Scientific Computing WS 2019/2020

Lecture 27

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de
Why parallelization?

Clock rate of processors limited due to physical limits
⇒ parallelization: main road to increase the amount of data processed
Parallel systems nowadays ubiquitous: even laptops and smartphones have multicore processors
Amount of accessible memory per processor is limited ⇒ systems with large memory can be created based on parallel processors
**TOP 500 2019 rank 1-9**

<table>
<thead>
<tr>
<th>Rank</th>
<th>System</th>
<th>Cores</th>
<th>Rmax [TFlop/s]</th>
<th>Rpeak [TFlop/s]</th>
<th>Power [kW]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States</td>
<td>2,414,592</td>
<td>148,600.0</td>
<td>200,794.9</td>
<td>10.096</td>
</tr>
<tr>
<td>2</td>
<td>Sierra - IBM Power System AC922, IBM POWER9 22C 3.10GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States</td>
<td>1,572,480</td>
<td>94,640.0</td>
<td>125,712.0</td>
<td>7.438</td>
</tr>
<tr>
<td>3</td>
<td>Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRPC National Supercomputing Center in Wuxi China</td>
<td>10,649,600</td>
<td>93,014.6</td>
<td>125,435.9</td>
<td>15.371</td>
</tr>
<tr>
<td>4</td>
<td>Tianhe-2A - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.20GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China</td>
<td>4,981,760</td>
<td>61,444.5</td>
<td>100,678.7</td>
<td>18.482</td>
</tr>
<tr>
<td>5</td>
<td>Frontera - Dell C6420, Xeon Platinum 8280 28C 2.70GHz, Mellanox InfiniBand HDR, Dell EMC Texas Advanced Computing Center/Univ. of Texas United States</td>
<td>648,448</td>
<td>23,516.4</td>
<td>38,745.9</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.60GHz, Arias interconnect, NVIDIA Tesla P100, Cray/HPE Swiss National Supercomputing Centre (CSCS) Switzerland</td>
<td>387,872</td>
<td>21,230.0</td>
<td>27,154.3</td>
<td>2.884</td>
</tr>
<tr>
<td>7</td>
<td>Trinity - Cray XC40, Xeon E5-2698v3 16C 2.30GHz, Intel Xeon Phi 7250 68C 1.40GHz, Arias interconnect, Cray/HPE DOE/NNSA/LANL/SLNL United States</td>
<td>979,072</td>
<td>20,158.7</td>
<td>41,461.2</td>
<td>7.578</td>
</tr>
<tr>
<td>8</td>
<td>AI Bridging Cloud Infrastructure (ABC1) - PRIMERGY CX2570 M4, Xeon Gold 6148 20C 2.40GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR, Fujitsu National Institute of Advanced Industrial Science and Technology (AIST) Japan</td>
<td>391,680</td>
<td>19,880.0</td>
<td>32,576.6</td>
<td>1.649</td>
</tr>
<tr>
<td>9</td>
<td>SuperMUC-NG - ThinkSystem SD650, Xeon Platinum 8174 24C 3.10GHz, Intel Omni-Path, Lenovo Leibniz Rechenzentrum Germany</td>
<td>305,856</td>
<td>19,476.6</td>
<td>26,873.9</td>
<td></td>
</tr>
</tbody>
</table>

- Based on linpack benchmark: solution of dense linear system
- Typical desktop computer: $R_{max} \approx 100 \ldots 1000 \text{GFlop/s}$

[Source: www.top500.org ]
Parallel paradigms

SIMD
Single Instruction Multiple Data

- prev instr
- load A(1)
- load B(1)
- C(1)=A(1)*B(1)
- store C(1)
- next instr

- prev instr
- load A(2)
- load B(2)
- C(2)=A(2)*B(2)
- store C(2)
- next instr

- prev instr
- load A(n)
- load B(n)
- C(n)=A(n)*B(n)
- store C(n)
- next instr

P1  P2  Pn

MIMD
Multiple Instruction Multiple Data

- prev instr
- load A(1)
- load B(1)
- C(1)=A(1)*B(1)
- store C(1)
- next instr

- prev instr
- call funcD
- x=y*z
- sum=x^2
- call sub1(i,j)
- next instr

- prev instr
- do 10 i=1,N
- alpha=w^*3
- zeta=C(i)
- 10 continue
- next instr

P1  P2  Pn

- "classical" vector systems: Cray, Convex . . .
- Graphics processing units (GPU)

- Shared memory systems
  - IBM Power, Intel Xeon, AMD Opteron . . .
  - Smartphones . . .
  - Xeon Phi R.I.P.

