High Performance Computing

Oliver Rheinbach

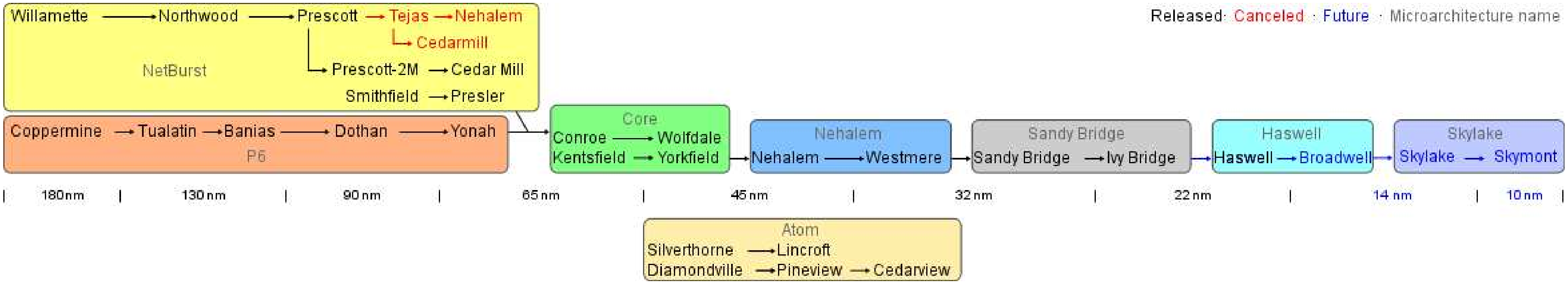
oliver.rheinbach@math.tu-freiberg.de

`http://www.mathe.tu-freiberg.de/nmo/`

Vorlesung Introduction to High Performance Computing

Übung Introduction to High Performance Computing

Intel Processor Roadmap



Released · Canceled · Future · Microarchitecture name

NetBurst
Willamette → Northwood → Prescott → Tejas → Nehalem
Cedarmill
Prescott-2M → Cedar Mill
Smithfield → Presler

P6
Coppermine → Tualatin → Banias → Dothan → Yonah

Core
Conroe → Wolfdale
Kentsfield → Yorkfield

Nehalem
Nehalem → Westmere

Sandy Bridge
Sandy Bridge → Ivy Bridge

Haswell
Haswell → Broadwell

Skylake
Skylake → Skymont

180nm | 130nm | 90nm | 65nm | 45nm | 32nm | 22nm | 14 nm | 10nm

Atom
Silverthorne → Lincroft
Diamondville → Pineview → Cedarview

1993: **486 processor**, die size 81mm^2, 1.2 Mio transistors (1000nm,600nm), 33 Mhz
1 flop per cycle, gives 33 Mflops peak
1999: **Pentium III (Coppermine)**, die size 140mm^2, 28 mio transistors, (250nm, 130nm), 450-1400 Mhz 2004: **Pentium 4 (Prescott)**, 122mm^2 (90nm), 2.8 Ghz - 3.4 Ghz
2009: **Core 2 Duo**, die size 111mm^2, 274 Mio transistors (45 nm), 3 Ghz
4 flop per cycle x 2 cores 24 Gflops peak; 727-fold increase from 486.
2010: **Corei7 (Bloomfield)**: 4 cores, 731 Mio transistors (45nm), 3.33 Ghz
2013: **Corei7 (Haswell)**: 4 cores, die size 177mm^2, 1400 Mio transistors (22nm), 2-3.5 Ghz
2014: **Xeon E7 (Ivy Bridge EX)**: 15 cores, die size 541mm^2, 4 310 Mio transistors (22nm), 2.3-2.8 Ghz
2015: **Broadwell (tick): 14nm** $\leq$3.7 Ghz
2016: **Skylake (tock): 14nm**: 4 cores die size 122.5mm^2
2017: **Kaby Lake (14nm)**
2018: **Coffee Lake (14nm)/Cannon Lake (10nm)**

