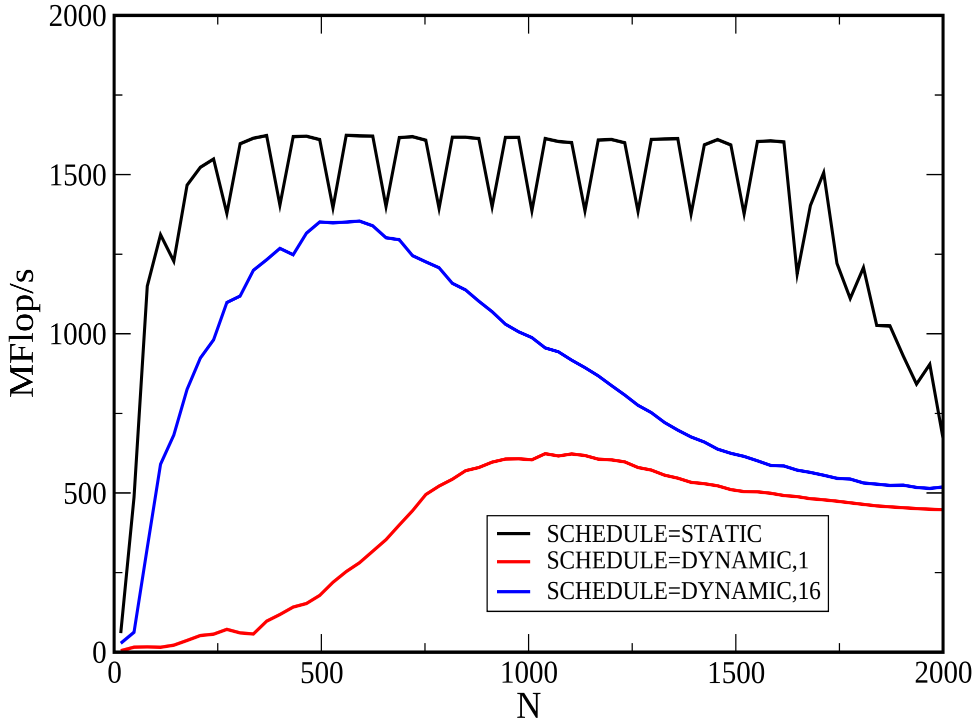
Dense matrix vector multiplication

```
start_time = ...
!$OMP PARALLEL PRIVATE(N,J,I)
do n = 1 , loops
!$OMP DO SCHEDULE(RUNTIME)
   do i=1,N
     do j=1,N
       y(i)=y(i)+a(j,i)*x(j)
     end do
   end do
!$OMP END DO
   call obscure(…) ! Do not interchange n & (i,j) loops
enddo
!$OMP END PARALLEL
end_time = ...
```

