Multiple Levels of Parallelism

- Distributed Machines with their own local memory connected via network
- Multiple CPUs within a single compute node/
  Cores within a single CPU
  accessing the same local memory
- Data-level parallelism within the Cache of a single CPU Core
- Instruction level parallelism within a CPU Core

Dot Product: sequential

```
std::vector<Real> a_store(0); a_store.reserve(N);
std::vector<Real> b_store(0); b_store.reserve(N);
Real * a = a_store.data();
Real * b = b_store.data();
// initialize
for (int i = 0; i < N; ++i) {
  a[i] = i * Real(1.1);
  b[i] = i * Real(1.2);
}
// compute
Real sum(0);
for (int i = 0; i < N; ++i) sum += a[i] * b[i];
```

Dot Product: vectorized

```
#include <boost/align/aligned_allocator.hpp>

std::size_t constexpr ALIGNMNENT = 64;
template <typename T>
using aligned_allocator = boost::alignment::aligned_allocator<T, ALIGNMNENT>;
template<class T>
using aligned_vector = std::vector<T, aligned_allocator<T>>;

aligned_vector<Real> a_store(0); a_store.reserve(N);
aligned_vector<Real> b_store(0); b_store.reserve(N);
Real * a = a_store.data();
Real * b = b_store.data();

#pragma omp simd aligned (a,b:ALIGNMNENT)
for (int i = 0; i < N; ++i) {
  a[i] = i * Real(1.1);
  b[i] = i * Real(1.2);
}

Real sum(0);
#pragma omp simd aligned (a, b:ALIGNMNENT) reduction(+:sum)
for (int i = 0; i < N; ++i) sum += a[i] * b[i];
```

Dot product: std::thread

```
void dot(int thread_id, int num_threads, Real & thread_sum,
         int N, Real const* a, Real const* b) {
  Real const fraction = N / Real(num_threads);
  int begin = (0 == thread_id) ? 0 : (fraction * thread_id + 0.5);
  int end = (num_threads - 1 == thread_id) ?
            N : (fraction * (thread_id + 1) + 0.5);
  for (int i = begin; i < end; ++i)
    thread_sum += a[i] * b[i];
}

std::vector<Real> sums(NUM_THREADS, 0.0);
std::vector<std::thread> threads(0);
for (int i = 0; i < NUM_THREADS; ++i)
  threads.emplace_back(dot, i, NUM_THREADS, std::ref(sums[i]), N, a, b);
for (int i = 0; i < NUM_THREADS; ++i) threads[i].join();
Real sum = std::accumulate(sums.begin(), sums.end(), 0.0);
```

Dot Product: OpenMP

```
std::vector<Real> a_store(0); a_store.reserve(N);
std::vector<Real> b_store(0); b_store.reserve(N);
Real * a = a_store.data();
Real * b = b_store.data();

#pragma omp parallel for
for (int i = 0; i < N; ++i) {
  a[i] = i * Real(1.1);
  b[i] = i * Real(1.2);
}

Real sum(0);
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < N; ++i)
  sum += a[i] * b[i];
```

OpenMP - History

- roots in supercomputing 1997
- diversity of programming models for shared-memory systems
- proposal of a portable API(!) consisting of *directives*, *runtime-library* and *environment variables*
- for both Fortran and C/C++
- published by ARB (Architecture Review Board)
  = Intel, AMD, ARM, NVIDIA, NASA, ...
- continuously updated: 1.0, 2.5, 3.1, 4.0 (2013)

OpenMP

- not a new language
- it is additional notation to added to a sequential program in Fortran, C or C++
- facilitates incremental parallelization with minimal effort -> start from correct(!) sequential version
- requires compatible compiler
- directives tell the compiler which parts of code to execute in parallel and how to share the work between those threads

OpenMP-Process

- write correct sequential version
- * identify hot-spots (code that consumes a substantial amount of execution time)
- identify parts within a hot-spot that can be potentially parallelized
- express the kind of parallelism and work-sharing with OpenMP directives
- compile, run (check correctness)
- if desired speed-up not achieved, goto *

