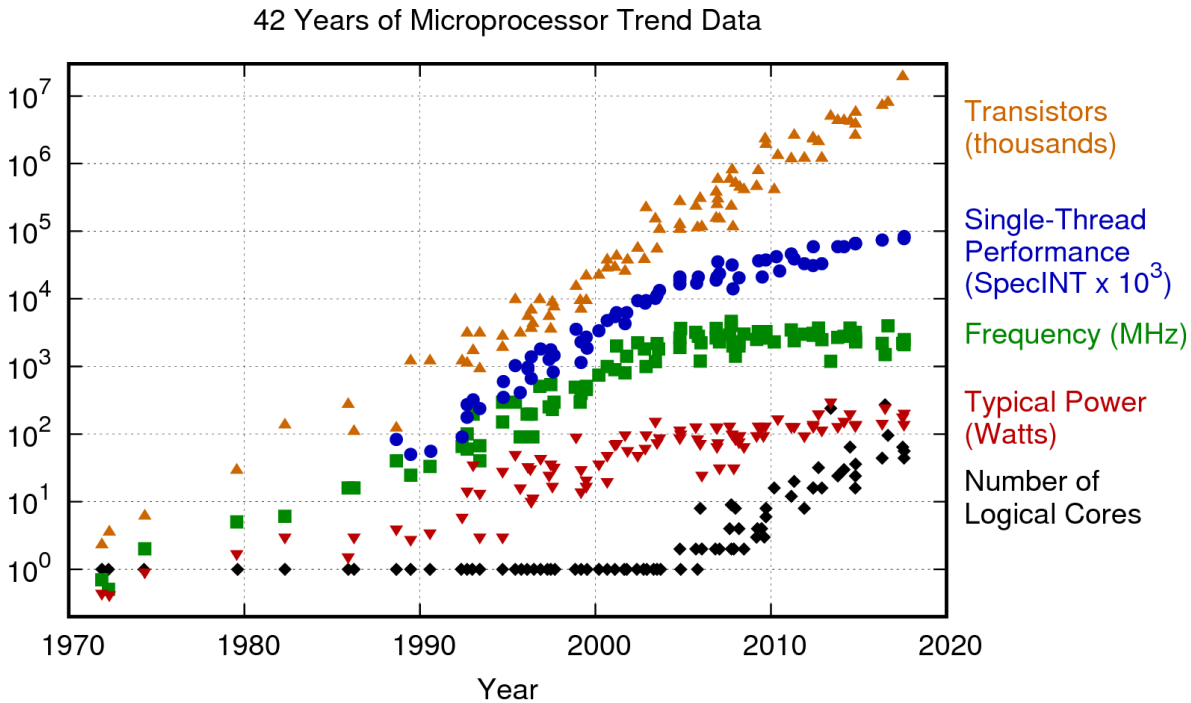
Scientific Computing WS 2019/2020

Lecture 27

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

Why parallelization ?



- Clock rate of processors limited due to physical limits
- $\Rightarrow$ 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 $\Rightarrow$ systems with large memory can be created based on parallel processors

TOP 500 2019 rank 1-9

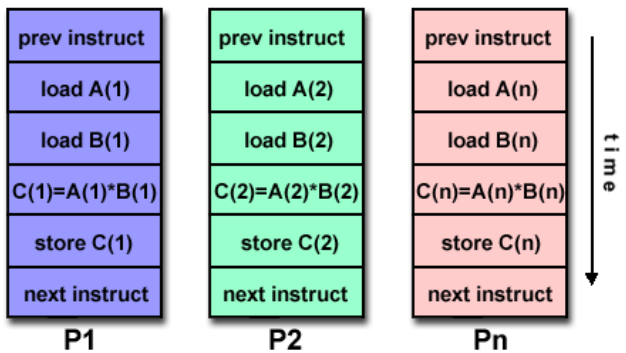
Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
2	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
3	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
4	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
5	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR , Dell EMC Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9	
6	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray/HPE Swiss National Supercomputing Centre (CSCS) Switzerland	387,872	21,230.0	27,154.3	2,384
7	**Trinity** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray/HPE DOE/NNSA/LANL/SNL United States	979,072	20,158.7	41,461.2	7,578
8	**AI Bridging Cloud Infrastructure (ABCI)** - PRIMERGY CX2570 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR , Fujitsu National Institute of Advanced Industrial Science and Technology (AIST) Japan	391,680	19,880.0	32,576.6	1,649
9	**SuperMUC-NG** - ThinkSystem SD650, Xeon Platinum 8174 24C 3.1GHz, Intel Omni-Path , Lenovo Leibniz Rechenzentrum Germany	305,856	19,476.6	26,873.9	