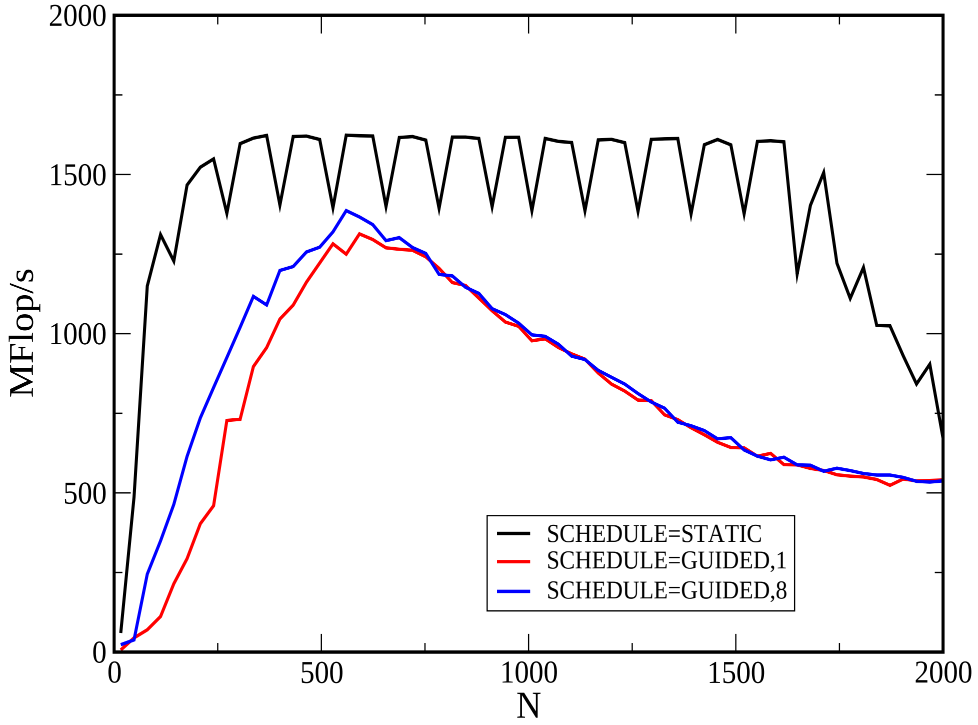
De
SG



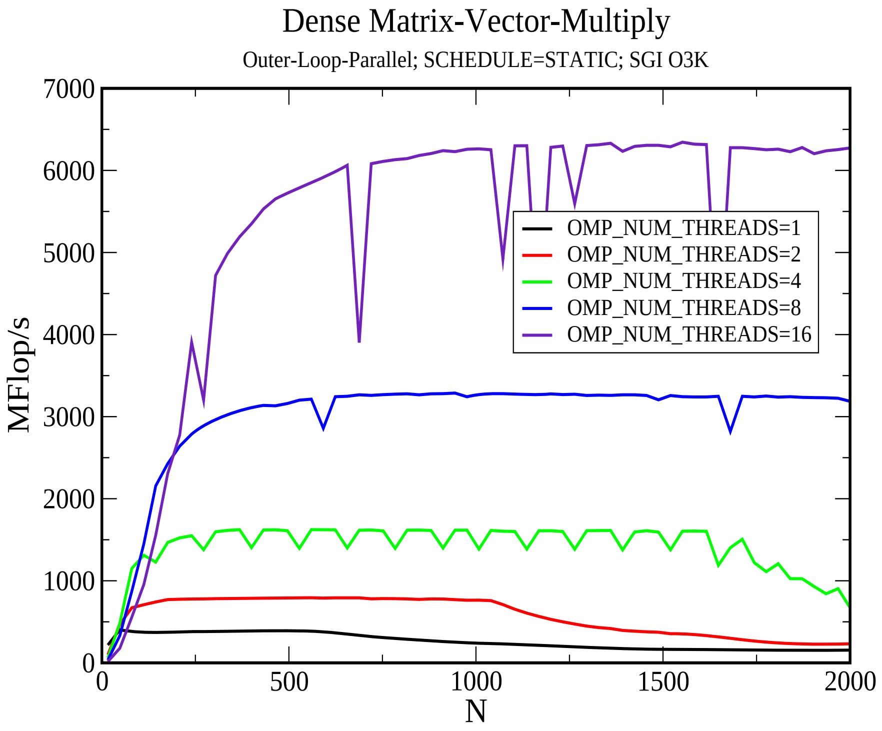
De
SG


2000
1500
1000
500
0
MFlop/s
0
500
1000
1500
2000
N
SCHEDULE=STATIC
SCHEDULE=DYNAMIC,1
SCHEDULE=DYNAMIC,16

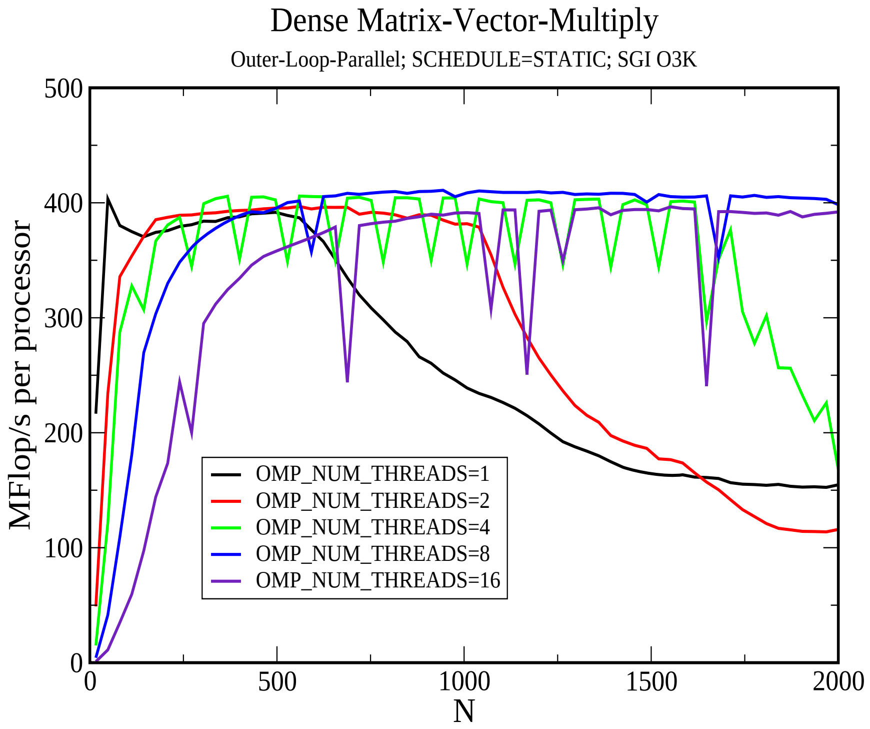
De
SG


2000
1500
1000
500
0
MFlop/s
0
500
1000
1500
2000
N
SCHEDULE=STATIC
SCHEDULE=GUIDED,1
SCHEDULE=GUIDED,8