OpenMP basics

- all pragmas start as #pragma omp dicrective
  e.g.: `#pragma omp parallel num_threads(2)`
- optional clauses, e.g. `num_threads`
- directives apply to the immediately following structured block (start at top - exit at bottom)

```
#pragma omp parallel
{...}
#pragma omp parallel for
for (int i = 0; i < N; ++i) std::cout << i << "\n";
```

- newline required at the end of a pragma

OpenMP library

- provides a set of functions to query or set the OpenMP state
- defined in header omp.h
- most frequently used functions
  omp_get_num_threads()
  total number of threads within the current block
  omp_get_thread_num()
  id of the current thread within the *team* executing the current block

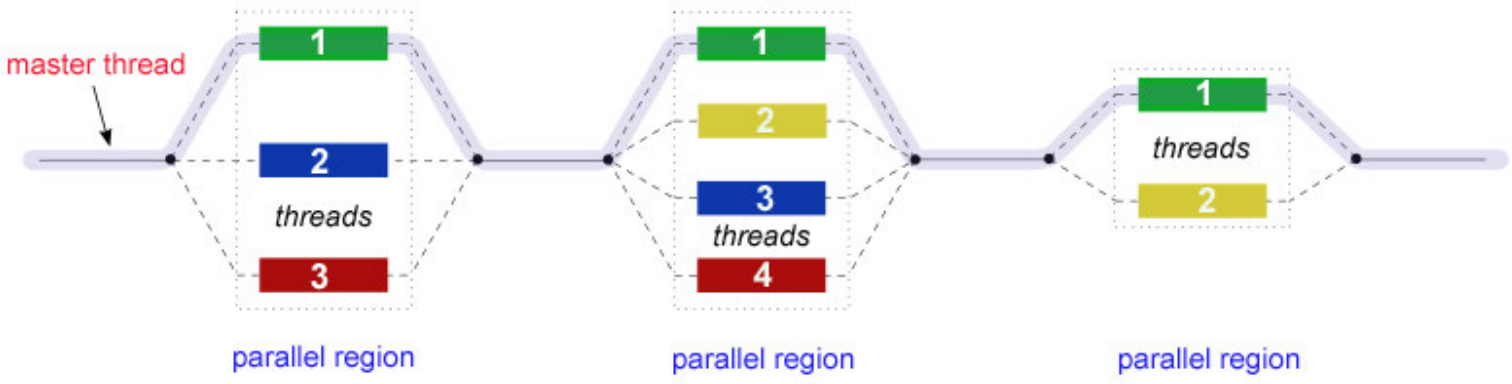
Hello World

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

int main() {

#pragma omp parallel
{
  std::cout << "hello world says thread "
            << omp_get_thread_num()
            << " of "
            << omp_get_num_threads()
            << std::endl;
}

}
```

- compile with -fopenmp when using gcc
- with -qopenmp when using Intel compiler
- by default uses all hardware threads of the OS