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

[Source:www.top500.org]

Parallel paradigms

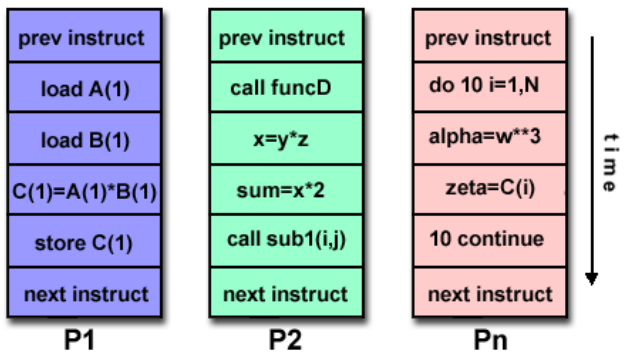
SIMD
Single Instruction Multiple Data



[Source: computing.llnl.gov/tutorials]

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

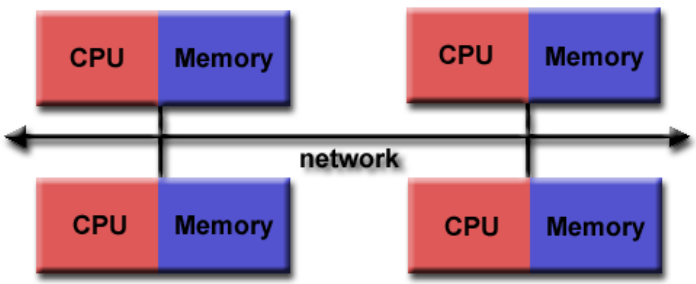
MIMD
Multiple Instruction Multiple Data



[Source: computing.llnl.gov/tutorials]

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

MIMD Hardware: Distributed memory



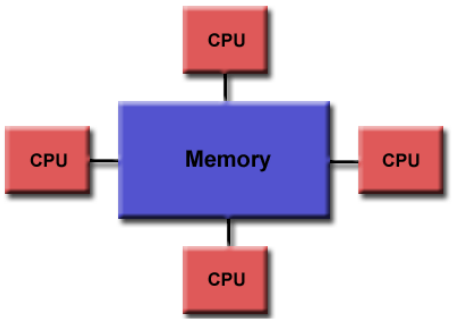
[Source: computing.llnl.gov/tutorials]

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

```
MPI_Send(buf,count,type,dest,tag,comm)
MPI_Recv(buf,count,type,src,tag,comm,stat)
```

MIMD Hardware: Shared Memory

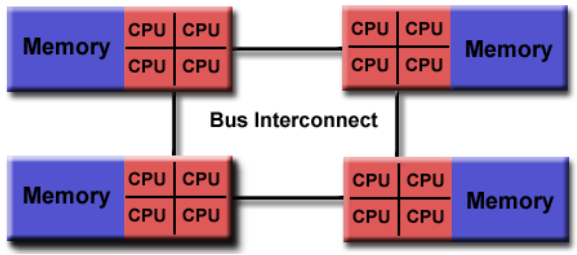
Symmetric Multiprocessing (SMP)/Uniform memory access (UMA)



[Source: computing.llnl.gov/tutorials]

- Similar processors
- Similar memory access times

Nonuniform Memory Access (NUMA)



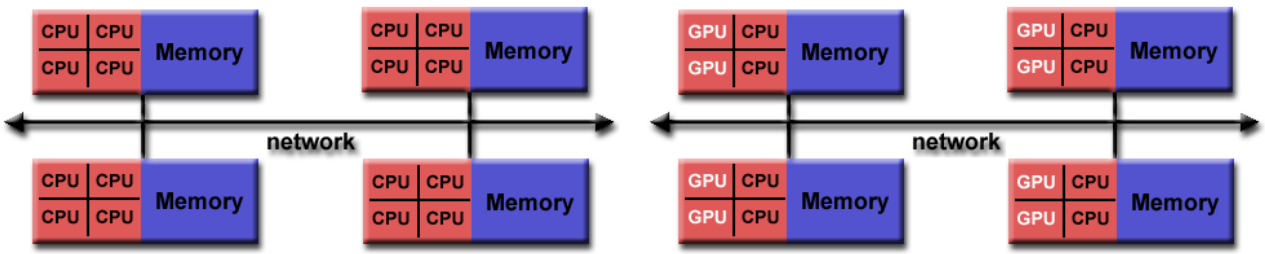
[Source: computing.llnl.gov/tutorials]