Dense matrix vector multiplication
SGI Origin: OMP_SCHEDULE=STATIC



Dense matrix vector multiplication
SGI Origin: OMP_SCHEDULE=STATIC



Conditional parallelism: if clause

- Allows execution of a code region in serial or parallel, depending on a condition

- Fortran:

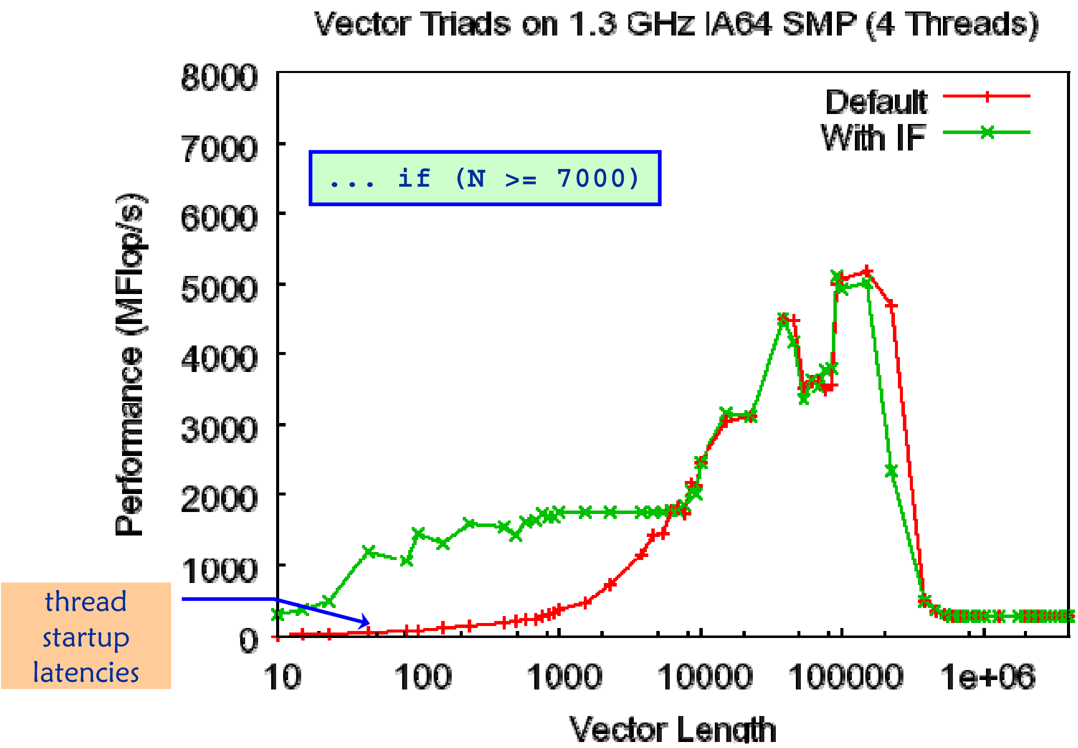
```
!$omp parallel if (condition)
 ... (block)
!$omp end parallel
```

