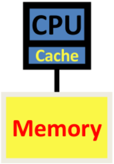
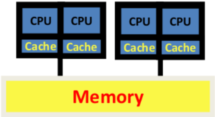
OpenMP

How to build faster computers?

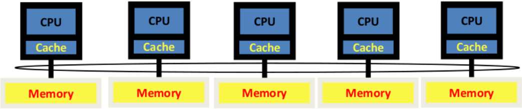
1. **Increase performance / throughput of CPU core**
   a) Reduce cycle time, i.e. increase clock speed (Moore)
   b) Increase throughput, i.e. superscalar + SIMD



2. **Improve data access time**
   a) Increase cache size
   b) Improve main memory access (bandwidth & latency)
3. **Use parallel computing (shared memory)**
   a) Requires shared-memory parallel programming
   b) Shared/separate caches
   c) Possible memory access bottlenecks

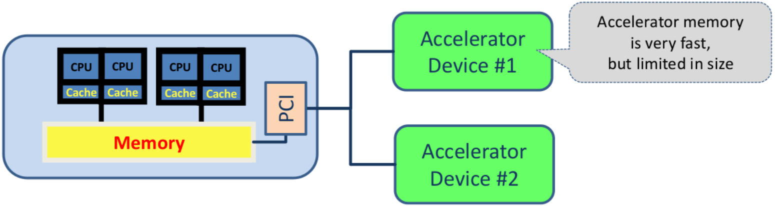


4. **Use parallel computing (distributed memory)**
   "Cluster" of computers tightly connected
   a) Almost unlimited scaling of memory and performance
   b) Distributed-memory parallel programming



How to build faster computers?

5. **Use an accelerator with your compute node**
   a) Requires offload of program regions
   (semantics may be limited)
   b) Host and accelerator memory are connected, but separate



   (Improvements are under way)
   c) Programming complexity is higher than for shared memory systems
   („heterogeneous parallel computing“)

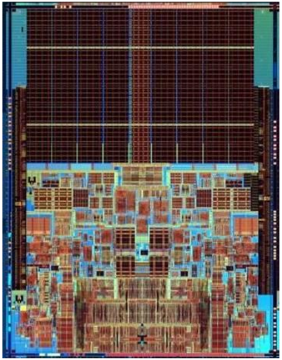
How to build faster computers?

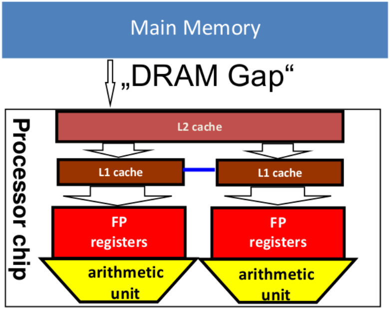
It is not a faster CPU – it is a parallel computer on a chip.

Put multiple processors (“cores”) on a chip which share resources
(example shows a dual core that shares L2 cache and memory bandwidth)

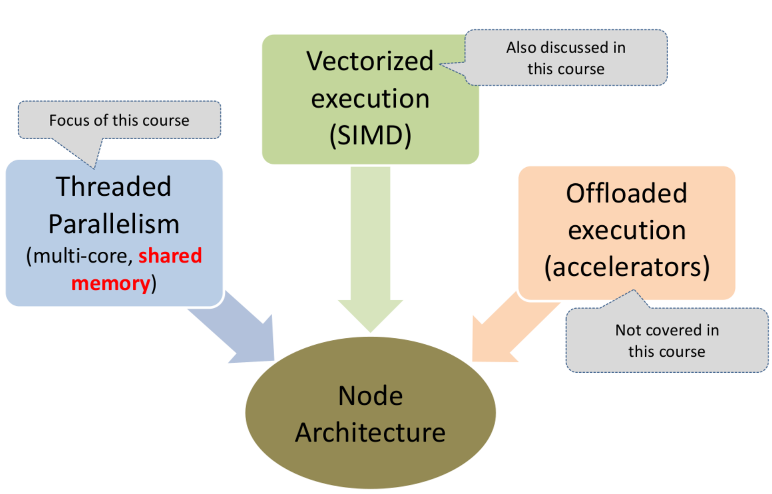
Efficient use of all cores for a single application → programmer

Intel Xeon (Woodcrest)





How to build faster computers?