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

- 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

Hybrid distributed/shared memory

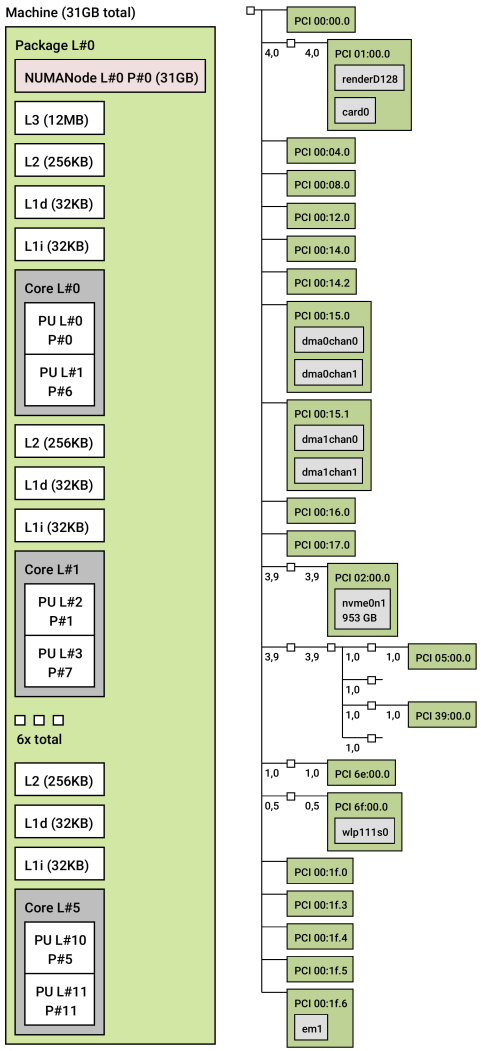
- Combination of shared and distributed memory approach
- Top 500 computers



[Source: computing.llnl.gov/tutorials]

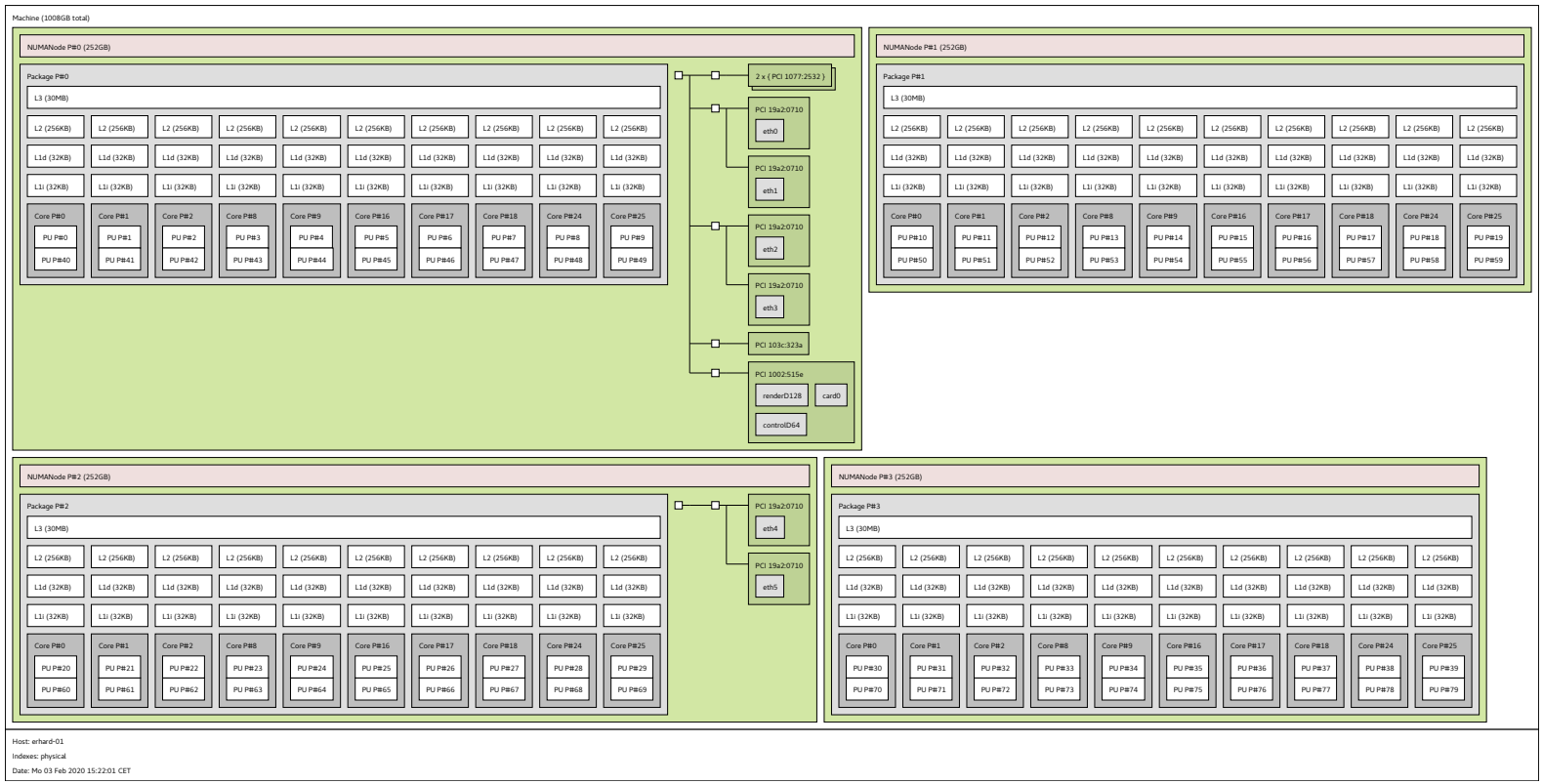
- Shared memory nodes can be mixed CPU-GPU
- Need to master three kinds of programming paradigms:
  - SIMD (GPU)
  - Shared memory
  - Distributed memory