- C/C++:

```
#pragma omp parallel if(condition)
      structured-block
```

- Usage:
  - disable parallelism dynamically
  - define crossover points for optimal performance
    - may require manual or semi-automatic tuning

Example for crossover points: Vector triad with 4 threads on IA64



```
... if (N >= 7000)
```

thread startup latencies

OpenMP reduction Clause

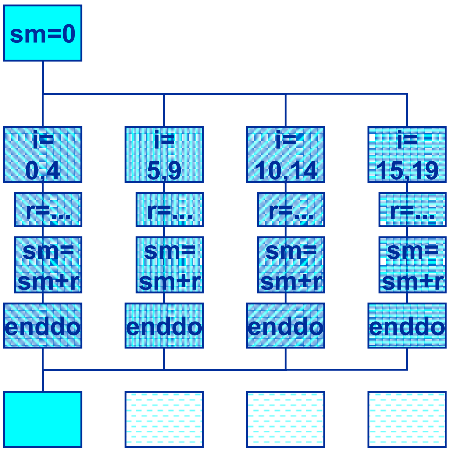
- `reduction` (*operator*:*list*)
- Performs a reduction on the variables that appear in *list*, with the operator *operator*
- *operator*: one of
  - Fortran:
    `+, *, -, .and., .or., .eqv., .neqv.` or
    `max, min, iand, ior,` or `ieor`
  - C/C++:
    `+, *, -, &, ^, |, &&,` or `||`
- Variables must be `shared` in the enclosing context
- At the end of the `reduction`, the shared variable is updated to reflect the result of combining the original value of the shared reduction variable with the final value of each of the private copies using the operator specified

□

OpenMP reduction — an example (C/C++)

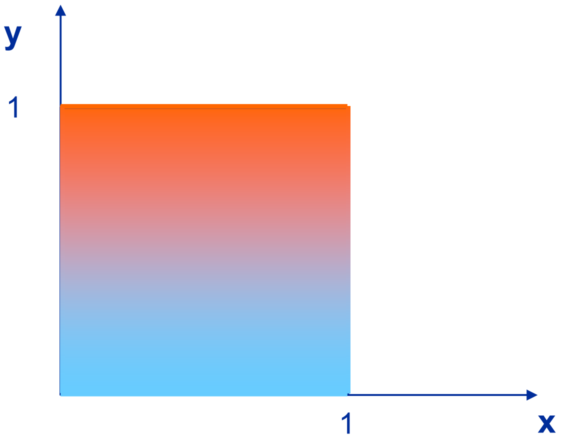
C / C++:

```
   sm = 0;
#pragma parallel
 {
#pragma omp for private(r)
             reduction(+:sm)
   for( i=0; i<20; i++)
   { r = work(i);
     sm = sm + r ;
   } /*end for*/
 } /*end parallel*/
   printf("sum=%f\n",sm);
```



Example: Solving the heat conduction equation

- **Square piece of metal**
  - Temperature Φ(x,y,t)
  - Boundary values:
    Φ(x,1,t) = 1, Φ(x,0,t) = 0,
    Φ(0,y,t) = y = Φ(1,y,t)
  - Initial values for all x, y < 1 are zero
- **Temporal evolution:**
  - to stationary state
  - partial differential equation