OpenMP and Portability

Syntactic portability

- Directives / pragmas
- Conditional compilation permits to masks API calls

Semantic portability

- Standardized across platforms → safe-to-use interface
- Unsupported/unavailable hardware features → irrelevant directives will be ignored (you might need a special compiler for your devices …)

Performance portability

- Unfortunately performance is not necessarily portable
- Has traditionally been a problem (partly due to differences in hardware/architectural properties)

OpenMP Standard

Responsible body: OpenMP Architecture Review Board

- Published OpenMP **4.5** in November 2015
- Development continues

History of OpenMP starts in 1997

Base languages

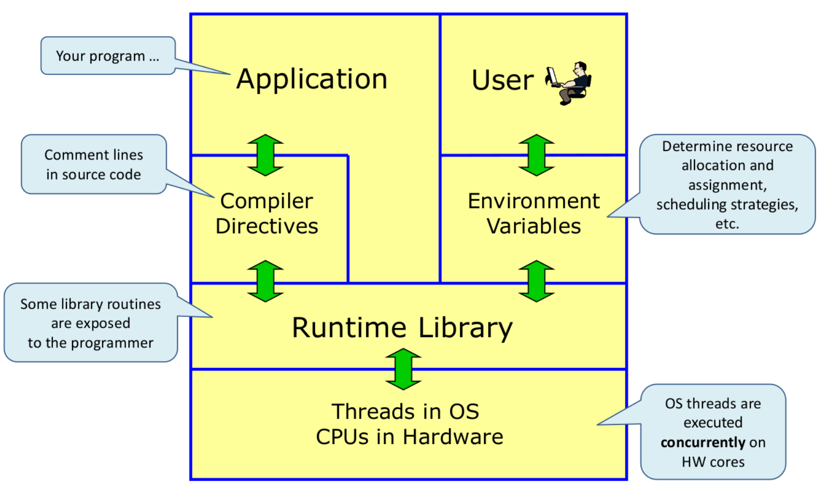
- Fortran (77, 95, 2003)
- C, C++
- (Java is not a base language)

Fortran and C examples will be displayed

Resources:

- http://www.openmp.org (including standard documents)
- http://www.compunity.org

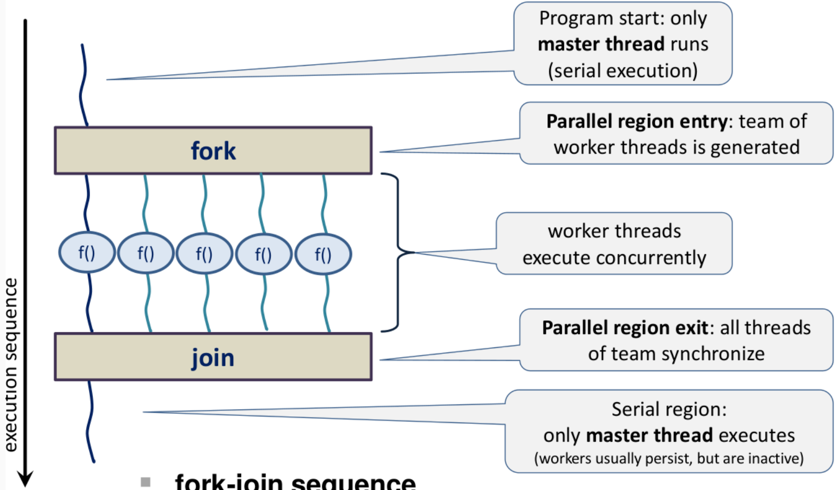
OpenMP Architecture



OpenMP Example

```
#include <stdio.h>
int main() {


  f();

  return 0;
}




void f() {
  printf("Hello\n");
}
```

OpenMP: Parallel Execution Model



- **fork-join sequence**
  - can repeat, with differing thread counts

OpenMP: Parallel Execution Model

C

```c
#include <stdio.h>
int main() {
#pragma omp parallel
  {
    f();
  }
  return 0;
}
```

```
#pragma omp <directive> [<clause>]
```

sentinel

- clauses, if present, modify a directives semantics
- multiple clauses per directive are possible
- continuation lines are supported for long directives: &, \

OpenMP: Library Calls

C

```c
#include <stdio.h>
#include <omp.h>
void f() {
  int me = 0;
#ifdef _OPENMP
  me = omp_get_thread_num();
#endif
  printf("Hello from thread %i\n",me);
}
```

OpenMP include file: prototypes for API

OpenMP-specific macro for conditional compilation

OpenMP: Compilation

C

```
cc -fopenmp -o hello.exe hello.c
```

```
export OMP_NUM_THREADS=4
./hello.exe
```

by default, parallel regions generate a team with 4 threads

```
Hello from 1
Hello from 3
Hello from 0
Hello from 2
```