“small” parallel system: this laptop



- 1 NUMANode (aka. CPU chip)
  - 12 MB L3 cache
  - 6 Cores
    - 256KB L2 Cache
    - 32KB L1 Cache
    - Hyperthreading $\rightarrow$ 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 . . .

Parallel processes

- 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 bookeeping, 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 bookeeping 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

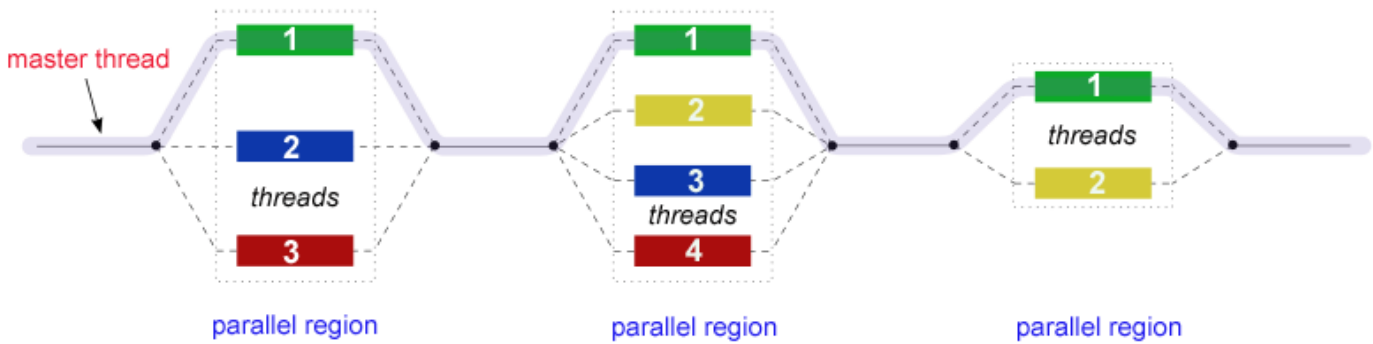
- pthreads (POSIX threads): widely available on different operating systems
- Threads introduced into C++ standard with C++11
- Cumbersome tuning + syncronization, but very flexible
- Basic structure for higher level interfaces
- Threads in Julia: 'Threads.@spawn' (since Julia 1.3), marked as experimental

```
... 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

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



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

```
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 languge
- 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

```
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"