2020: **Ice Lake (10nm)**

Theoretical Limits

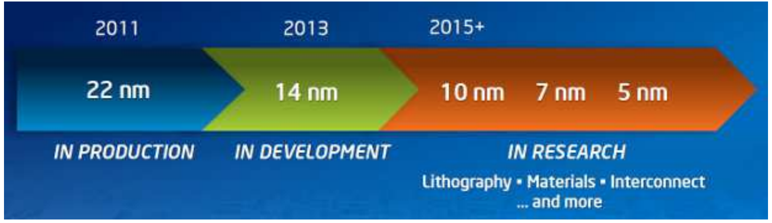
According to (a variant of) Moore's law the floating point performance of single processor cores has increased ten-fold every 5 years since the 1950ies. (Original Moore's law states doubling of number of transistors every 12-24 months). Today's processors cores are by a factor of $> 10^{10}$ faster than the ones from 1950ies.

There are theoretical limits to the performance of **synchronized** single processors.

Structure size: Today's processors have structures of 10 nm=$10 \cdot 10^{-9}$m (Cannon Lake 2018). The size of an atom is at the order of 10^{-10}m.

A factor of 100 (rather 100^2) to go!

Intel Roadmap



(2014: 14 nm Broadwell pushed to 2015)

Speed of light: $c = 3 \cdot 10^{10} cm/s$.
This gives a limit of $30/\sqrt{2}$ Ghz for a chip with diameter of 1 cm. **A factor of 10 to go!**

Current TOP500 List 11/2017

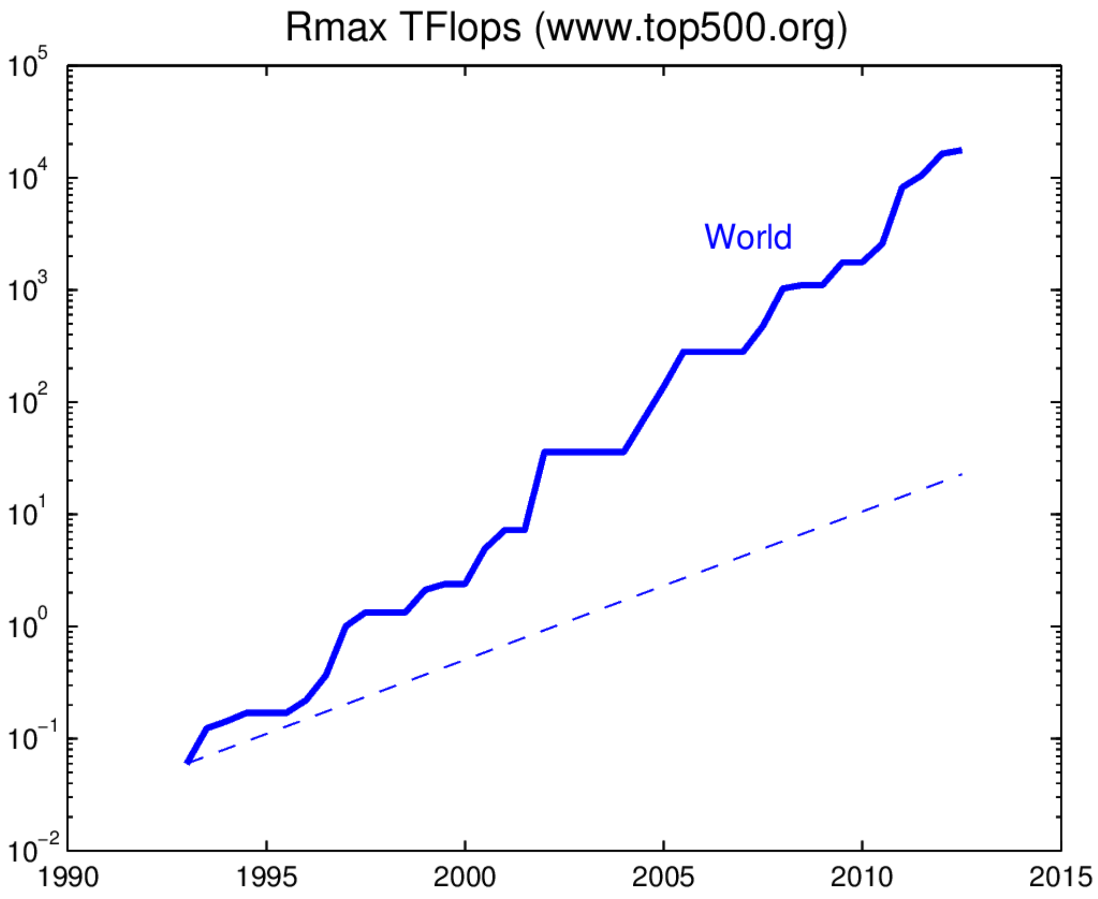
#Cores	Name	Institution
2 282 544	Summit	Oak Ridge, USA
10 649 600	TaihuLight	Nat. U of Defense Tech, China
1 572 480	Sierra	LLNL, USA
4 981 760	Tianhe-2A	Nat. U of Defense Tech, China
391 680	AI Bridging	AIST, Japan
361 760	Piz Daint	CSCS, Switzerland
560 540	Titan	Oak Ridge, USA
1 572 864	Sequoia	Lawrence Livermore, USA
979 968	Trinity	Los Alamos, USA
622 336	Cori	Lawrence Berkeley, USA
(23) **114 480**	Juwels Module 1	JSC, Germany
(27) **185 088**	Hazel Hen	HLRS, Germany
(28) **127 520**	Cobra	MPI, Germany
(57) **147 456**	SuperMUC	LRZ München