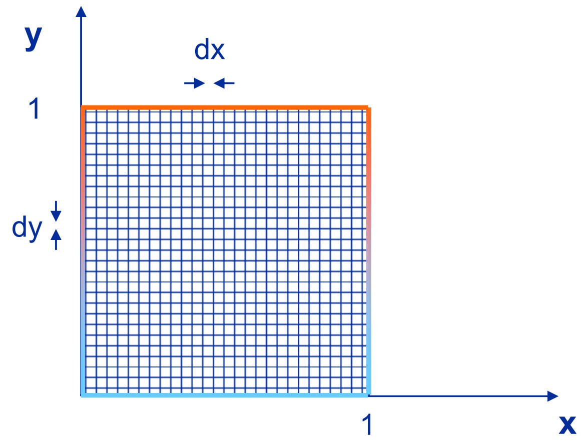


$$\frac{\partial\Phi}{\partial t} = \frac{\partial^2\Phi}{\partial x^2} + \frac{\partial^2\Phi}{\partial y^2}$$

Heat conduction (2): algorithm for solution

- **Interested in stationary state**
  - discretization in space: x_i, y_i
    → 2-D Array Φ
  - discretization in time:
    → steps δt

repeatedly calculate increments



$$\delta\Phi(i,k) = \delta t \cdot \left[\frac{\Phi(i+1,k)+\Phi(i-1,k)-2\Phi(i,k)}{dx^2} + \frac{\Phi(i,k+1)+\Phi(i,k-1)-2\Phi(i,k)}{dy^2} \right]$$

until δΦ=0 reached.

Heat Conduction (3): data structures

- 2-dimensional array `phi` for heat values
- equally large `phin`, to which updates are written
- Iterate updates until stationary value is reached
- Both arrays *shared*
- Tile grid area to OpenMP threads

Heat Conduction (3): code for updates

```
! iteration
do it=1,itmax
   dphimax=0.
!$OMP parallel do private(dphi,i) reduction(max:dphimax)
   do k=1,kmax-1
   do i=1,imax-1
      dphi=(phi(i+1,k)+phi(i-1,k)-2.0_8*phi(i,k))*dy2i   &
          +(phi(i,k+1)+phi(i,k-1)-2.0_8*phi(i,k))*dx2i
      dphi=dphi*dt
      dphimax=max(dphimax,abs(dphi))
      phin(i,k)=phi(i,k)+dphi
   enddo
   enddo
!$OMP end parallel do
```

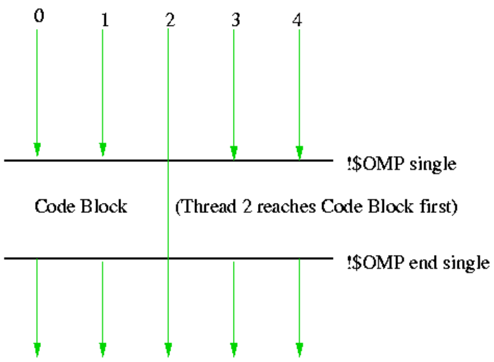
```
!$OMP parallel do
   do k=1,kmax-1
   do i=1,imax-1
      phi(i,k)=phin(i,k)
   enddo
   enddo
!$OMP end parallel do
!required precision reached?
   if(dphimax.lt.eps) goto 10
enddo
10 continue
```

OpenMP Synchronization

- **Implicit Barrier**
  - beginning and end of **parallel** constructs
  - end of all other control constructs
  - implicit synchronization can be removed with **nowait** clause
- **Explicit synchronization**
  - **critical**
  - atomic
  - **single**
  - master
  - barrier
  - flush
  - omp_set_lock() and similar API functions

Synchronization Constructs: single directive

- **The enclosed code is executed by exactly one thread, which one is unspecified**



- **Fortran:**
  `!$OMP SINGLE` ***[clause[[,]clause]…]***
  block
  `!$OMP END SINGLE [NOWAIT]`
-
- **C/C++:**
  `#pragma omp single` ***[clause[[,]clause]…] [nowait] new-line***
  structured-block

Synchronization Constructs: single directive

- The other threads in the team skip the enclosed section of code and continue execution. There is an implied barrier at the exit of the `single` section!

