HPC Lab for CSE

401-3670-00

High-Performance Computing Lab for CSE

FS2024

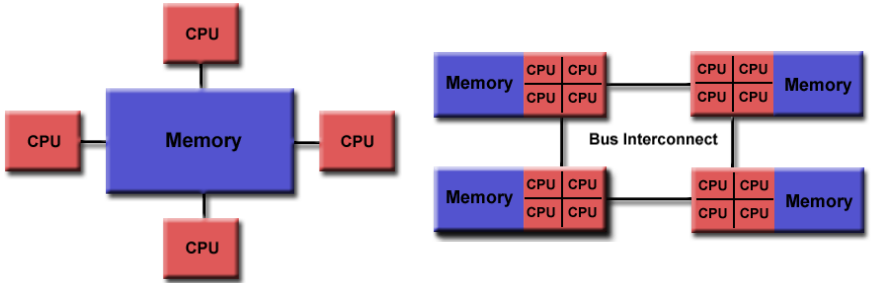
Moodle

Project 02

Parallel Programming using OpenMP

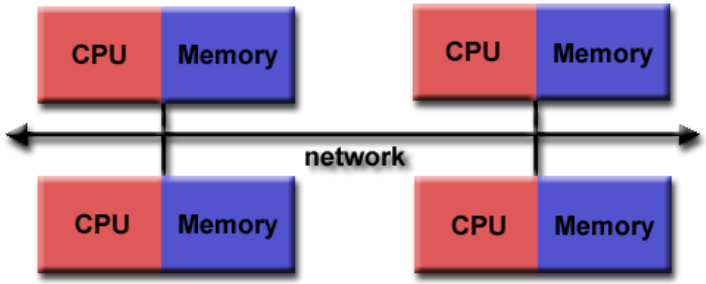
Parallelization: Shared & distributed memory

- Even laptop processors have several “cores” which can perform independent operations
  - all cores can access the same memory: “**shared memory architecture**”



From https://hpc-tutorials.llnl.gov/openmp/

- Today's supercomputers consist of many separate shared memory nodes
  - Data needs to be passed explicitly between nodes: “**distributed memory architecture**”



From https://hpc-tutorials.llnl.gov/mpi/

Parallel scalability

- How well does the program use the parallel resource

Problem size

$$T_1(N) \qquad T_p(N)$$

Runtime

Serial Processors Parallel

- Strong scaling (Amdahl's law)
  - Performance as a function of processors for a fixed-size problem.
- Weak scaling (Gustafson's law)
  - Performance as a function of processors with a proportionally growing problem size.