- Distributed memory systems
  - interconnected CPUs

[Source: computing.llnl.gov/tutorials]
MIMD Hardware: Distributed memory

- Create large computer system by connecting standard mainboards via fast network
- Memory scales with number of CPUs interconnected
- High latency for communication
- Mostly programmed using MPI (Message passing interface)
- Explicit programming of communications: gather data, pack, send, receive, unpack, scatter

```c
MPI_Send(buf,count,type,dest,tag,comm)
MPI_Recv(buf,count,type,src,tag,comm,stat)
```
MIMD Hardware: Shared Memory

Symmetric Multiprocessing (SMP)/Uniform memory access (UMA)

- Similar processors
- Similar memory access times

Shared memory: one (virtual) address space for all processors involved
- Communication hidden behind memory access
- Not easy to scale large numbers of CPUs
- MPI works on these systems as well

Nonuniform Memory Access (NUMA)

- Possibly varying memory access latencies
- Combination of SMP systems
- ccNUMA: Cache coherent NUMA

[Source: computing.llnl.gov/tutorials]
Hybrid distributed/shared memory

- Combination of shared and distributed memory approach
- Top 500 computers

- Shared memory nodes can be mixed CPU-GPU
- Need to master three kinds of programming paradigms:
  - SIMD (GPU)
  - Shared memory
  - Distributed memory

[Source: computing.llnl.gov/tutorials]
“small” parallel system: this laptop

- **1 NUMANode (aka. CPU chip)**
  - 12 MB L3 cache
  - 6 Cores
    - 256KB L2 Cache
    - 32KB L1 Cache
    - Hyperthreading → 2 logical cores (PU)
- **32GB RAM accessible via 3.9 GB/s DMA channels (dma0, dma1)**
- **Graphics card card0 (NVIDIA T1000) via 4GB/s connect**
- **SSD nvme0n1 (1TB) via 3.9 GB/s connect**
- **WIFI (wlp111s0), LAN (em1) ...**
"large" parallel system: WIAS compute server erhard-01

- 4 NUMANodes
  - each node: 256 GB RAM, 30 MB L3 cache, 10 cores
    - each core: 256KB L2 Cache, 32KB L1 Cache, 2 logical cores (PU)
  - Network . . .
Modern operating systems allow to run several programs at once.

Each of these programs corresponds to a *process*.

Processes can be launched from the command line and require large bookkeeping, each process has its own address space.

On multicore systems, processes can run on different cores, and ideally, they don’t interfere with each other.

Data exchange between different processes needs an extra protocol for inter-process communication.
Threads vs processes

- Threads are lightweight subprocesses within a process and share its address space, they can run on a different core.
- Managing a thread requires significantly less bookkeeping and resources compared to a process.
- Parallel programming using threads aka. multithreading is easy, as inter-thread communication can be realized via the common address space.
- Multithreading is hard since threads share data structures that should only be modified by one thread at a time.
Thread based programming model

- pthreads (POSIX threads): widely available on different operating systems
- Threads introduced into C++ standard with C++11
- Cumbersome tuning + synchronization, but very flexible
- Basic structure for higher level interfaces
- Threads in Julia: ‘Threads.@spawn‘ (since Julia 1.3), marked as experimental

... sequential code ...

function run_in_thread(params) // function to be run in separate thread
    ...
end

t=start_thread (run_in_thread, params) //
...

wait_and_fetch_result(t)
...
Fork-Join programming model

- OpenMP for C++, C, Fortran
- ‘Threads.@threads‘ in Julia
- Compiler directives (pragmas) describe parallel regions
- Automatically mapped onto thread based model

```plaintext
... sequential code ... // joined code
#pragma omp parallel for // `fork' -> parallel execution
{
  ... parallel code ...
}
(implicit barrier) // wait for tasks to finish
... sequential code ...
```