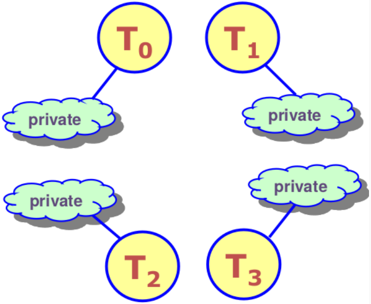
ordering will vary between runs (asynchronous execution)

OpenMP: Independent Execution Contexts

As many independent function calls as there are threads

Thread-individual memory management within function call

- local variables ("me") are created in the thread-specific stack
- malloc() or ALLOCATE create memory in the heap separately for each thread



Private variables

- associated with a particular thread are **inaccessible** by any other thread
- **pro:** **safe** to use
- **con:** **communication** is not possible (it is needed by many parallel algorithms), unnecessary replication of objects may happen.

Thread-individual stack limit

- control via environment variable (example: 100 MByte)

```
export OMP_STACKSIZE=100M
```

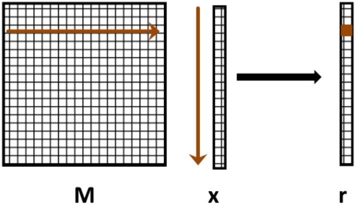
OpenMP: Matrix times vector

We know how to set up threading, but

- how can a large work item be divided up among threads?
  (using the API for this works in principle, but is tedious)
- what happens with objects that already exist before the parallel region starts?

Example:

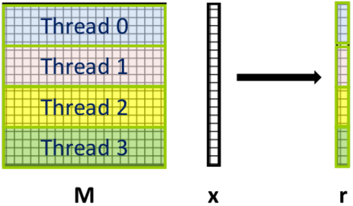
- matrix-vector multiplication r = M · x i.e. $r_i = \sum_{j=1}^{n} M_{ij} x_j$



A bunch of scalar products

OpenMP: Work Sharing

The idea is to split the work among threads



Note that

- all elements of **x** must be available to **all** threads
- Matrix-Vector is often deployed iteratively → **r** becomes **x** in the next iteration
  → copying of data must be possible

Consequence:

- need for variables that are accessible to **all** threads
  → "data sharing" is often a prerequisite for "work sharing"
  → a natural concept for a shared memory programming model

OpenMP: Work Sharing: Matrix times vector

C

```
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
     r[j] = r[j] + a[k*n+j] * x[k];
  }
}
```

```
#pragma omp parallel
{
  #pragma omp for
  for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
      r[j] = r[j] + a[k*n+j] * x[k];
    }

  }
  … = r[…];
}
```

applies to j-loop

no race condition against previous definitions

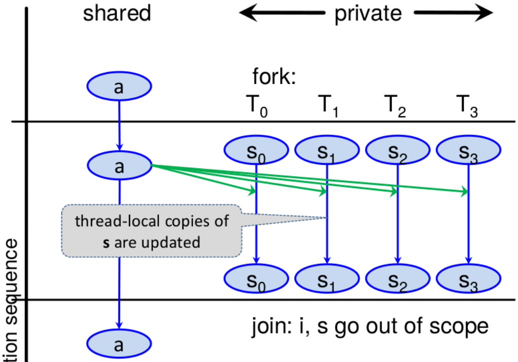
OpenMP: Private Variables

C

```c
   a[k] = …;
#pragma omp parallel \
        shared(a)
  { int i; float s;
    s = 0.0;
    for (i=…;i<…;i++) {
      s += a[i];
    }
  }
```

example calculates **thread-individual** sums

useless, from a practical point of view. But bear with me - we'll fix this, eventually



- **Block-local variables in C/C++**
  - are automatically private

Note: I would expect the same behaviour for the Fortran 2008 BLOCK construct, but this is currently not specified in the OpenMP standard