```
#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

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

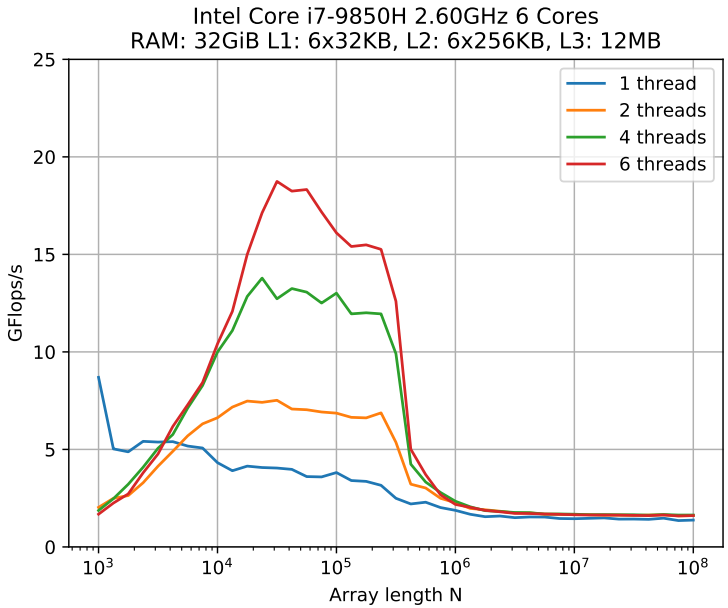
- In standard situations, reduction variables can be used to avoid write conflicts, no need to organize this by programmer

Parallelization of vector operations

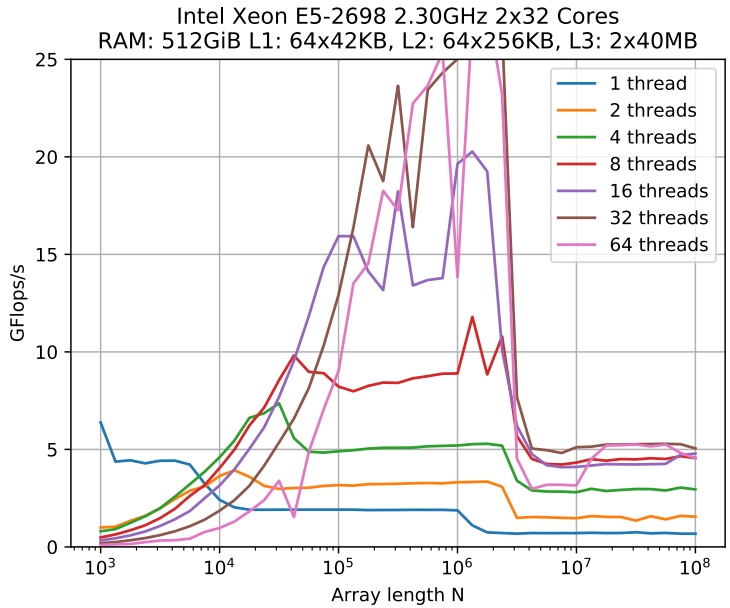
- For iterative methods it is important to parallelize vecotor operations:
  - Scalar product
  - Basic operationx
  - 'axpy' $\mathbf{x} = a\mathbf{x} + \mathbf{y}$
  - Sparse matrix $\times$ vector
- Few operations per memory access, relatively fine grained parallelism
- $\rightarrow$ benchmark this
  - STREAM benchmark
  - "Schönauer vector triad": $\mathbf{d} = \mathbf{a} + \mathbf{b} * \mathbf{c}$
    4 vectors, 2 Flops per index.

Memory performance: vector triad I

This laptop (gcc)



WIAS compute server (gcc)