Factor of 1.5 million
from 1993 to 2018,
factor of 10 000 through more
factor 150 through performanc

The challenge for implicit methods are even higher than for other classes of parallel algorithms.

New approaches may be needed for exascale computing.

Supercomputers 1993 to 2013

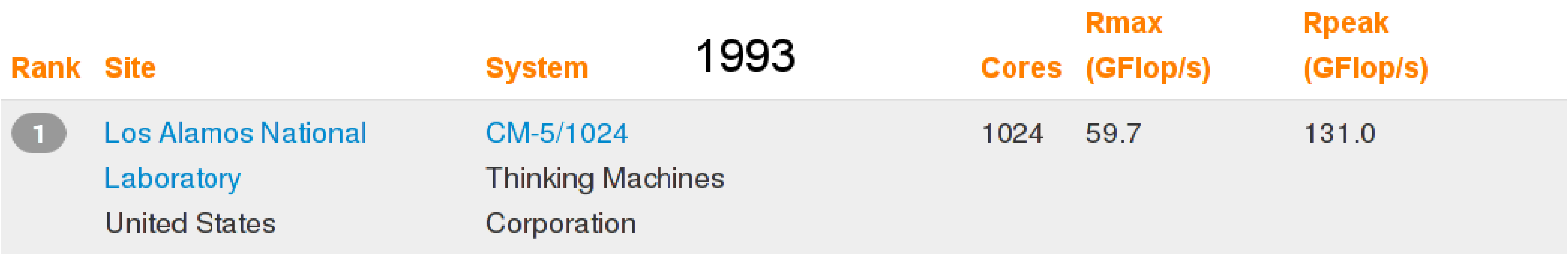


Faktor 3000 through more parallelism

High Performance Computing 1993 – 2013