OpenMP: Atomic

C

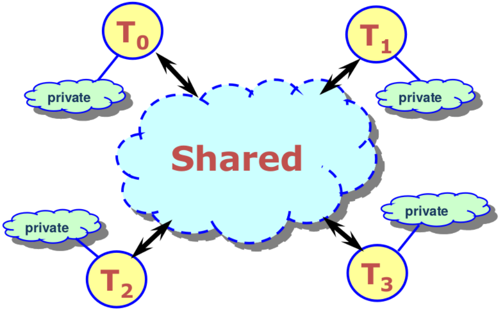
```c
  float stot;
  stot = 0.0;
#pragma omp parallel \
        shared(a,stot)
  { int i; float s;
    s = 0.0;
#pragma omp for
    for (i=0;i<N;i++) {
      s += a[i];
    }
#pragma omp atomic update
    stot += s;
  }
```

parallel array summation

legacy notation
`omp atomic`
is also permitted

- **Properties of atomic operations**
  - the **atomic** directive applies only for a **single update** to a **scalar** shared variable of intrinsic type
  - this way of updating can be made safe when executed concurrently (explicit use of race condition!)
  - otherwise, no synchronising effect imposed by semantics
  - if hardware atomic instructions are available, likely to be more efficient than a critical region

OpenMP: Two Kinds of Memory



- **Data accessed by can be shared or private**
  - shared data – one instance of an entity available to all threads (in principle)
  - private data – each per-thread copy only available to thread that owns it
- **Data transfer** transparent to programmer
- **Synchronization** necessary for accessing shared data from different threads to avoid race conditions
  - implicit barrier
  - explicit directive

OpenMP: Scoping

C

```
#pragma omp parallel default(none) \
        shared(…) private(…) …
  …
```

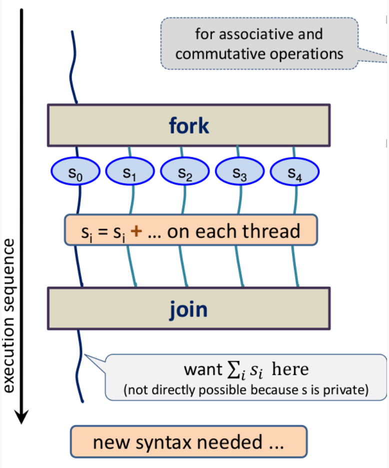
- this **forces** you to explicitly consider and specify scoping for all pre-existing objects

pre-existing objects are by default **shared**, except for loop variables, which are **private**.

objects declared inside the lexical or dynamic scope of the construct are **private**.

this cannot be changed, of course

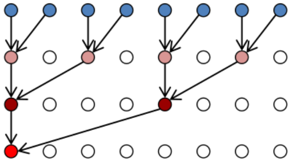
OpenMP: Reduction



OpenMP: Reduction

OpenMP reductions:

- sometimes more efficient - implementation tunings like



  reduce complexity from $O(n_{threads})$ to $O(\log_2(n_{threads}))$
- always easier to understand and maintain

OpenMP: Reduction

C

```c
  float s;
  s = 2.2;
#pragma omp parallel reduction(+:s)
  {
    …
    s += …;
  }
  … *= s;
```

operation consistent

- value of s after end of parallel region: $s_{\text{incoming}} + \sum_i s_i$

OpenMP: Initial Values of Reduction Variables

C / C++

<table>
<tr><th>Operation</th><th>Initial value</th></tr>
<tr><td>+</td><td>0</td></tr>
<tr><td>-</td><td>0</td></tr>
<tr><td>*</td><td>1</td></tr>
<tr><td>&</td><td>0</td></tr>
<tr><td>|</td><td>0</td></tr>
<tr><td>^</td><td>0</td></tr>
<tr><td>&&</td><td>1</td></tr>
<tr><td>||</td><td>0</td></tr>
<tr><td>MAX</td><td>smallest representable value</td></tr>
<tr><td>MIN</td><td>largest representable value</td></tr>
</table>

OpenMP: Array Reductions

C/C++

```
float *a;
float b[N];

#pragma omp parallel \
    reduction(+:b[:]) \
    reduction(*:a[0:m])
 …
```

pointee created e.g. via malloc()

same as b[0:N]

OpenMP: Array Reductions

General rules:

- array section must be a **contiguous** object (→ no strides permitted)
- dynamic objects must be associated / allocated, and the status must not be modified for the private copies

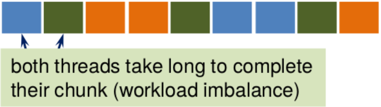
no deallocate/free within reduction region

OpenMP: Scheduling

- **1. Static scheduling**
  - `schedule(static,10)`

  - minimal overhead (precalculate work assignment)
  - default chunk value: see left
- **2. Dynamic scheduling**
  - after a thread has completed a chunk, it is assigned a new one, until no chunks are left

    `schedule(dynamic, 10)`



  - synchronization **overhead**
  - default chunk value is **1**