- may not appear within a `parallel do` (deadlock!)
- `nowait` clause after `end single` (or at start of parallel region in C/C++) suppresses synchronization

Synchronization Constructs: barrier directive

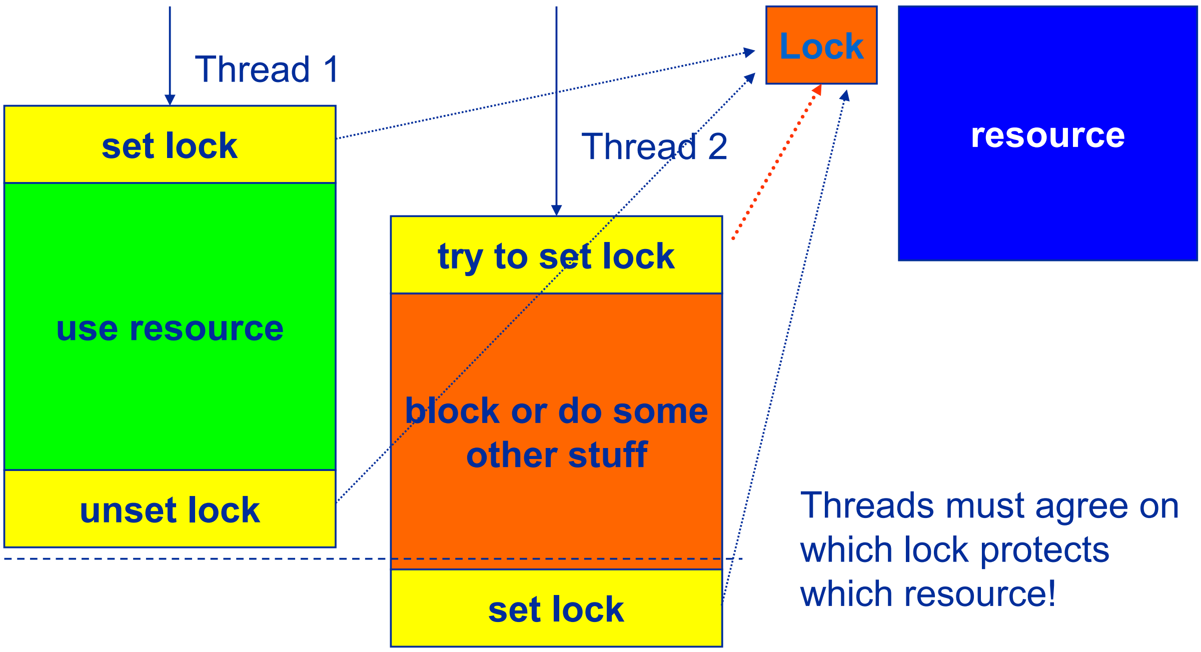
- Synchronizes all threads in the team
- Fortran:
  `!$OMP BARRIER`
- C/C++:
  `#pragma omp barrier` *new-line*
  - In C(++) the directive must appear inside a block or compound statement
- After all threads have encountered the barrier, they continue to execute the code after it in parallel

- Barrier is a collective operation: it must either be encountered by all threads in the team or none at all
  - else: deadlock!

Synchronization Constructs: API Locking Functions

- **OpenMP API provides some functions that allow explicit locking (POSIX: „mutex“)**
- **Explicit locking has user-defined semantics**
  - The compiler knows nothing about the binding of a lock to a resource
- **Simple variables can be protected by directives (`atomic`/`critical`), but how about more complicated constructs?**
  - User-defined data structures
  - Thread-unsafe library routines
  - Arrays of objects
  - …
- **API functions allow more flexible strategies when a resource is locked**
  - Lock may be tested without blocking

API Locking Functions



Threads must agree on which lock protects which resource!

API Locking Functions:
Lock Definitions

- **A lock must be defined and initialized before it can be used**
- **Fortran:**

```
INTEGER (KIND=OMP_LOCK_KIND) :: lockvar
CALL OMP_INIT_LOCK(lockvar)
```

- **C/C++:**

```
#include <omp.h>
omp_lock_t lockvar;
omp_init_lock(&lockvar);
```

- **Initialization is required to use the lock afterwards**
- **Lock can be removed (uninitialized) if not needed any more**
  - `OMP_DESTROY_LOCK` subroutine, `omp_destroy_lock()` function