```
hello world says thread 0 of 4
hello world says thread 2 of 4
hello world says thread 1 of 4
hello world says thread 3 of 4
```

OpenMP Execution Model



- program starts single threaded: master-thread
- when encountering a parallel region, forks into a team of threads, all executing the contained code
- when leaving the parallel region, all team threads join and terminate, leaving only the master thread
- parallel regions can be nested

OpenMP communication

- OpenMP uses threads as the parallelization primitive
- threads within a program use the same shared address space
- communicate/synchronize by sharing variables
- unintended sharing can lead to race conditions:
  a program's outcome changes with different thread scheduling (unpredictable semantics)
- synchronization primitives to control data conflicts
- malicious synchronization can lead to dead lock

The parallel directive

- `#pragma omp parallel [clause ...]  newline`
- default (**shared** | none)
- shared(*list*)
- private(*list*)
  every thread owns a private copy
- firstprivate(*list*)
  like private, but initialize to value before parallel region
- if(*condition*)
  when false, execute by master-thread only
- num_threads(integer-expression)

How many threads?

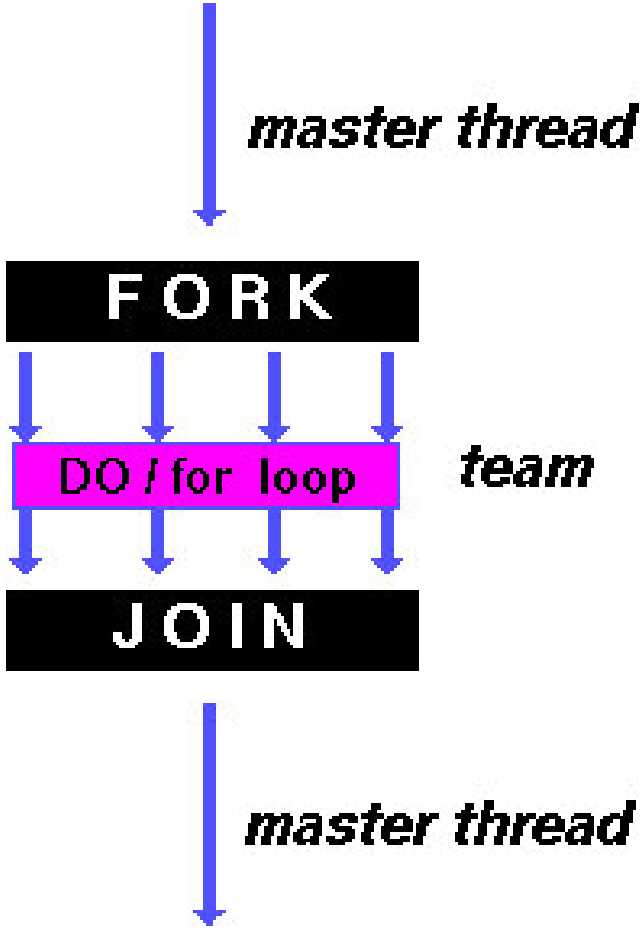
1) evaluate the if()-clause
2) evaluate the num_threads()-clause
3) preceding call of omp_set_num_threads()
4) OMP_NUM_THREADS environment variable
5) default: usually nr. of OS threads

Work-sharing

- so far we create a MISD (multiple instruction single data) program (redundant work)
- to achieve speed-up we need to share work between threads
- work-sharing constructs divide work enclosed within a following structured block among the threads of the enclosed parallel region
- work-sharing constructs do not launch new threads!
- there is an implied barrier at the end (but not at the beginning)

work-sharing: for directive

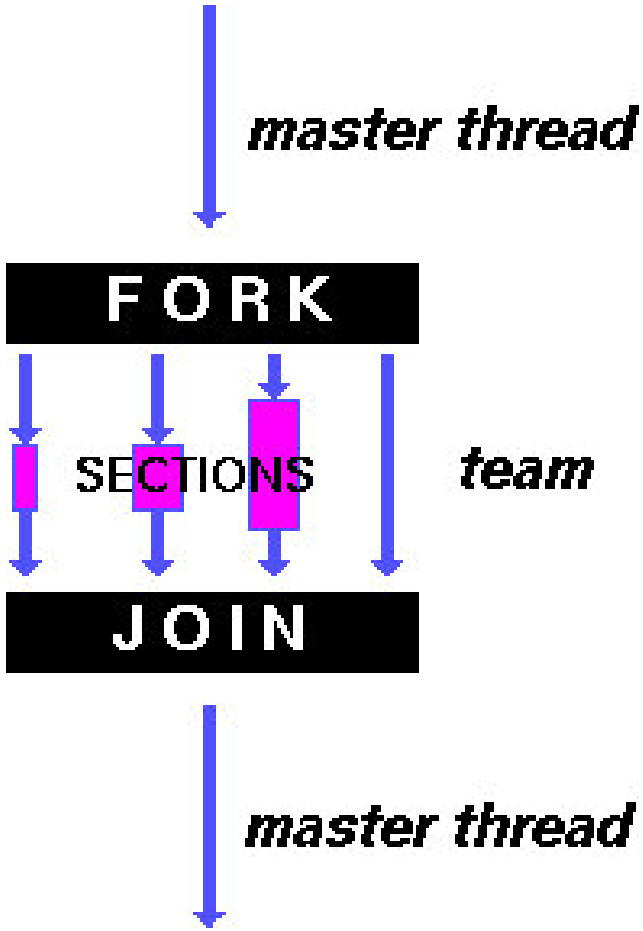
- threads share iterations of a single loop
- #pragma omp for [clause...]
- private, firstprivate, shared
- ordered
  same order as sequential
- schedule(**static** | dynamic [, chunk])
- reduction(operator:variable)
  e.g. reduction(+:sum)



Example: for-directive