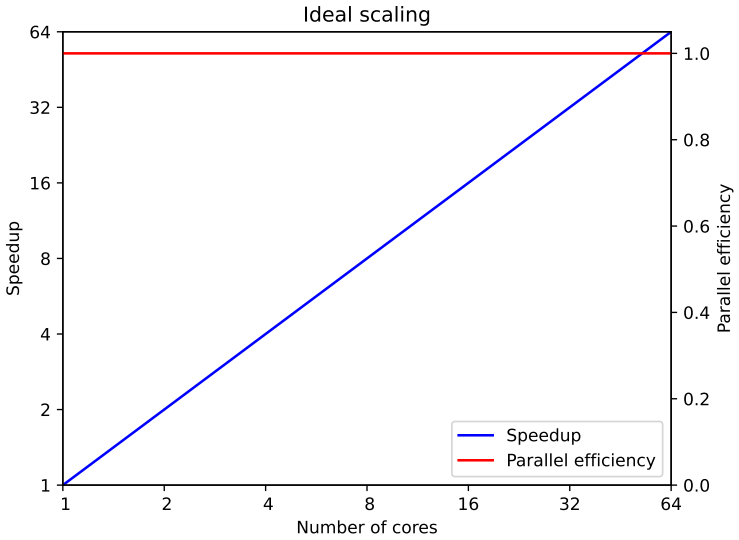
Speedup / Parallel efficiency

- Speedup

$$S_p = \frac{T_1}{T_p}$$

- Parallel efficiency

$$E_p = \frac{T_1}{p \times T_p} = \frac{S_p}{p}$$



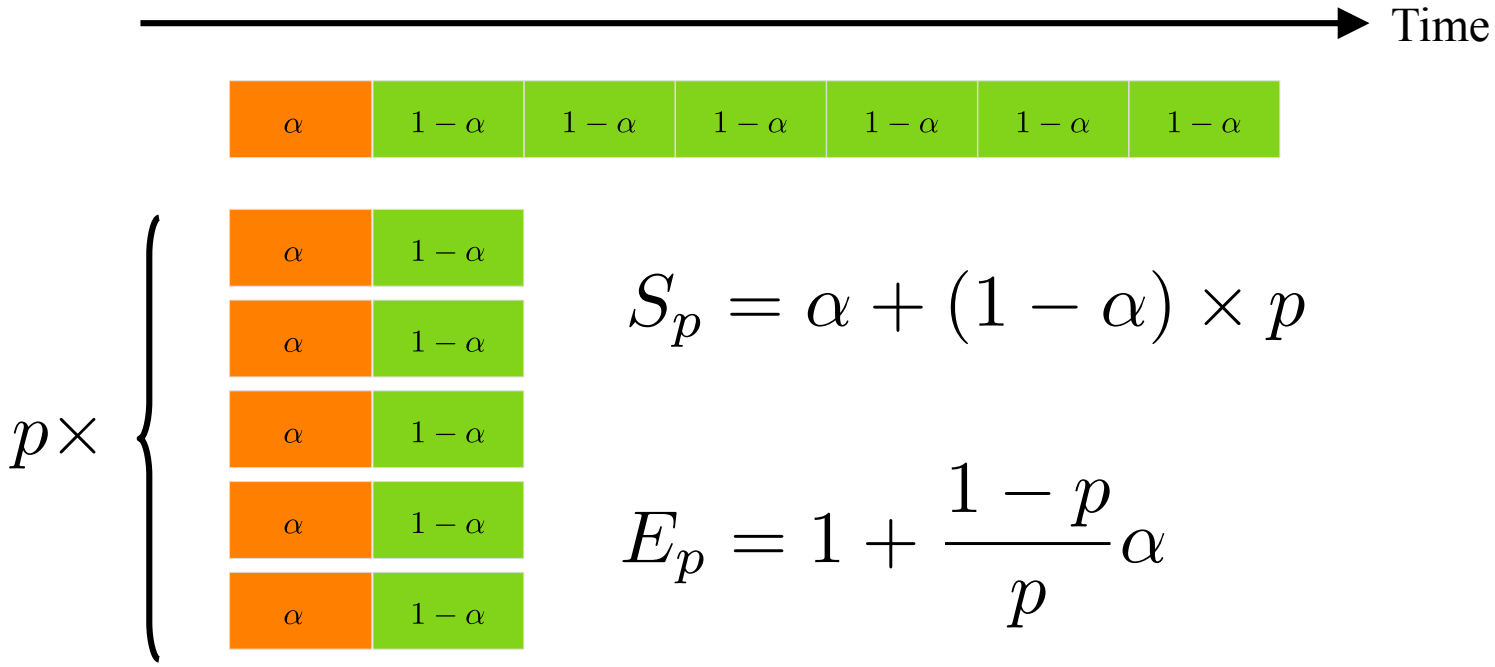
Strong scaling (Amdahl's law)

- Performance as a function of processors for a fixed-size problem.

$$S_p = \frac{T_1}{T_p} = \frac{1}{\alpha + \frac{1-\alpha}{p}} \qquad E_p = \frac{T_1}{p \times T_p} = \frac{1}{(p-1)\alpha + 1}$$

Weak scaling (Gustafson's law)

- Performance as a function of processors with a proportionally growing problem size.



$$S_p = \alpha + (1-\alpha) \times p$$

$$E_p = 1 + \frac{1-p}{p}\alpha$$

Work per processor stays constant.

