- Part 2 -
Parallel Programming

Vincent Heuveline
The Challenge

- Make your algorithms ready for
  - Fine-grained parallelism
  - Scalability with respect to thousands of threads
  - Data locality

But how?
The big picture

**Hardware evolution**

- Memory wall: Data movement cost prohibitively expensive
- Power wall: Nuclear power plant for each machine (in the cloud)?
- ILP wall: ‘Automagic’ maximum resource utilisation?
- Memory wall + power wall + ILP wall = brick wall

**Inevitable paradigm shift: Parallelism and heterogeneity**

- In a single chip: singlecore → multicore, manycore, …
- In a workstation (cluster node): NUMA, CPUs and GPUs, …
- In a big cluster: different nodes, communication characteristics, …

**This is our problem as applied mathematicians**

- Affects all machines we use, including workstations and laptops
Conflicting developments

Complexity of cores
Heterogeneous Platforms

Responsibility of programmer

Easy living + Limited performance

Scalability + Performance increase
Speedup in parallel computing

The **speedup** is defined as the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on $p$ processors.

$$S = \frac{T_s}{T_p}$$
Speedup in parallel computing
Efficiency in parallel computing

The efficiency is defined as the ratio of speedup to the number of processors. Efficiency measures the fraction of time for which a processor is usefully utilized.

\[
E = \frac{S}{p} = \frac{T_s}{pT_p}
\]
Amdahl’s law

\[ T = (1 - p)T + pT \]

\[ T(s) = (1 - p)T + \frac{p}{s}T \]

\[ \frac{T}{T(s)} = \frac{1}{1 - p + \frac{p}{s}} \]
Amdahl’s law

Parallelism on the Chip Level

- Parallelism on the Chip-Level
  - Multi-Core CPUs
  - GPUs
  - ...

- Memory shared over the cores

=> direct memory accesses possible!
Parallelism on the Cluster Level

- HPC-Cluster consists of several nodes, each of which has
  - Computing instance
  - Memory instances

- Memory distributed within the cluster

=> direct memory accesses only locally
=> beyond: network communication
Paradigms of Parallel Programming

• **Shared memory**
  - Different processors/threads share main memory

• **Distributed memory (message passing)**
  - Each processor has its own memory

• Hybrid approach
  PGAS (Partitioned Global Address Space)
  Global memory address space, but portions of the memory space may have an affinity for particular processes/threads
Paradigms of Parallel Programming

- **Shared memory**
  - Different processors/threads share main memory
  - => **OpenMP** (Open Multi-Processing) \(\rightarrow\) C/C++/Fortran API for shared memory multiprocessing

- **Distributed memory (message passing)**
  - Each processor has its own memory
  - => **MPI** (Message Passing Interface) \(\rightarrow\) Standard for library routines for message-passing programs (C/C++/Fortran/Java/...)

- **Hybrid approach**
  PGAS (Partitioned Global Address Space)
  Global memory address space, but portions of the memory space may have an affinity for particular processes/threads
OpenMP: Overview

- OpenMP is an easy, portable specification for node-level parallelisation
- Thread-based, shared memory, single-node (in contrast to MPI)

How does it work?
- Annotate the C/C++/FORTRAN source code with pragmas
- The compiler transparently generates the necessary code
- Fork and join model
  - Non-parallel blocks are only executed by the main (or master) thread
  - Parallel blocks are executed in parallel by a team-of-threads

- OpenMP pragmas are ignored of not activated
OpenMP: Fork and join model

OMP parallel region

Master thread

Fork workers
Worker threads
Syncronization

OMP parallel region

Syncronization

SC14 OpenMP tutorial, C. Terboven et al
OpenMP: Core syntax

- Most OpenMP pragmas apply to a „structured block“ or „parallel region“

```c
#pragma omp parallel
{
    // statements
}
```

- Only statements inside a block marked with the „parallel“ clause will be executed in parallel
OpenMP: Core syntax / Compiling