API Locking Functions:
Setting and Unsetting Locks

- **Setting and unsetting a lock is an atomic operation**
- **Fortran:**

```
CALL MP_SET_LOCK(lockvar)
CALL MP_UNSET_LOCK(lockvar)
```

- **C/C++:**

```
omp_set_lock(&lockvar);
omp_unset_lock(&lockvar);
```

- ***lockvar* must be an initialized lock variable**
- **Setting the lock implies blocking if the lock is not available (i.e. set by another thread)**
  - threads waits until lock becomes available

API Locking Functions: Testing Locks

- Test a lock and set it if it is unlocked (non-blocking)
- Fortran:

```
LOGICAL locked
locked = OMP_TEST_LOCK(lockvar)
```

- C/C++:

```
int locked;
locked = omp_test_lock(&lockvar);
```

- If the lock is already locked, returns with `.FALSE.` or zero, else sets it and returns `.TRUE.` or nonzero
- Only way to overlap work and resource sharing

API Locking Functions: Example

```
    program uselock
    integer omp_get_thread_num
    logical omp_test_lock
    external omp_get_thread_num , omp_test_lock
    integer LCK,id
    call OMP_INIT_LOCK(LCK)
!$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
    id=OMP_GET_THREAD_NUM()
    do while(.not. OMP_TEST_LOCK(LCK))
       call dosomework(id)  ←——— Work while waiting for lock
    end do
    print*,'thread id=', id , 'calls work'  }
    call work(id)                           }  protected by LCK
    call OMP_UNSET_LOCK(LCK)
!$OMP END PARALLEL
    call OMP_DESTROY_LOCK
    end
```

OpenMP library routines

- **Querying routines**
  - how many threads are there?
  - who am I?
  - where am I?
  - what resources are available?
- **Controlling parallel execution**
  - set number of threads
  - set execution mode
  - implement own synchronization constructs

OpenMP library routines (1)

Function calls return type INTEGER unless specified

`OMP_GET_NUM_THREADS()`

yields number of threads in present environment
always 1 within sequentially executed region

in serial part only!

`call OMP_SET_NUM_THREADS(nthreads)` **(Subroutine call)**

set number of threads to a definite value

`0 ≤ nthreads < omp_get_max_threads()`

- useful for specific algorithms
- dynamic thread number assignment must be deactivated
- overrides setting of `OMP_NUM_THREADS`

`OMP_GET_THREAD_NUM()`

yields index of executing thread (`0, ..., nthreads-1`)

`OMP_GET_NUM_PROCS()`

yields number of processors available for multithreading
→ Always 8 for SR8000, # of processors for SGI (28 at RRZE)

OpenMP library routines (2)

`OMP_GET_MAX_THREADS()`

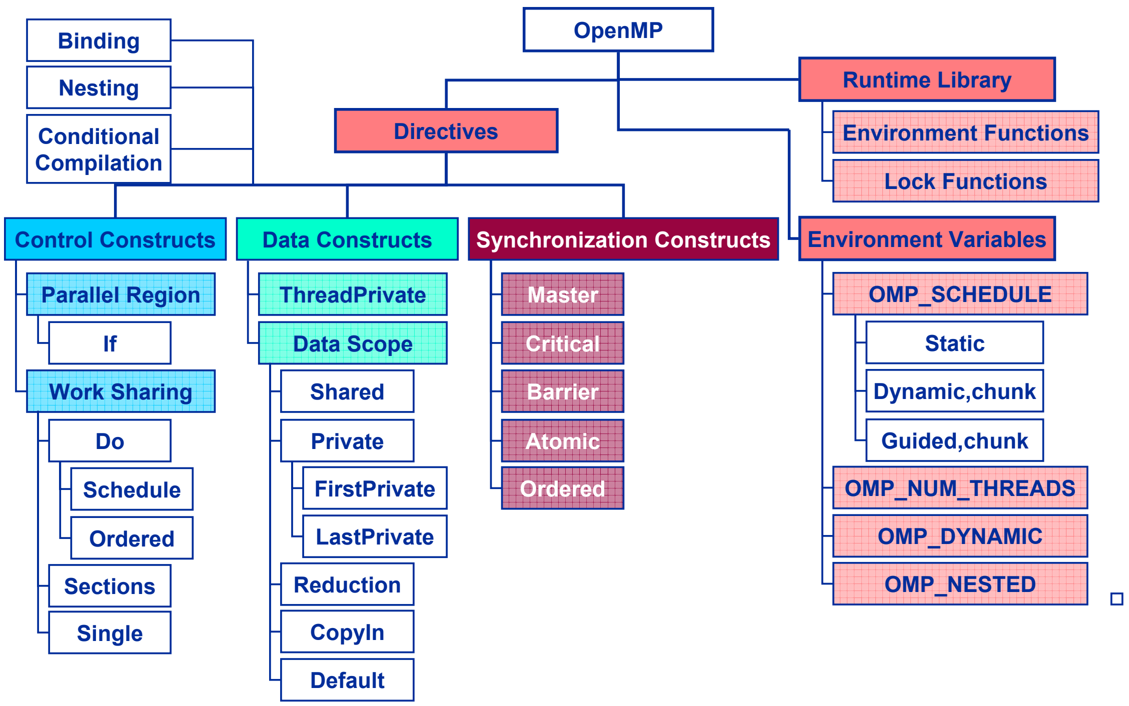
maximum number of threads potentially available
(e.g., as set by operating environment/batch system)