OpenMP

- OpenMP (**Open** **M**ulti-**P**rocessing) is a parallel programming model for shared-memory architectures.
- Industry standard
  - https://www.openmp.org/
- Scientific computing: “loop-centric”
- Set of compiler directives, library functions and environment variables For writing multi-threaded programs in Fortran, C / C++

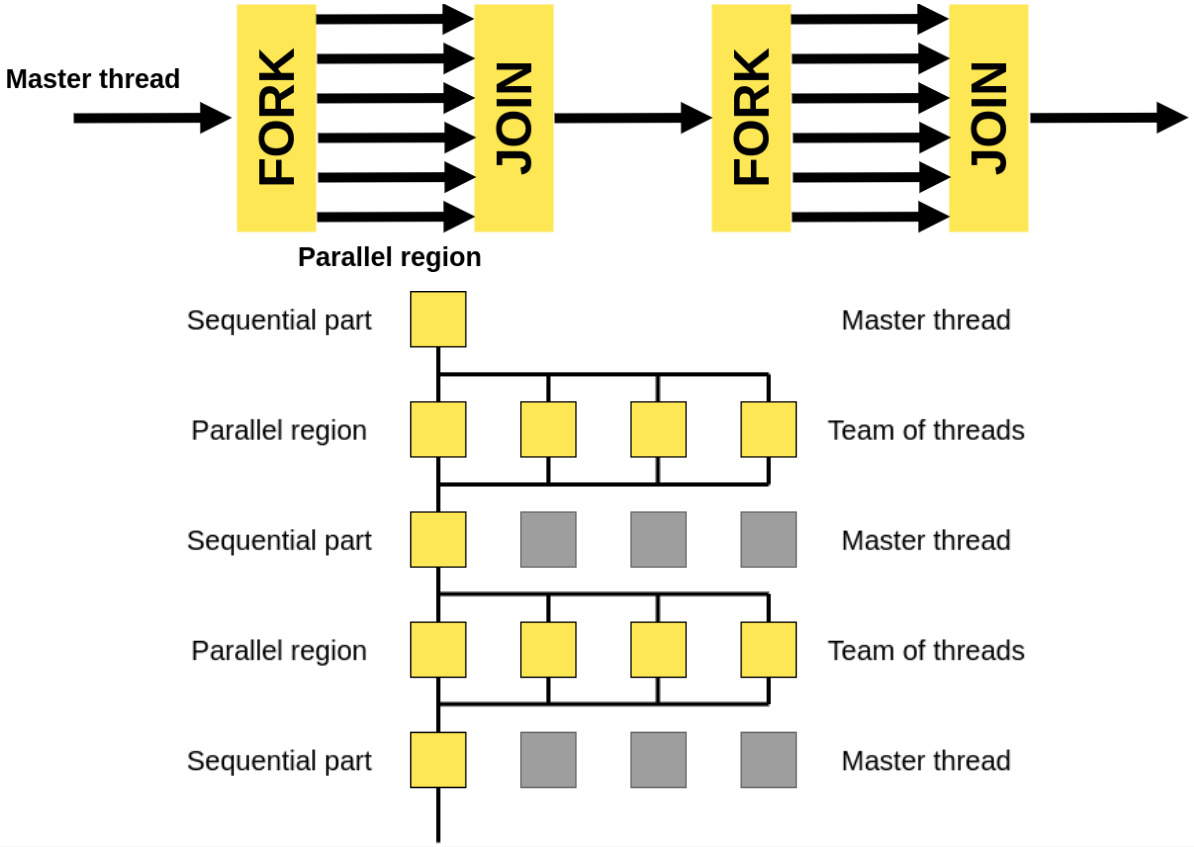
OpenMP: History

- OpenMP 1.x (1997-1999): Parallel loops
- OpenMP 2.x (2000-2005): More loops, F90/F95
- OpenMP 3.x (2008-2011): Tasks
- OpenMP 4.x (2013-2015): SIMD, Accelerators
- OpenMP 5.x (2018-2021): Tasks, latest language support
- OpenMP 6.x (2024?)