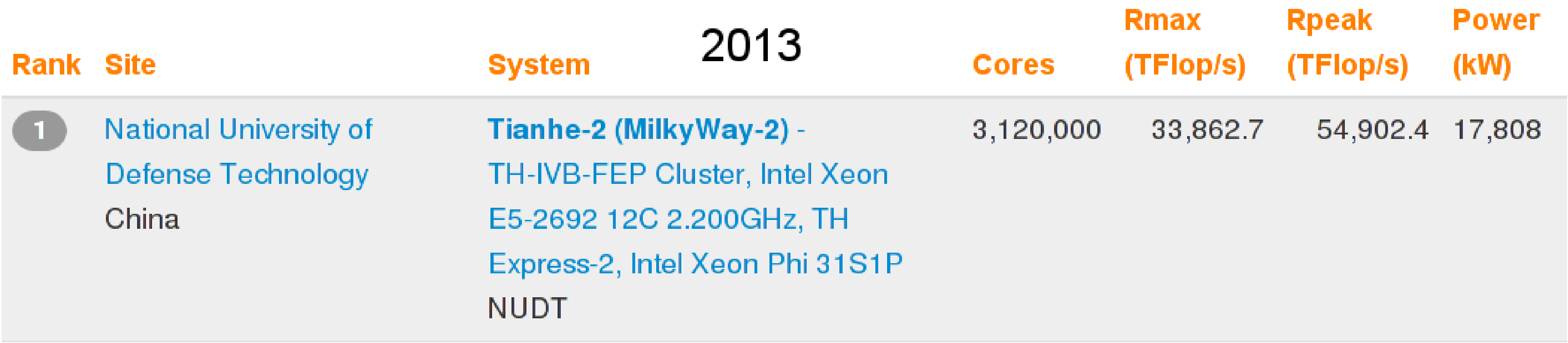
TOP500-List: `www.top500.org`

1993



Rank	Site	System	Cores	Rmax (GFlop/s)	Rpeak (GFlop/s)
1	Los Alamos National Laboratory United States	CM-5/1024 Thinking Machines Corporation	1024	59.7	131.0

2013



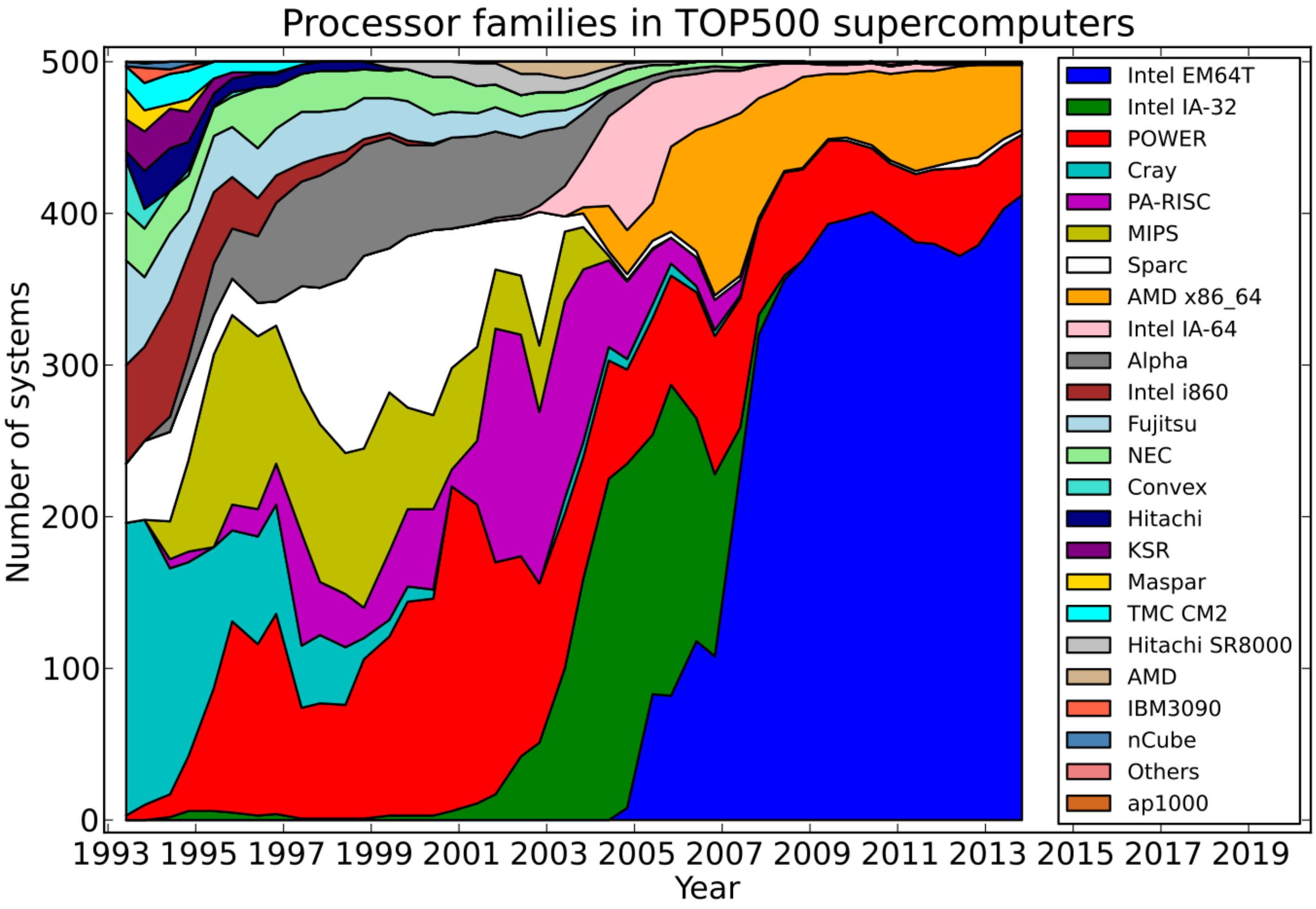
Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National University of Defense Technology China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808

