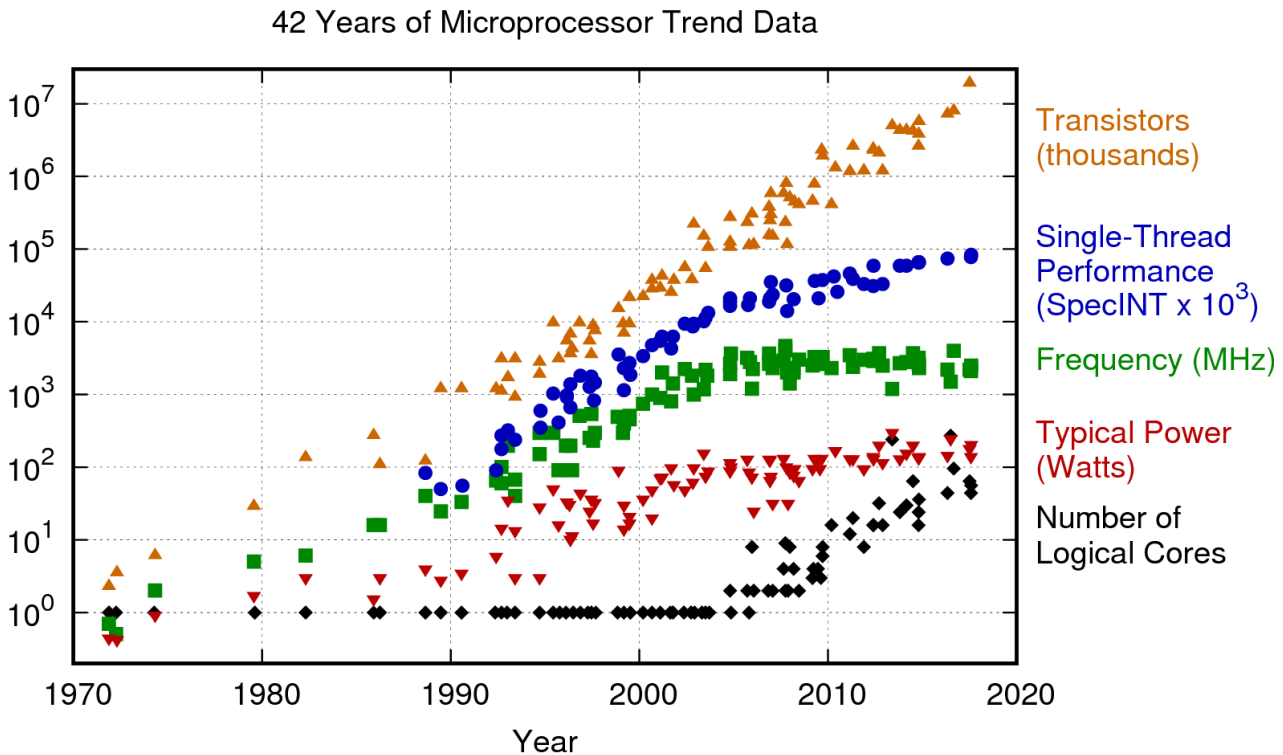
Scientific Computing WS 2019/2020

Lecture 26

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

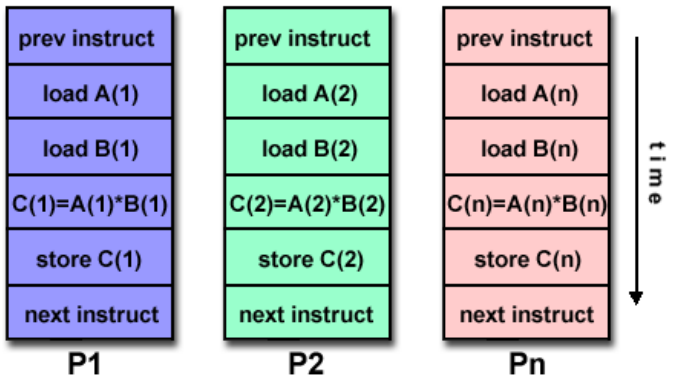
Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
2	**Sierra** - IBM Power System 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

