∼

Parallelization, OpenMP

Scientific Computing Winter 2016/2017

Lecture 26

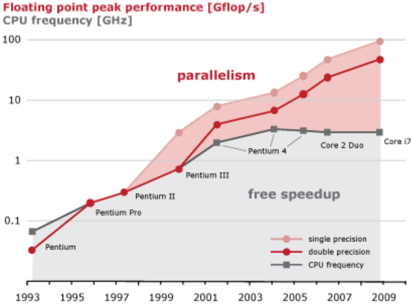
Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

made wit pandoc

Why parallelization ?

- Computers became faster and faster without that. . .

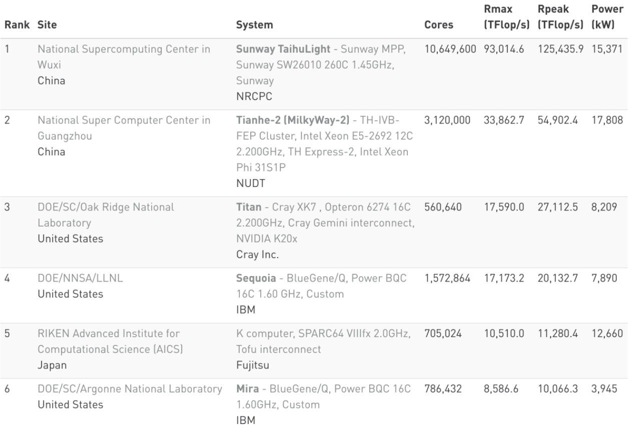


[Source: spiralgen.com]

- But: clock rate of processors limited due to physical limits
- ⇒ parallelization is the 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 2016 rank 1-6

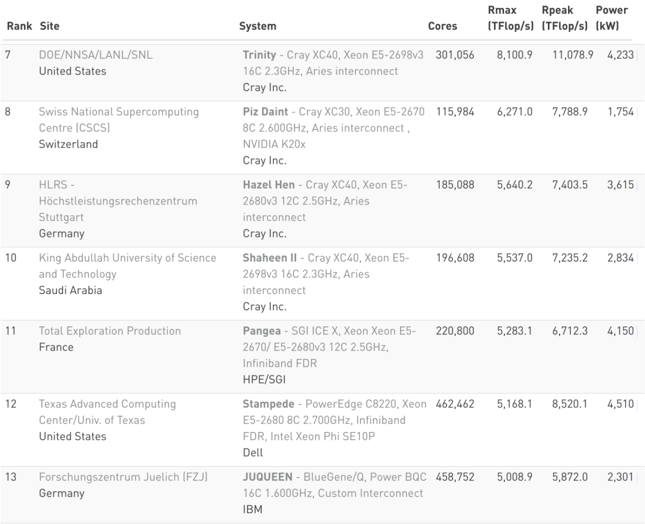
Based on linpack benchmark: solution of dense linear system. Typical desktop computer: $R_{max} \approx 100 \dots 1000 GFlop/s$



Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
2	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
3	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
4	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
5	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
6	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945

[Source:www.top500.org]

TOP 500 2016 rank 7-13

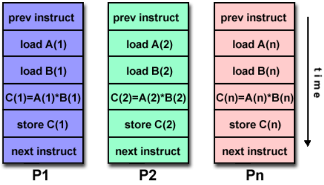


Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
7	DOE/NNSA/LANL/SNL United States	**Trinity** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.	301,056	8,100.9	11,078.9	4,233
8	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	1,754
9	HLRS - Höchstleistungsrechenzentrum Stuttgart Germany	**Hazel Hen** - Cray XC40, Xeon E5-2680v3 12C 2.5GHz, Aries interconnect Cray Inc.	185,088	5,640.2	7,403.5	3,615
10	King Abdullah University of Science and Technology Saudi Arabia	**Shaheen II** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.	196,608	5,537.0	7,235.2	2,834
11	Total Exploration Production France	**Pangea** - SGI ICE X, Xeon Xeon E5-2670/ E5-2680v3 12C 2.5GHz, Infiniband FDR HPE/SGI	220,800	5,283.1	6,712.3	4,150
12	Texas Advanced Computing Center/Univ. of Texas United States	**Stampede** - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462,462	5,168.1	8,520.1	4,510
13	Forschungszentrum Juelich (FZJ) Germany	**JUQUEEN** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	458,752	5,008.9	5,872.0	2,301