A factor of 3000 in parallelism!
A factor of 600 000 in performance!

Hermit (HLRS Stuttgart)

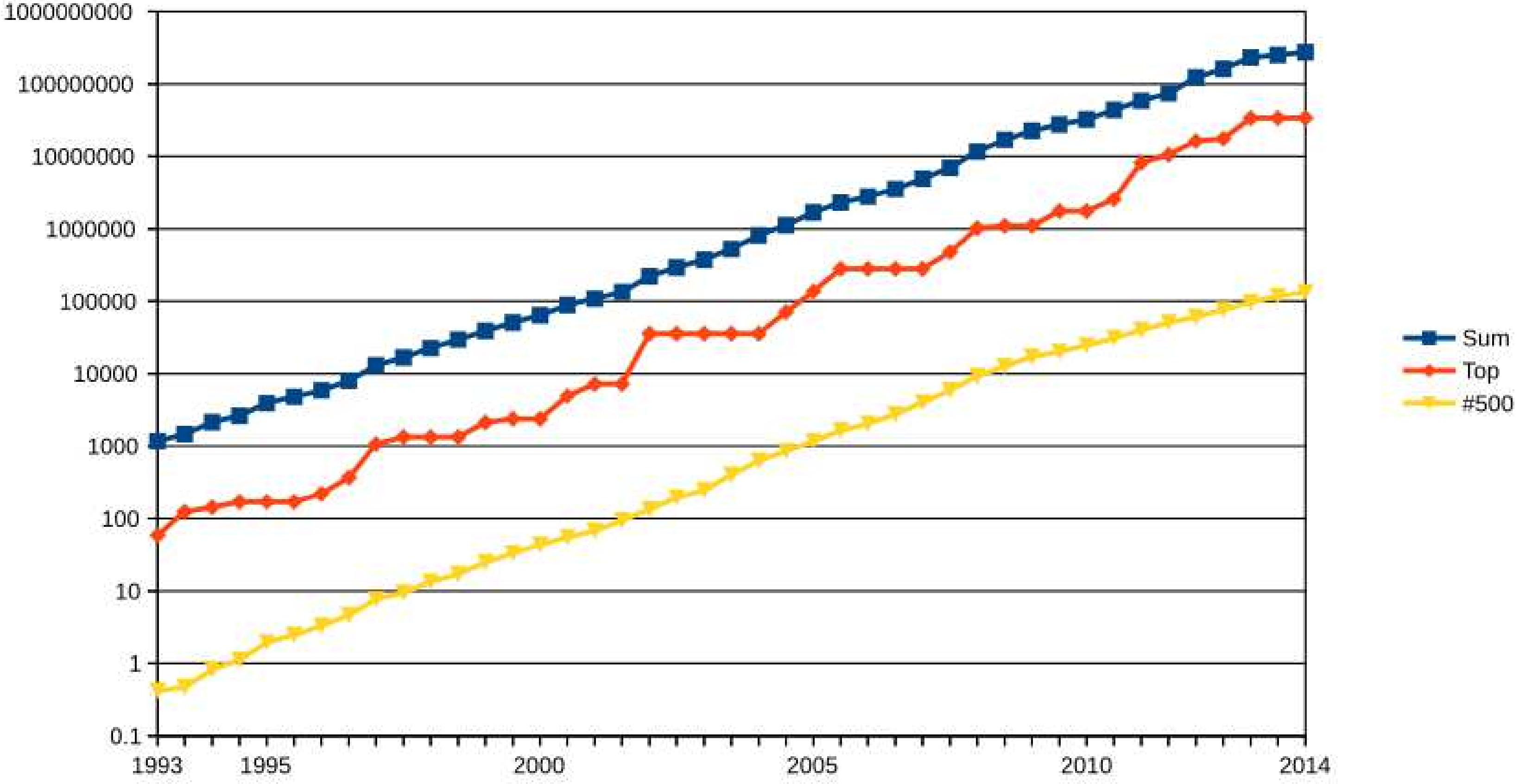
- Rank 12 in TOP500 of 11/2011; rank 34 in TOP500 of 06/2013.
- The processor-dies of the 7092 *Opteron-6276*-Processors cover an area of $> 0.7m^2$ of silicon. $\Rightarrow$ Max (synchronous) frequency $\approx$ 300Mhz.
- Actual frequency: 2.3Ghz.