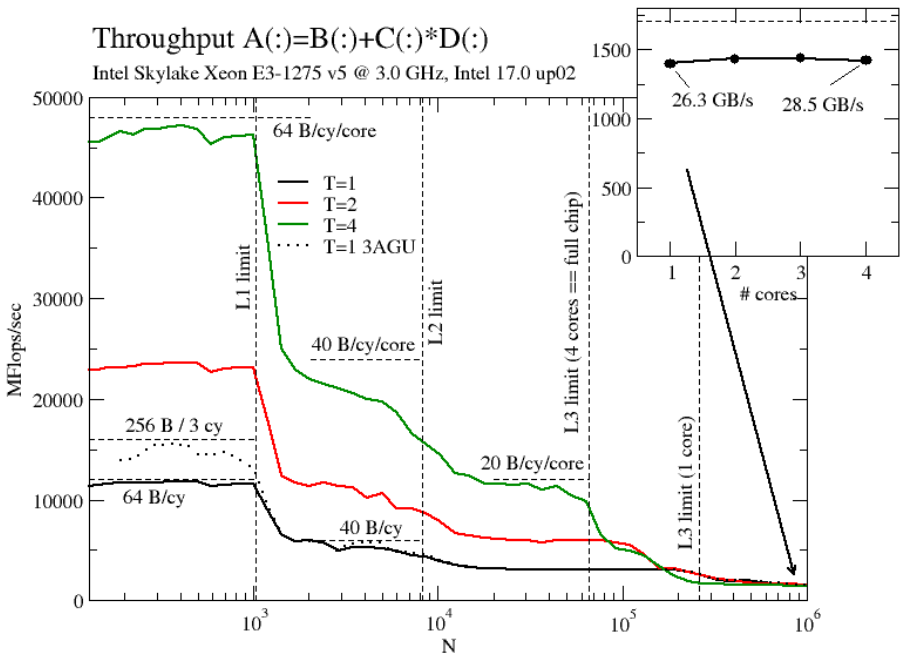


- Small problems: scalar is fastest due to scheduling overhead
- Medium problems: parallel is fast
- Large problems: no big difference due to memory bandwidth: all laptop cores access the memory through the same bottleneck
- "sweet spot" for parallel between 10^4 and $5 \cdot 10^5 \ldots 5 \cdot 10^6$

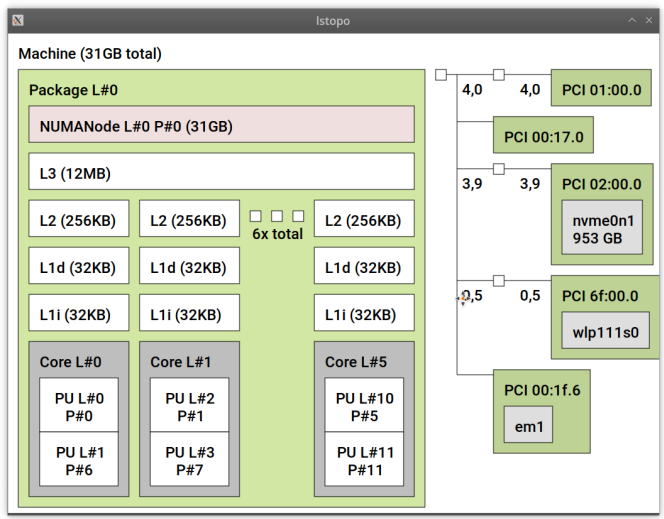
Memory performance: vector triad II

Benchmarking site of G. Hager

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



lstopo for this laptop

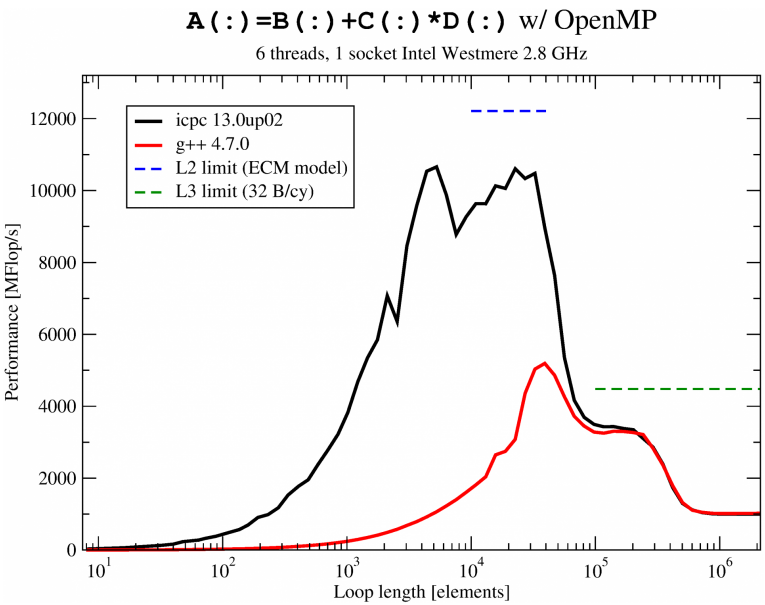


- Performance drops are correlated with cache sizes
- Most important: large L3 cache