`OMP_IN_PARALLEL()` (logical)

query whether program is executed in parallel or sequentially

In the example program, thread ID is used to distribute work

OpenMP Constructs reviewed



- OpenMP
  - Directives
    - Binding
    - Nesting
    - Conditional Compilation
    - Control Constructs
      - Parallel Region
        - If
      - Work Sharing
        - Do
          - Schedule
          - Ordered
        - Sections
        - Single
    - Data Constructs
      - ThreadPrivate
      - Data Scope
        - Shared
        - Private
          - FirstPrivate
          - LastPrivate
        - Reduction
        - CopyIn
        - Default
    - Synchronization Constructs
      - Master
      - Critical
      - Barrier
      - Atomic
      - Ordered
  - Runtime Library
    - Environment Functions
    - Lock Functions
  - Environment Variables
    - OMP_SCHEDULE
      - Static
      - Dynamic,chunk
      - Guided,chunk
    - OMP_NUM_THREADS
    - OMP_DYNAMIC
    - OMP_NESTED

OpenMP Pitfalls: Correctness

OpenMP Pitfalls: Three Types of Shared-Memory Errors

- **Race Condition**
  - Def.: *Two threads access the same shared variable* ***and*** *at least one thread modifies the variable* ***and*** *the sequence of the accesses is undefined, i.e. unsynchronized*
  - The result of a program depends on the detailed timing of the threads in the team.
  - This is often caused by unintended sharing of data
- **Deadlock**
  - Threads lock up waiting on a locked resource that will never become free.
    - Avoid lock functions if possible
    - At least avoid nesting different locks
- **Livelock**
  - multiple threads work forever on individual tasks

Example for race condition (1)

```
!$omp parallel sections
       A = B + C
!$omp section
       B = A + C
!$omp section
       C = B + A
!$omp end parallel sections
```

```
ic = 0
!$omp parallel sections
!$omp section
  a = b + c
  ic = 1
!$omp section
  do while (ic < 1)
!$omp flush(ic)
  end do
  b = a + c
  ic = 2
  ... (etc)
!$omp end parallel sections
```

might effectively serialize code!

- The result varies unpredictably based on specific order of execution for each section.
- Wrong answers produced without warning!
- Solution: Apply synchronization constructs

Example for race condition (2)

```
!$OMP PARALLEL SHARED (X), PRIVATE(TMP)
       ID = OMP_GET_THREAD_NUM()
!$OMP DO REDUCTION(+:X)
       DO 100 I=1,100
              TMP = WORK1(I)
              X = X + TMP
100  CONTINUE
!$OMP END DO NOWAIT
       Y(ID) = WORK2(X,ID)
!$OMP END PARALLEL
```

- The result varies unpredictably because the value of X isn’t dependable until the barrier at the end of the do loop.
- Solution: Be careful when using `NOWAIT`.