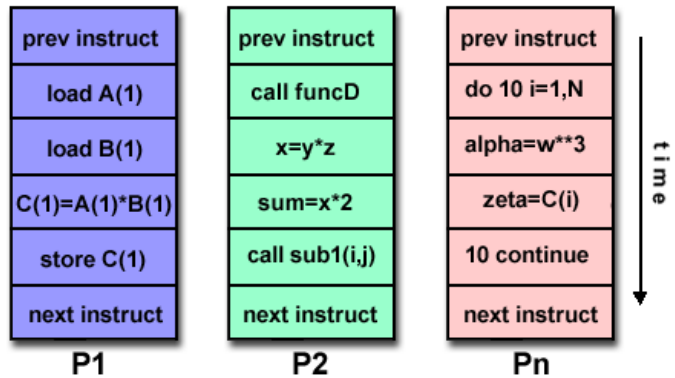
SIMD
Single Instruction Multiple Data



[Source: computing.llnl.gov/tutorials]

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

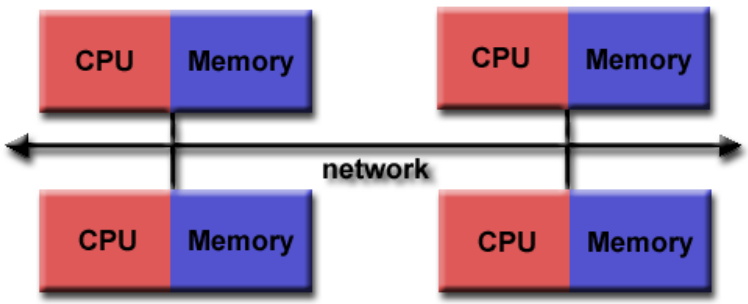
MIMD
Multiple Instruction Multiple Data



[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



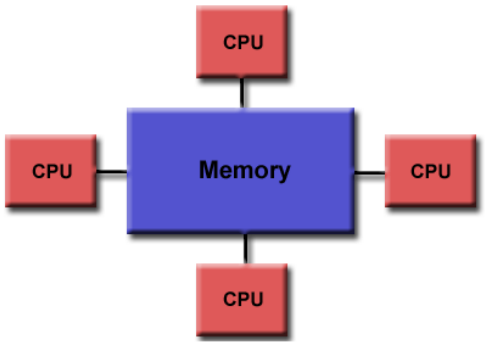
[Source: computing.llnl.gov/tutorials]

- 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

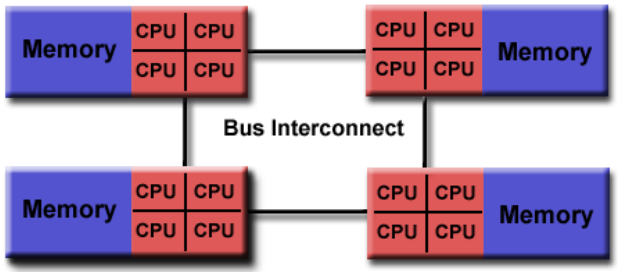
Symmetric Multiprocessing (SMP)/Uniform memory access (UMA)



[Source: computing.llnl.gov/tutorials]

- Similar processors
- Similar memory access times

Nonuniform Memory Access (NUMA)



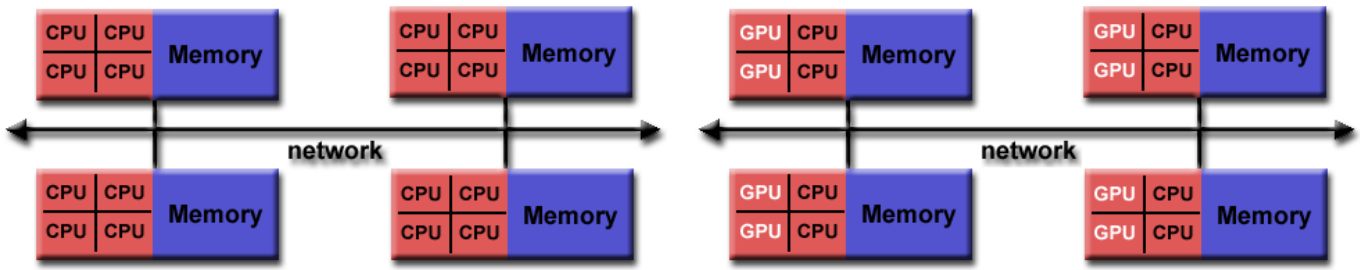
[Source: computing.llnl.gov/tutorials]