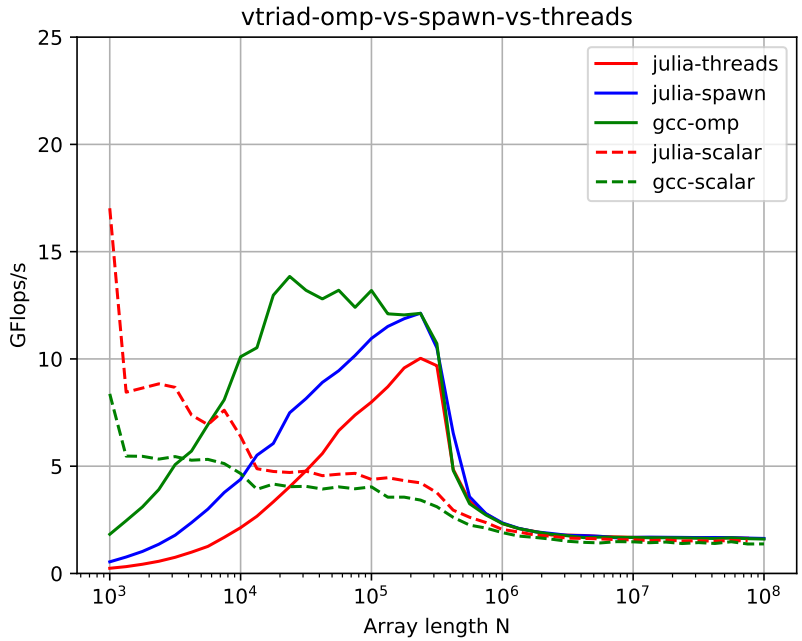
Memory performance: vector triad III

gcc vs. icc (2013)

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



this laptop (4 threads)



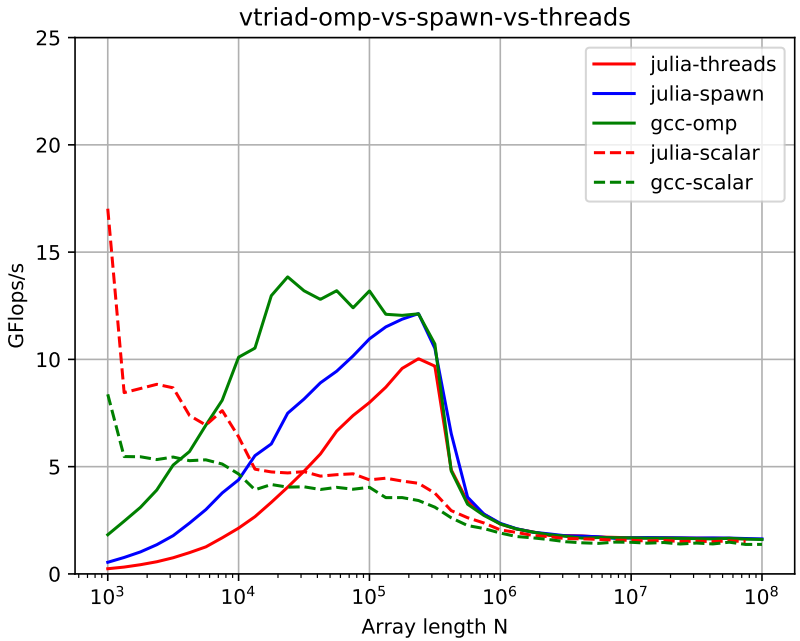
- Picture similar to early times of gnu compiler vs Intel
- Julia barrier implementation seems to need improvement
- "hand crafted" threading works better

Memory performance: vector triad: update

Julia vs gcc on Server

pic/vtriad-omp-vs-spawn-vs-t

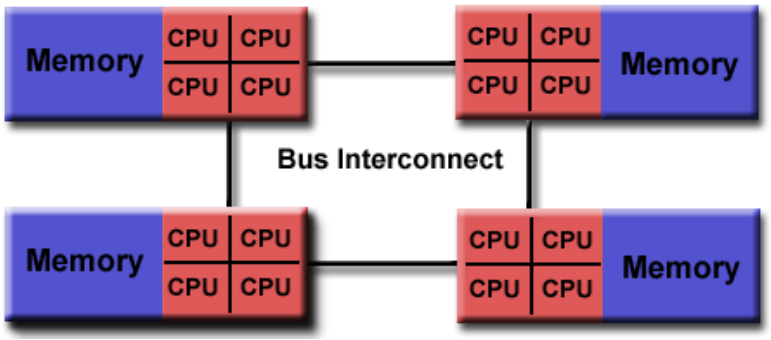
Julia vs gcc on Laptop



- Picture similar to early times of gnu compiler vs Intel
- Julia barrier implementation seems to need improvement
- "hand crafted" threading works better