TOP500 06/2014 Statistics



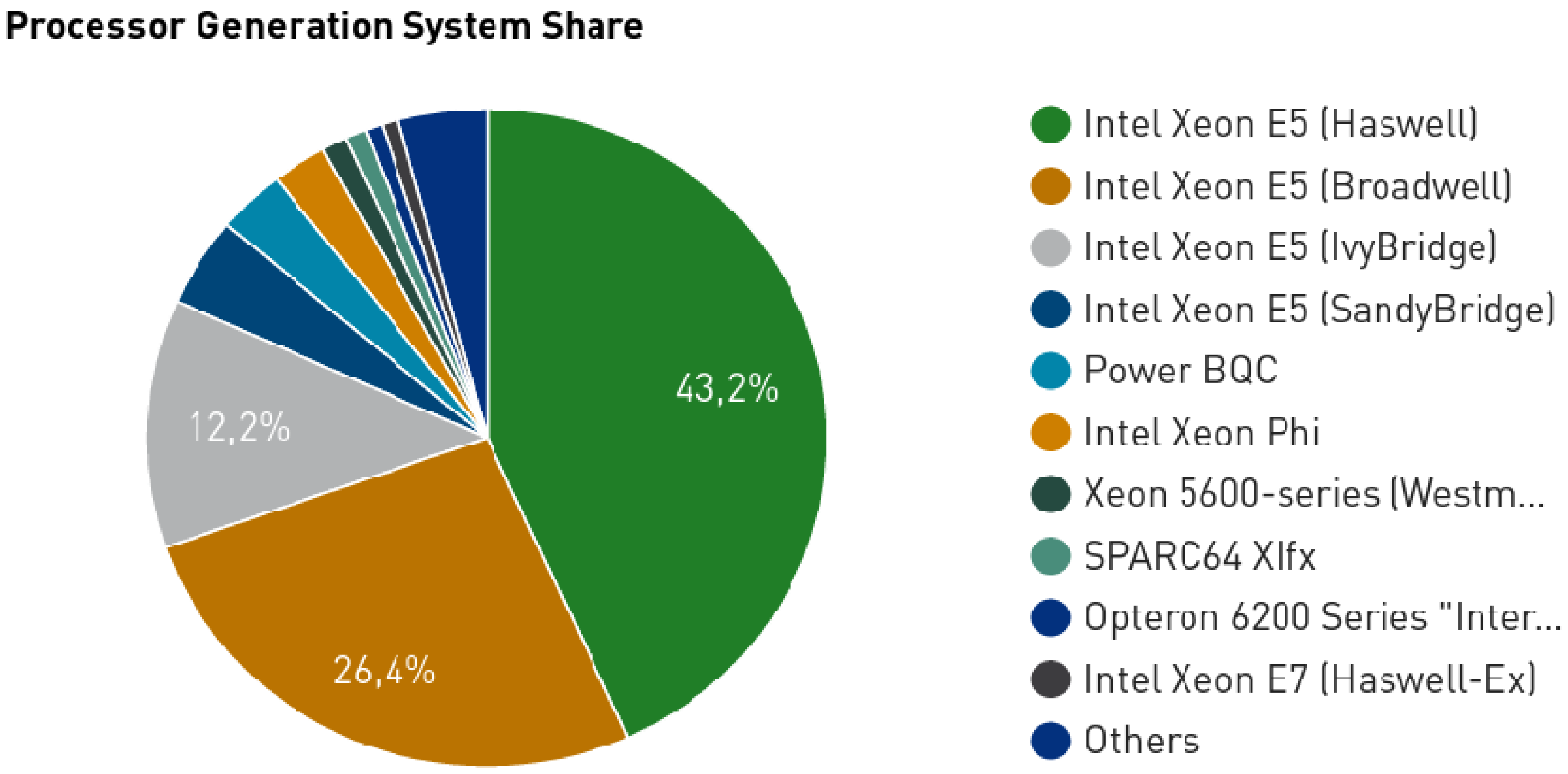
(Wikipedia)