OpenMP: Specs, Books, Tutorials

- Specifications
  - https://www.openmp.org/specifications/
  - Contains quite a lot of good examples and can therefore be used as a reference for more information on OpenMP.
- Books:
  - Mattson et al., “The OpenMP Common Core”, 2019: Gives a thorough introduction to OpenMP’s “Common Core”.
  - Chapman et al., “Using OpenMP”, 2017: Gives a thorough introduction to OpenMP but also covers the most relevant performance and correctness issues, together with best practices.
  - ...
- Tutorials:
  - Intel (T. Mattson): https://youtu.be/cMWGeJyrc9w?si=N1ZUdcPrRtdZ5_I4
  - LLNL: https://hpc-tutorials.llnl.gov/openmp/
  - ...

OpenMP: Fork-Join model



OpenMP: An overview

```
/* Environment variables */
export OMP_NUM_THREADS=N // Set the number of threads
/* Library functions */
#include <omp.h>        // Header
int omp_get_num_threads // Get number of threads
int omp_get_thread_num  // Get thread ID [0, 1, ... N-1]
/* Parallel worksharing */
#pragma omp parallel        // Parallel regions
#pragma omp for             // Parallel loop
#pragma omp parallel for    // Combined
reduction(OP:list)          // Reduction of values over team of threads
schedule(static [, chunk])  // Loop scheduling (overhead, load balance)
schedule(dynamic [, chunk])
schedule(guided [, chunk])
/* Data */
private(list), shared(list), firstprivate(list), lastprivate(list), default
/* Synchronization */
#pragma omp barrier
#praga omp critical
nowait
/* Tasks */
#pragma omp single   // Work done by one single thread
#pragma omp task     // Task
#pragma omp taskwait // Task completion
```

OpenMP: Hello world!

```
#include <stdio.h>
#include <omp.h> // OpenMP

int main(int argc, char *argv[]) {
  // Get some OpenMP info & report collected info
  int nthreads = 0;
  int tid = 0;
#pragma omp parallel private(nthreads, tid)
  {
    nthreads = omp_get_num_threads();
    tid = omp_get_thread_num();
    printf("Hello world from thread %3d out of %d\n", tid, nthreads);
  }
  return 0;
}
```

Data management (Here: each thread has its own)

Structured block

```
$ gcc -fopenmp hello_omp.c -o hello_omp
$ export OMP_NUM_THREADS=8
$ ./hello_omp
Hello world from thread   0 out of 8
Hello world from thread   6 out of 8
Hello world from thread   7 out of 8
Hello world from thread   1 out of 8
Hello world from thread   3 out of 8
Hello world from thread   5 out of 8
Hello world from thread   2 out of 8
Hello world from thread   4 out of 8
```

OpenMP: Dot

```
double dot = 0.;
for (int i = 0; i < n; ++i) {
  dot += a[i] * b[i];
}
```

```
double dot = 0.;
#pragma omp parallel
{
  for (int i = 0; i < n; ++i) {
    dot += a[i] * b[i];
  }
}
```

OpenMP: Dot

Partial dot for each thread

Get parallel environment

Evenly distribute loop

Synchronize

```
double dot = 0.;
#pragma omp parallel
{
  double partial_dot = 0.;
  int nthreads = omp_get_num_threads();
  int tid = omp_get_thread_num();
  int i_beg =  tid      * n / nthreads;
  int i_end = (tid + 1) * n / nthreads;
  for (int i = i_beg; i < i_end; ++i) {
    partial_dot += a[i] * b[i];
  }
  #pragma omp critical
  dot += partial_dot;
}
```

OpenMP: Dot

```
double dot = 0.;
#pragma omp parallel
{
  #pragma omp for reduction(+:dot)
  for (int i = 0; i < n; ++i) {
    dot += a[i] * b[i];
  }
}
```