OpenMP: further aspects

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



[Quelle: computing.llnl.gov/tutorials]

- 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

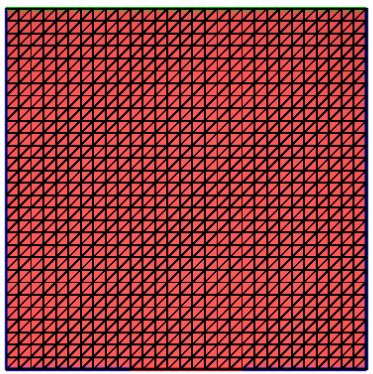
$$\Delta u = f \text{ in} \Omega, \qquad u|_{\partial\Omega} = 0$$

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

- Solution in $x \in \Omega$ is influenced by values of f in all points in Ω
- $\Rightarrow$ global coupling: any solution algorithm needs global communication

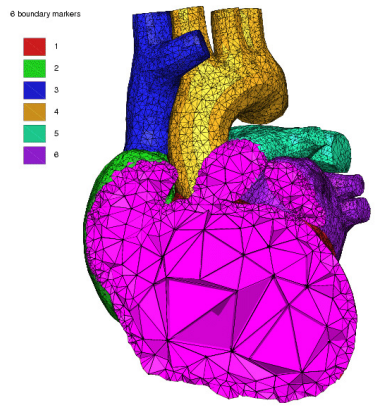
Structured and unstructured grids

Structured grid



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

Unstructured grid



[Quelle: tetgen.org]

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

Stiffness matrix assembly for Laplace operator for P1 FEM

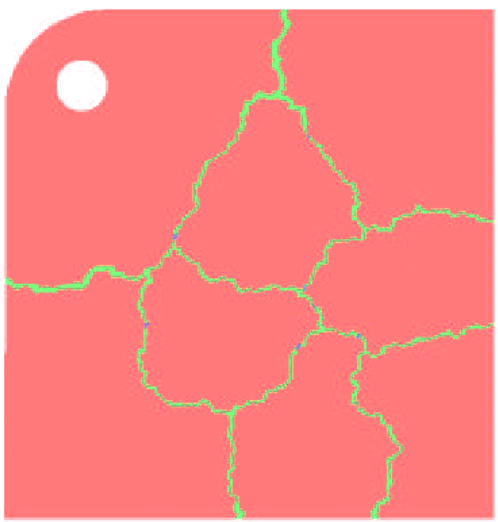
$$a_{ij} = a(\phi_i, \phi_j) = \int_\Omega \nabla\phi_i \nabla\phi_j \; dx$$
$$= \int_\Omega \sum_{K \in \mathcal{T}_h} \nabla\phi_i|_K \nabla\phi_j|_K \; dx$$

Assembly loop:
Set $a_{ij} = 0$.
For each $K \in \mathcal{T}_h$:
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 $\mathcal{T}_h$, and color the neighborhood graph of the partitions
- Result:
  - $\mathcal{C}$: set of colors
  - $\mathcal{P}_c$: set of partitions of given color
- Then: $\mathcal{T}_h = \bigcup_{c \in \mathcal{C}} \bigcup_{p \in \mathcal{P}_c} \{K\}_{K \in p}$



Sample algorithm:

- Subdivision of grid cells into equally sized subsets by METIS (Karypis/Kumar) $\rightarrow$ Partitions of color 1
- Create separators along boundaries $\rightarrow$ Partitions of color 2
- "triple points" $\rightarrow$ 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 $c \in \mathcal{C}$
#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

Linear system solution

- 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 explicitely.
- 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`
  - 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, ocassionally 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 bookeeping, each process has its own address space
- Threads are created within a process and share its address space, require significantly less bookeeping 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 identiy of a running program and the region of the data to work on
- Communication + barriers have to be programmed explicitely.

MPI Hello world

```
//  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

MPI hostfile

```
host1  slots=n1
host2  slots=n2

...
```

- 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 Broadcast

```
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