TOP500 06/2014 Statistics



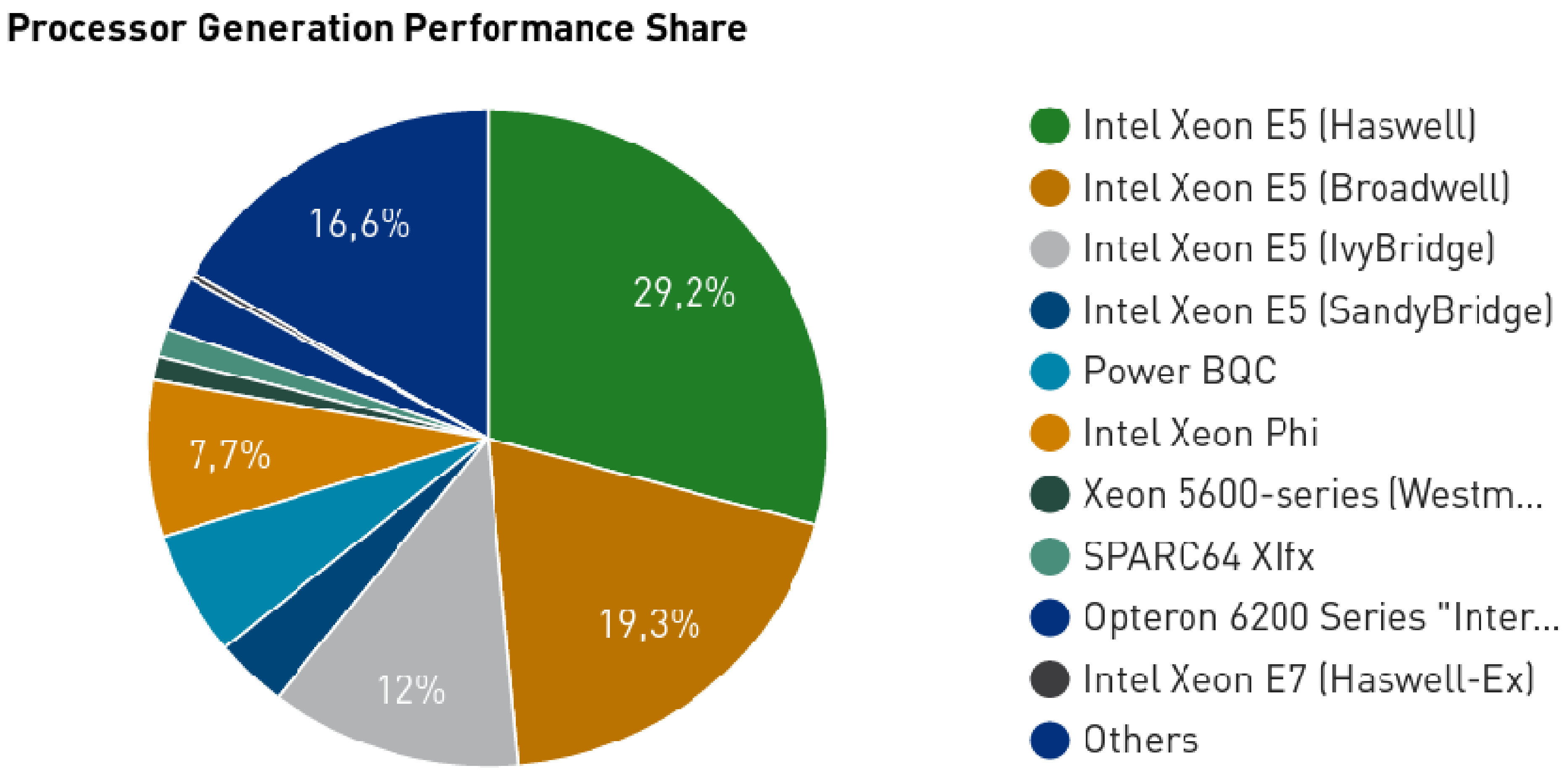
(Wikipedia)