- Most OpenMP pragmas apply to a „structured block“ or „parallel region“

```c
#pragma omp parallel
{
    // statements
}
```

- Only statements inside a block marked with the „parallel“ clause will be executed in parallel

- GCC
  ```
gcc -fopenmp -o example example.c
  ```

- Intel
  ```
  icc -openmp -o example example.c
  ```
OpenMP: Hello world example

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

using namespace std;

int main(int argc, char **argv)
{
    // fork threads
    #pragma omp parallel
    {
        cout << "Thread "
        << omp_get_thread_num() << endl;
    }
    // join threads
    cout << "All done!" << endl;
    return 0;
}
```

$ export OMP_NUM_THREADS=4
$ ./main
Thread 0
Thread 3
Thread 2
Thread 1
All done!
MPI: Overview

- **MPI** is a **standardized** and **portable message-passing system** which defines the syntax and the semantics of **core library routines**
- Several implementations of MPI exist, e.g. for C, C++, Fortran, Java

- **How does it work?**
  - Message-passing between several processes through calls of MPI functions
  - Communicators (defines groups of processes)
  - Point-to-point communication, e.g. MPI_Send or MPI_Receive
  - Collective functions, e.g. all-to-one or one-to-all communication
  - Linking against a MPI library necessary
  - Parallel program is typically started using mpirun/mpiexec (agent)
  - → assignment of processes and CPUs
  By default, full code is executed by every process
MPI: Message passing approach

Process 0 often treated as special process (not necessary)

One-to-all communication

All-to-one communication
MPI: Core syntax (1)

- The library header has to be included, e.g. in C/C++

```c
#include <mpi.h>
```

- The MPI execution environment has to be initialized at the beginning and terminated at the end of a program, e.g. in C/C++

```c
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    // statements
    MPI_Finalize();
}
```

- All processes run the same code
MPI: Core syntax (2)

• One-to-one communication

```c
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

```c
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Explanation of parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>initial address of send/receive buffer (choice)</td>
</tr>
<tr>
<td>count</td>
<td>number of elements in send/receive buffer (nonnegative integer)</td>
</tr>
<tr>
<td>datatype</td>
<td>datatype of each send/receive buffer element (handle)</td>
</tr>
<tr>
<td>dest / source</td>
<td>rank of destination/source (integer)</td>
</tr>
<tr>
<td>tag</td>
<td>message tag (integer)</td>
</tr>
<tr>
<td>comm</td>
<td>communicator (handle)</td>
</tr>
<tr>
<td>status</td>
<td>status object (Status)</td>
</tr>
</tbody>
</table>
MPI: Hello world example

```cpp
#include <iostream>
#include <mpi.h>

using namespace std;

int main(int argc, char **argv)
{
    int size, rank;

    // initialize the MPI execution environment
    MPI_Init(&argc, &argv);

    // determine rank and size
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    cout << "Hello world from rank " << rank << endl;
    if (rank == 0)
        cout << "Size = " << size << endl;

    // terminate MPI execution environment
    MPI_Finalize();

    return 0;
}
```

$ mpirun -np 2 ./main
Hello world from rank 0
Size = 2
Hello world from rank 1
Debugging: Approaches and Tools

- New problems due to parallelism, e.g.
  - Deadlocks
  - Race conditions
  - Irreproducibility

- Approaches and Tools, e.g.
  - Valgrind (Memory Tracing Tools)
  - Marmot (MPI-Analysis Tool)
  - Total View (Debugger)
  - DDT (Debugger)
Multi-GPU implementation for multiple compute nodes

- Multiple compute nodes, each equipped with GPUs, are connected over the network using MPI
- An additional layer of asynchronism is added to the algorithm
Hybrid parallel configurations

• MPI, OpenMP & CUDA

- from Kepler on: HyperQ feature, 32 concurrent CUDA streams
- must use Multi-Process Service to share one GPU
Thank you