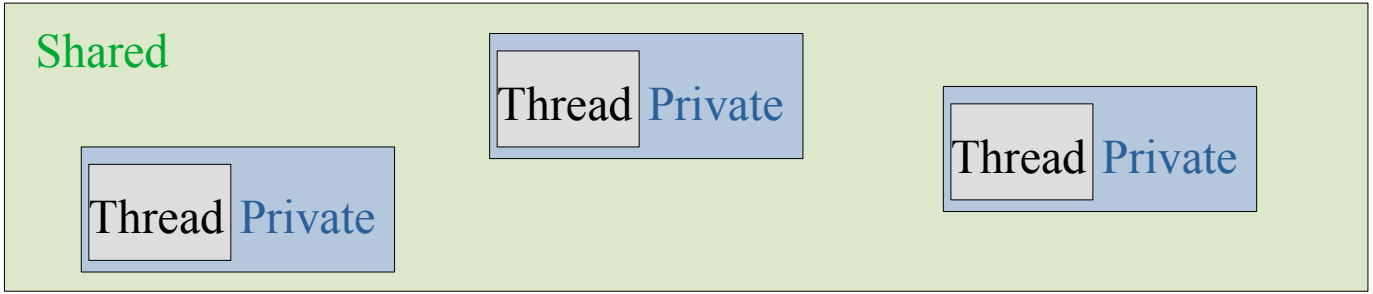
Reduction clause

Combined construct

```
double dot = 0.;
#pragma omp parallel for reduction(+:dot)
for (int i = 0; i < n; ++i) {
  dot += a[i] * b[i];
}
```

OpenMP: Data environment

- OpenMP storage attributes
  - Shared
    - All threads in a team can access (rw) the variable
    - Default: Variables declared outside of parallel region
  - Private:
    - Only one thread can access (rw) the variable
    - Default: Variables declared inside of parallel region



OpenMP: Data environment

```
double dot = 0.;
#pragma omp parallel
{
  double partial_dot = 0.;
  int nthreads = omp_get_num_threads();
  int tid = omp_get_thread_num();
  int i_beg =  tid      * n / nthreads;
  int i_end = (tid + 1) * n / nthreads;
  for (int i = i_beg; i < i_end; ++i) {
    partial_dot += a[i] * b[i];
  }
  #pragma omp critical
  dot += partial_dot;
}
```

Shared Private

OpenMP: Data environment

```
double dot = 0.;
#pragma omp parallel
{
  double partial_dot = 0.;
  int nthreads = omp_get_num_threads();
  int tid = omp_get_thread_num();
  int i_beg =  tid      * n / nthreads;
  int i_end = (tid + 1) * n / nthreads;
  for (int i = i_beg; i < i_end; ++i) {
    partial_dot += a[i] * b[i];
  }
  #pragma omp critical
  dot += partial_dot;
}
```

Shared Private

OpenMP: Data environment

```
double dot = 0.;
double partial_dot = 0.;
int nthreads, tid, i_beg, i_end;
#pragma omp parallel                          \
        default(none)                         \
        shared(n, a, b, dot)                  \
        private(nthreads, tid, i_beg, i_end) \
        firstprivate(partial_dot)
{
  nthreads = omp_get_num_threads();
  tid = omp_get_thread_num();
  i_beg =  tid       * n / nthreads;
  i_end = (tid + 1) * n / nthreads;
  for (int i = i_beg; i < i_end; ++i) {
    partial_dot += a[i] * b[i];
  }
  #pragma omp critical
  dot += partial_dot;
}
```

OpenMP: Tasks

- For not “loop-centric” problems
- Example: Fibonacci

```c
#include <stdio.h>

int fib(int n) {
  if (n < 2) return n;
  return fib(n - 2) + fib(n - 1);
}

int main() {
  int n = 42;
  printf("Fibonacci fib(%2d) = %d\n", n, fib(n));
  return 0;
}
```

```c
#include <stdio.h>

int fib(int n) {
  int Fnm, Fnmm;
  if (n < 2) return n;
  #pragma omp task shared(Fnmm)
  Fnmm = fib(n - 2);
  #pragma omp task shared(Fnm)
  Fnm  = fib(n - 1);
  #pragma omp taskwait
  return Fnmm + Fnm;
}

int main() {
  int n = 42;
  int fibn;
  #pragma omp parallel
  {
    #pragma omp single
    fibn = fib(n);
  }
  printf("Fibonacci fib(%2d) = %d\n", n, fibn);
  return 0;
}
```

Only executed by one thread

This is just for the task example. This is indeed a terrible way to compute the Fibonacci numbers!