[Source: computing.llnl.gov/tutorials]
Fork-join vs thread based

- Usually, the fork-join model is implemented on top of the threading model
- OpenMP essentially performs automatic code transformation
- Well adapted to numerical tasks with large loops
- Easy to handle
- Performance depends on compiler implementation, memory bandwidth etc.
OpenMP $s = u \cdot v$: primitive implementation

```c
double s=0.0;
#pragma omp parallel for
for(int i=0; i<n ; i++)
s+=u[i]*v[i];
```

- Code can be parallelized by introducing compiler directives
- Compiler directives are ignored if not in parallel mode
- Compiler directives are not part of the language
- Write conflict with $s+ =:_{several}$ threads may access the same variable
Preventing conflicts in OpenMP

- Atomic updates are performed only by one thread at a time

```c
double s=0.0;
#pragma omp parallel for
for(int i=0; i<n ; i++)
{
    #pragma omp atomic update
    s+=u[i]*v[i];
}
```

- Expensive, parallel program flow is interrupted

- Similar to Julia atomic variables
Do it yourself reduction

- Remedy: accumulate partial results per thread, combine them after main loop

- "Reduction"

```c
#include <omp.h>
int maxthreads=omp_get_max_threads();
double s0[maxthreads];
double u[n], v[n];
for (int ithread=0; ithread<maxthreads; ithread++)
    s0[ithread]=0.0;

#pragma omp parallel for
for (int i=0; i<n; i++)
{
    int ithread=omp_get_thread_num();
    s0[ithread]+=u[i]*v[i];
}

double s=0.0;
for (int ithread=0; ithread<maxthreads; ithread++)
    s+=s0[ithread];
```
OpenMP Reduction Variables

- In standard situations, reduction variables can be used to avoid write conflicts, no need to organize this by programmer.
Parallelization of vector operations

- For iterative methods it is important to parallelize vector operations:
  - Scalar product
  - Basic operation: \( \text{x} = ax + y \)
  - Sparse matrix \( \times \) vector
- Few operations per memory access, relatively fine grained parallelism
- \( \rightarrow \) benchmark this
  - STREAM benchmark
  - “Schönauer vector triad”: \( d = a + b \times c \)
    - 4 vectors, 2 Flops per index.
Memory performance: vector triad I

This laptop (gcc)

- Intel Core i7-9850H 2.60GHz 6 Cores
- RAM: 32GiB L1: 6x32KB, L2: 6x256KB, L3: 12MB

- 1 thread
- 2 threads
- 4 threads
- 6 threads

<table>
<thead>
<tr>
<th>Array length N</th>
<th>GFlops/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^3</td>
<td>10^4</td>
</tr>
<tr>
<td>10^4</td>
<td>10^5</td>
</tr>
<tr>
<td>10^5</td>
<td>10^6</td>
</tr>
<tr>
<td>10^6</td>
<td>10^7</td>
</tr>
<tr>
<td>10^7</td>
<td>10^8</td>
</tr>
</tbody>
</table>

WIAS compute server (gcc)

- Intel Xeon E5-2698 2.30GHz 2x32 Cores
- RAM: 512GiB L1: 64x42KB, L2: 64x256KB, L3: 2x40MB

- 1 thread
- 2 threads
- 4 threads
- 8 threads
- 16 threads
- 32 threads
- 64 threads

<table>
<thead>
<tr>
<th>Array length N</th>
<th>GFlops/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^3</td>
<td>10^4</td>
</tr>
<tr>
<td>10^4</td>
<td>10^5</td>
</tr>
<tr>
<td>10^5</td>
<td>10^6</td>
</tr>
<tr>
<td>10^6</td>
<td>10^7</td>
</tr>
<tr>
<td>10^7</td>
<td>10^8</td>
</tr>
</tbody>
</table>

- Small problems: scalar is fastest due to scheduling overhead
- Medium problems: parallel is fast
- Large problems: no big difference due to memory bandwidth: all laptop cores access the memory through the same bottleneck
- “sweet spot” for parallel between $10^4$ and $5 \cdot 10^5 \ldots 5 \cdot 10^6$
Memory performance: vector triad II