```
int i = 0;
#pragma omp parallel
{
  #pragma omp for firstprivate(i)
  for (i = 0; i < N; ++i) {
    a[i] = 0.5 * i;
    b[i] = 2.0 * i;
  }
}

double sum = 0;
#pragma omp parallel for firstprivate(i) reduction(+:sum)
for (i = 0; i < N; ++i)
  sum += a[i] * b[i];
```

work-sharing: sections-directive



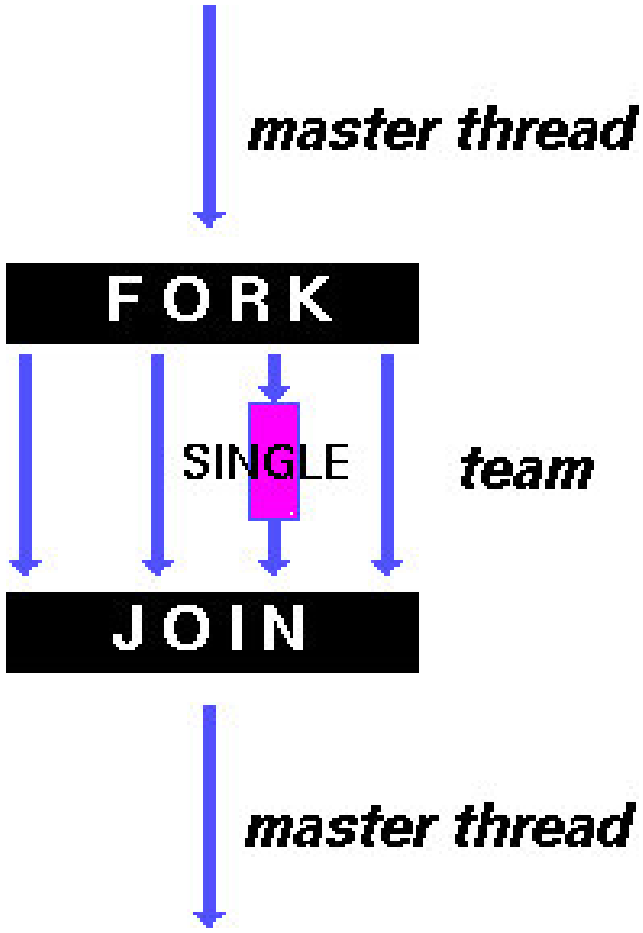
- non-iterative work sharing
- embodies independent section-directives
- each section within sections-directive is executed once(!)
- different sections can be run by different threads of a team

Example: sections-directive

```
#pragma omp parallel
{
  #pragma omp sections
  {
    // execute foo() and bar() concurrently
    #pragma section
    foo();
    #pragma section
    bar();
  }
}
```

work-sharing: single-directive

- code enclosed within a single-directive is executed by only 1 thread in the team
- all other threads wait at the end of the directive
- useful for I/O operations
- similar: master directive with no barrier at the end



Example: single/master-directive

```
#pragma omp parallel
{
  #pragma omp single
  std::cout << "thread " << omp_get_thread_num()
            << " is printing, all others wait\n";
  #pragma omp master
  std::cout << "master is printing, all others continue work\n";
}
```