- 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

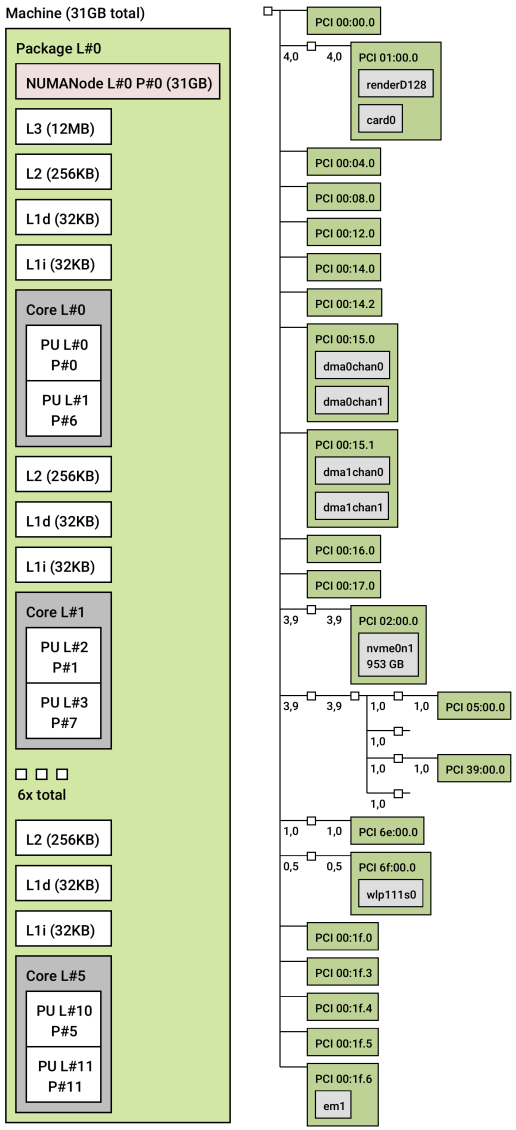
- Combination of shared and distributed memory approach
- Top 500 computers



[Source: computing.llnl.gov/tutorials]