Benchmarking site of G. Hager

https://blogs.fau.de/hager/archives/tag/benchmarking

Istopo for this laptop

- Performance drops are correlated with cache sizes
- Most important: large L3 cache
Memory performance: vector triad III

gcc vs. icc (2013)

https://blogs.fau.de/hager/archives/tag/benchmarking

this laptop (4 threads)

- Picture similar to early times of gnu compiler vs Intel
- Julia barrier implementation seems to need improvement
- “hand crafted” threading works better
Memory performance: vector triad: update

- Picture similar to early times of gnu compiler vs Intel
- Julia barrier implementation seems to need improvement
- “hand crafted” threading works better
double u[n], v[n];
#pragma omp parallel for
for (int i=0; i<n; i++)
u[i] += a*u[i];

- Distribution of indices with thread is implicit and can be influenced by scheduling directives
- Number of threads can be set via OMP_NUM_THREADS environment variable or call to omp_set_num_threads()
- First Touch Principle: first thread which “touches” data triggers the allocation of memory with the NUMA node where the thread is running on
Parallelization of PDE solution with multithreading

\[ \Delta u = f \text{ in } \Omega, \quad u|_{\partial \Omega} = 0 \]

\[ \Rightarrow u = \int_{\Omega} f(y)G(x, y)dy. \]

- Solution in \( x \in \Omega \) is influenced by values of \( f \) in all points in \( \Omega \)
- \( \Rightarrow \) global coupling: any solution algorithm needs global communication
Structured and unstructured grids

Structured grid
- Easy next neighbor access via index calculation
- Efficient implementation on SIMD/GPU
- Strong limitations on geometry

Unstructured grid
- General geometries
- Irregular, index vector based access to next neighbors
- Hardly feasible for SIMD/GPU

[Quelle: tetgen.org]
Stiffness matrix assembly for Laplace operator for P1 FEM

\[ a_{ij} = a(\phi_i, \phi_j) = \int_{\Omega} \nabla \phi_i \nabla \phi_j \, dx \]

\[ = \int_{\Omega} \sum_{K \in T_h} \nabla \phi_i |_K \nabla \phi_j |_K \, dx \]

Assembly loop:
Set \( a_{ij} = 0 \).
For each \( K \in T_h \):
For each \( m, n = 0 \ldots d \):

\[ s_{mn} = \int_{K} \nabla \lambda_m \nabla \lambda_n \, dx \]

\[ a_{j_{dof}(K,m),j_{dof}(K,n)} = a_{j_{dof}(K,m),j_{dof}(K,n)} + s_{mn} \]
Mesh partitioning

- Partition set of elements $K$ in $T_h$, and color the neighborhood graph of the partitions
- Result:
  - $C$: set of colors
  - $P_c$: set of partitions of given color
- Then: $T_h = \bigcup_{c \in C} \bigcup_{p \in P_c} \{K\}_{K \in p}$

Sample algorithm:
- Subdivision of grid cells into equally sized subsets by METIS (Karypis/Kumar) → Partitions of color 1
- Create separators along boundaries → Partitions of color 2
- “triple points” → Partitions of color 3
Parallel stiffness matrix assembly for P1 FEM

- No interference between assembly loops for partitions of the same color
- Immediate parallelization without critical regions

Set $a_{ij} = 0$.

For each color $c \in \mathcal{C}$

```c
#pragma omp parallel for
    For each $p \in \mathcal{P}_c$:
        For each $K \in p$:
            For each $m, n = 0 \ldots d$:
                $s_{mn} = \int_K \nabla \lambda_m \nabla \lambda_n \, dx$
                $a_{j_{dof}(K,m),j_{dof}(K,n)} += s_{mn}$
```

- Prevent write conflicts by loop organization
- No need for critical sections
- Similar structure for Voronoi finite volumes, nonlinear operator evaluation, Jacobi matrix assembly
Sparse matrices

Direct solvers are hard to parallelize though many efforts are undertaken, e.g. Pardiso

Iterative methods easier to parallelize

- partitioning of vectors + coloring inherited from cell partitioning
- keep loop structure (first touch principle)
- parallelize
  - vector algebra
  - scalar products
  - matrix vector products
  - preconditioners

But: barrier overhead, memory access bandwidth are essential for efficiency
Distributed memory computing