[Source:www.top500.org]

Parallel paradigms

SIMD
Single Instruction Multiple Data

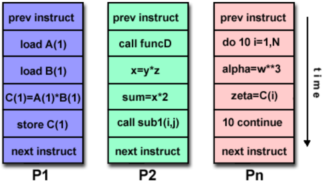


P1	P2	Pn
prev instruct	prev instruct	prev instruct
load A(1)	load A(2)	load A(n)
load B(1)	load B(2)	load B(n)
C(1)=A(1)*B(1)	C(2)=A(2)*B(2)	C(n)=A(n)*B(n)
store C(1)	store C(2)	store C(n)
next instruct	next instruct	next instruct

[Source: computing.llnl.gov/tutorials]

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

MIMD
Multiple Instruction Multiple Data

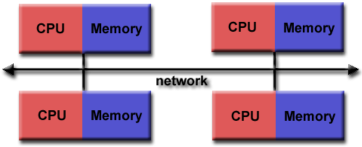


P1	P2	Pn
prev instruct	prev instruct	prev instruct
load A(1)	call funcD	do 10 i=1,N
load B(1)	x=y*z	alpha=w**3
C(1)=A(1)*B(1)	sum=x*2	zeta=C(i)
store C(1)	call sub1(i,j)	10 continue
next instruct	next instruct	next instruct

[Source: computing.llnl.gov/tutorials]

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

MIMD Hardware: Distributed memory



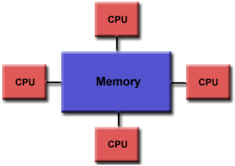
[Source: computing.llnl.gov/tutorials]

- “Linux Cluster”
- “Commodity Hardware”
- 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

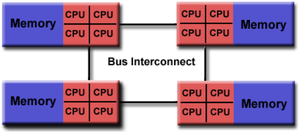
Symmetric Multiprocessing (SMP)/Uniform memory acces (UMA)



[Source: computing.llnl.gov/tutorials]

- Similar processors
- Similar memory access times

Nonuniform Memory Access (NUMA)



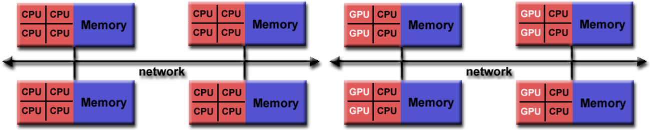
[Source: computing.llnl.gov/tutorials]

- 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 acces
- 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 both kinds of programming paradigms

Shared memory programming: pthreads

- Thread: lightweight process which can run parallel to others
- pthreads (POSIX threads): widely distributed
- cumbersome tuning + syncronization
- basic structure for more high level interfaces

```
#include <pthread.h>

void *PrintHello(void *threadid)
{  long tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{  pthread_t threads[NUM_THREADS];
   int rc;   long t;

   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc) {printf("ERROR; return code from pthread_create() is %d\n", rc); exit(-1);}
   }
   pthread_exit(NULL);
}
```

Source: computing.llnl.gov/tutorials

- compile and link with

```
gcc -pthread -o pthreads pthreads.c
```

Shared memory programming: C++11 threads

- Threads introduced into C++ standard with C++11
- Quite late... many codes already use other approaches
- But intersting for new applications

```
#include <iostream>
#include <thread>

void call_from_thread(int tid) {
    std::cout << "Launched by thread " << tid << std::endl;
}

int main() {
    std::thread t[num_threads];
    for (int i = 0; i < num_threads; ++i) {
        t[i] = std::thread(call_from_thread, i);
    }
    std::cout << "Launched from the main\n";
    //Join the threads with the main thread
    for (int i = 0; i < num_threads; ++i) {
        t[i].join();
    }
    return 0;
}
```

Source: https://solarianprogrammer.com/2011/12/16/cpp-11-thread-tutorial/

- compile and link with

```
g++ -std=c++11 -pthread cpp11threads.cxx -o cpp11threads
```

Thread programming: mutexes and locking

- If threads work with common data (write to the same memory address, use the same output channel) access must be syncronized
- Mutexes allow to define regions in a program which are accessed by all threads in a sequential manner.