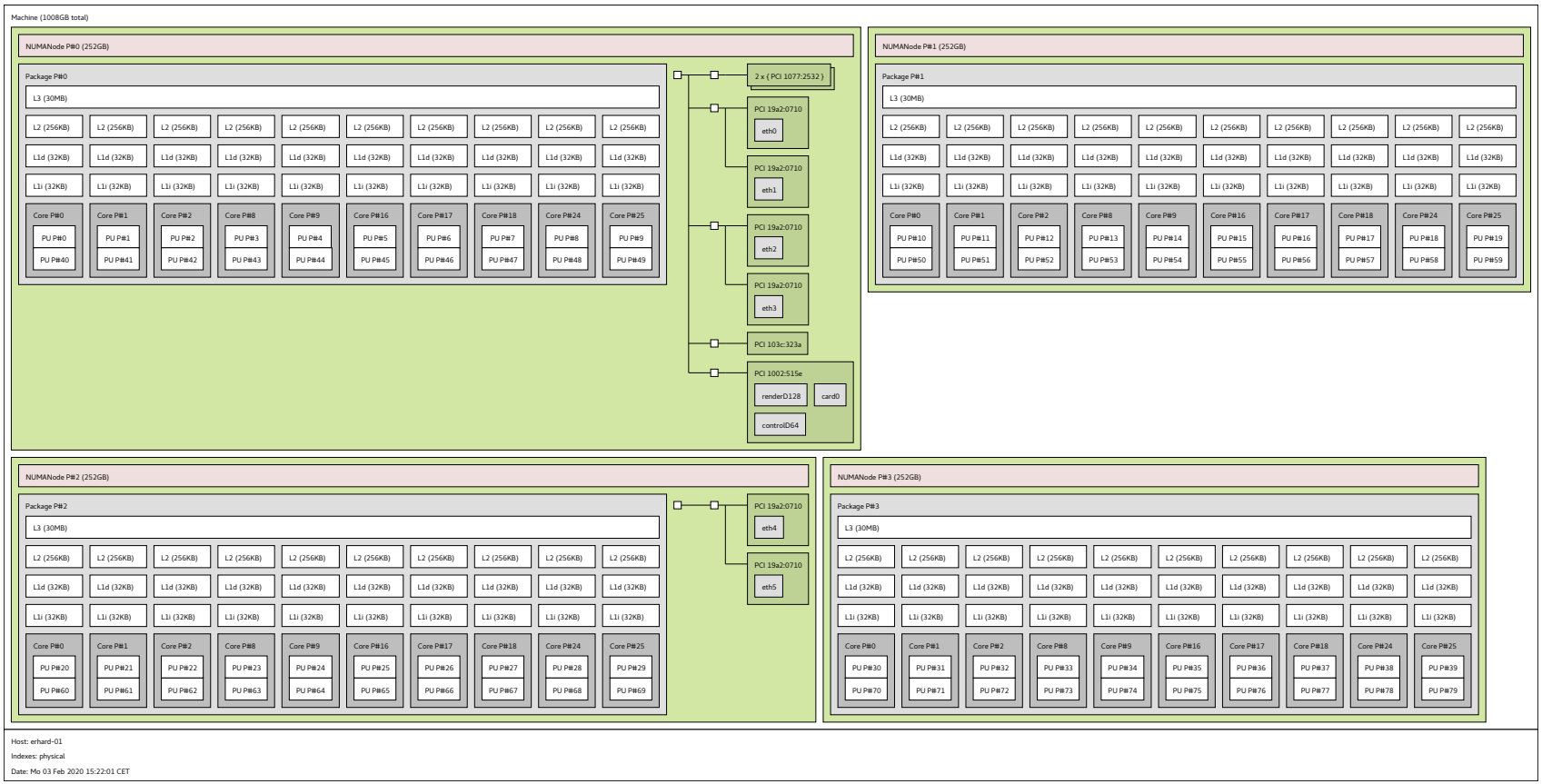
- Shared memory nodes can be mixed CPU-GPU
- Need to master three kinds of programming paradigms:
  - SIMD (GPU)
  - Shared memory
  - Distributed memory

“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

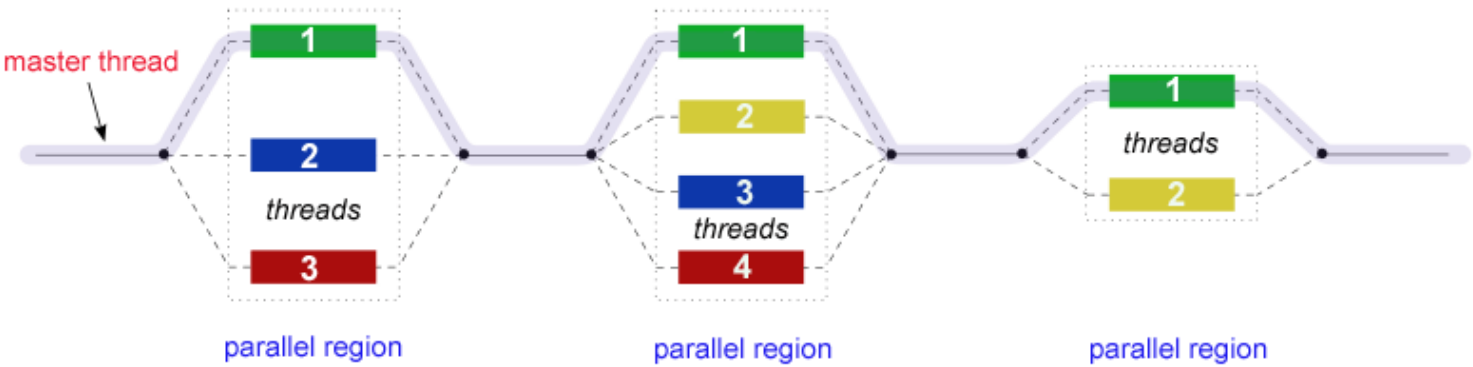
- pthreads (POSIX threads): widely available on different operating systems
- Threads introduced into C++ standard with C++11
- Cumbersome tuning + syncronization, but very flexible
- Basic structure for higher level interfaces
- Threads in Julia: ‘Threads.@spawn‘ (since Julia 1.3), marked as experimental

```
... 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