- Based on different processes (instead of threads) running on one or multiple hosts
- Generally: Communication via network
- Communication via POSIX shared memory if running on the same host
- Communications need to be programmed explicitly.
- Paradigms:
  - Master - Worker
  - Single program - multiple data (SPMD)
MPI - Message passing interface

- library, can be used from C, C++, Fortran, python
- de facto standard for programming on distributed memory systems (since \( \approx 1995 \))
- highly portable
- MPI.jl julia package
- support by hardware vendors: optimized communication speed
- based on sending/receiving messages over network
- SPMD paradigm
- need to hand-craft communications
How to install

- OpenMP/C++11 threads come along with compiler
- MPI needs to be installed in addition
- Can run on multiple systems
- openmpi available for Linux/Mac (homebrew)/ Windows (cygwin)
  - [https://www.open-mpi.org/faq/?category=mpi-apps](https://www.open-mpi.org/faq/?category=mpi-apps)
  - Compiler wrapper mpic++
    - wrapper around (configurable) system compiler
    - proper flags + libraries to be linked
  - Process launcher mpirun
- launcher starts a number of processes which execute statements independently, occasionally waiting for each other
Threads vs processes

- MPI is based on processes, C++11 threads and OpenMP are based on threads.
- Processes are essentially like commands launched from the command line and require large bookkeeping, each process has its own address space.
- Threads are created within a process and share its address space, require significantly less bookkeeping and resources.
- Multithreading requires careful programming since threads share data structures that should only be modified by one thread at a time. Unlike threads, with processes there can be no write conflicts.
- When working with multiple processes, one becomes responsible for inter-process communication.
MPI Programming Style

- Generally, MPI allows to work with completely different programs.
- Typically, one writes one program which is started in multiple incarnations on different hosts in a network or as different processes on one host.
- MPI library calls are used to determine the identity of a running program and the region of the data to work on.
- Communication + barriers have to be programmed explicitly.
// Initialize MPI.
MPI_Init ( &argc, &argv );

// Get the number of processes.
MPI_Comm_size ( MPI_COMM_WORLD, &nproc );

// Determine the rank (number, identity) of this process.
MPI_Comm_rank ( MPI_COMM_WORLD, &iproc );

if ( iproc == 0 )
{
    cout << "Number of available processes: " << nproc << "\n";
}
cout << "Hello from proc " << iproc << endl;
MPI_Finalize ( );

- Compile with mpic++ mpi-hello.cpp -o mpi-hello
- All MPI programs begin with MPI_Init() and end with MPI_Finalize()
- the communicator MPI_COMM_WORLD designates all processes in the current process group, there may be other process groups etc.
- The whole program is started \( N \) times as system process, not as
Distribute code execution over several hosts
- MPI gets informed how many independent processes can be run on which node and distributes the required processes accordingly
- MPI would run more processes than slots available. Avoid this situation!
- Need ssh public key access and common file system access for proper execution
- Telling mpi to use host file:
  mpirun --hostfile hostfile -np N mpi-hello
MPI Send

MPI\_Send (start, count, datatype, dest, tag, comm)

- Send data to other process(es)
- The message buffer is described by (start, count, datatype):
  - start: Start address
  - count: number of items
  - datatype: data type of one item
- The target process is specified by dest, which is the rank of the target process in the communicator specified by comm
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.
- The tag codes some type of message
MPI Receive

MPI_Recv(start, count, datatype, source, tag, comm, status)

- Waits until a matching (on source and tag) message is received from the system, and the buffer can be used.
- source is rank in communicator specified by comm, or MPI_ANY_SOURCE
- status contains further information
- Receiving fewer than count occurrences of datatype is OK, but receiving more is an error.
MPI_Bcast(start, count, datatype, root, comm )

- Broadcasts a message from the process with rank “root” to all other processes of the communicator
- Root sends, all others receive.
Differences with OpenMP

- Programmer has to care about all aspects of communication and data distribution, even in simple situations.

- In simple situations (regularly structured data) OpenMP provides reasonable defaults. For MPI these are not available.

- For PDE solvers (FEM/FVM assembly) on unstructured meshes, in both cases we have to care about data distribution.

- We need explicit handling of data at interfaces with MPI, while with OpenMP, possible communication is hidden behind the common address space.