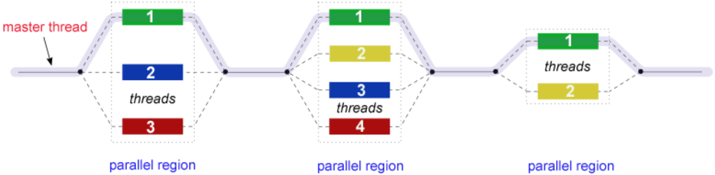
```cpp
#include <iostream>
#include <thread>
#include <mutex>
std::mutex mtx;
void call_from_thread(int tid) {
    mtx.lock()
    std::cout << "Launched by thread " << tid << std::endl;
    mtx.unlock()
}
int main() {
    std::thread t[num_threads];
    for (int i = 0; i < num_threads; ++i) {
        t[i] = std::thread(call_from_thread, i);
    }
    std::cout << "Launched from the main\n";
    //Join the threads with the main thread
    for (int i = 0; i < num_threads; ++i) {
        t[i].join();
    }
    return 0;
}
```

- *Barrier*: all threads use the same mutex for the same region
- *Deadlock*: two threads block each other by locking two different locks and waiting for each other to finish

Shared memory programming: OpenMP

- Mostly based on pthreads
- Available in C++,C,Fortran for all common compilers
- Compiler directives (pragmas) describe *parallel regions*

```
... sequential code ...
#pragma omp parallel
{
 ... parallel code ...
}
(implicit barrier)
... sequential code ...
```



[Source: computing.llnl.gov/tutorials]

Shared memory programming: OpenMP II

```cpp
#include <iostream>
#include <cstdlib>

void call_from_thread(int tid) {
  std::cout << "Launched by thread " << tid << std::endl;
}

int main (int argc, char *argv[])
{
  int num_threads=1;
  if (argc>1) num_threads=atoi(argv[1]);

#pragma omp parallel for
  for (int i = 0; i < num_threads; ++i)
  {
    call_from_thread(i);
  }
  return 0;
}
```

- compile and link with

```
g++ -fopenmp -o cppomp cppomp.cxx
```

Example: $u = au + v$ und $s = u \cdot v$

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

//implicit barrier
double s=0.0;
#pragma omp parallel for reduction(+:s)
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
- Write conflict with $+s$: several threads may access the same variable
- In standard situations, reduction variables can be used to avoid conflicts

Do it yourself reduction

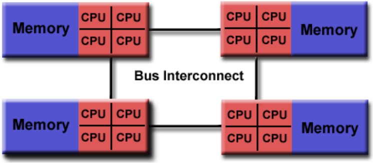
```
#include <omp.h>
int maxthreads=omp_get_max_threads();
double s0[maxthreads];
double u[n],v[n];
for (int ithread=0;ithread<maxthreads; ithread++)
  s0[ithread]=0.0;

#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: 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 (NUMA): first thread which “touches” data triggers the allocation of memory with the processeor where the thread is running on

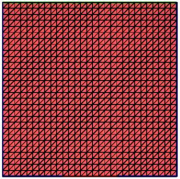
Parallelization of PDE solution

$$
\begin{aligned}
\Delta u &= f \text{ in}\Omega, \qquad\qquad\qquad\qquad u|_{\partial\Omega} = 0\\
\Rightarrow u &= \int_\Omega f(y)G(x,y)dy.
\end{aligned}
$$

- 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

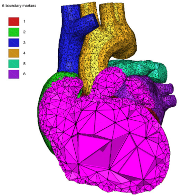
Structured and unstructured grids

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