TOP500 06/2017 Statistics



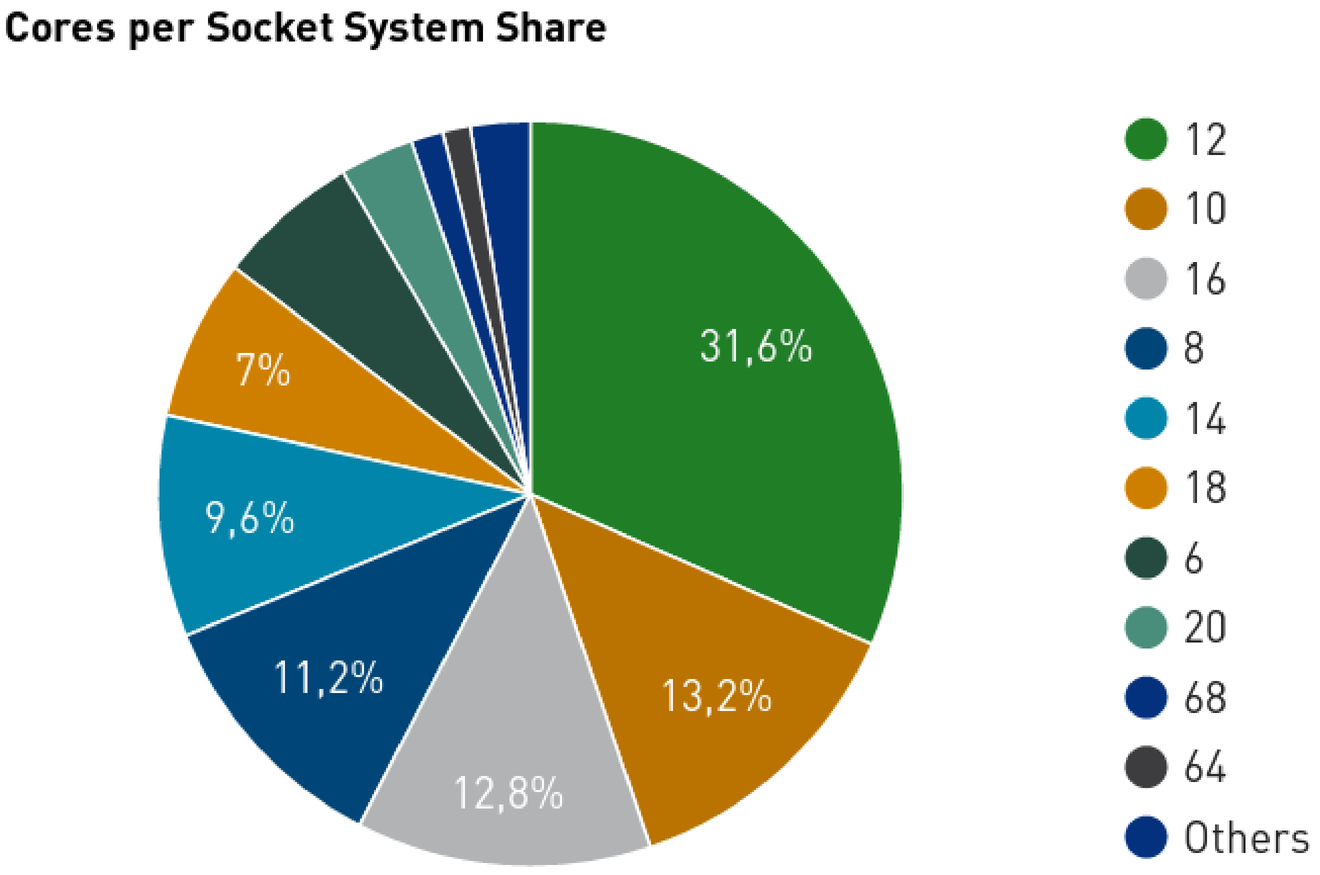
(www.top500.org)

TOP500 06/2017 Statistics



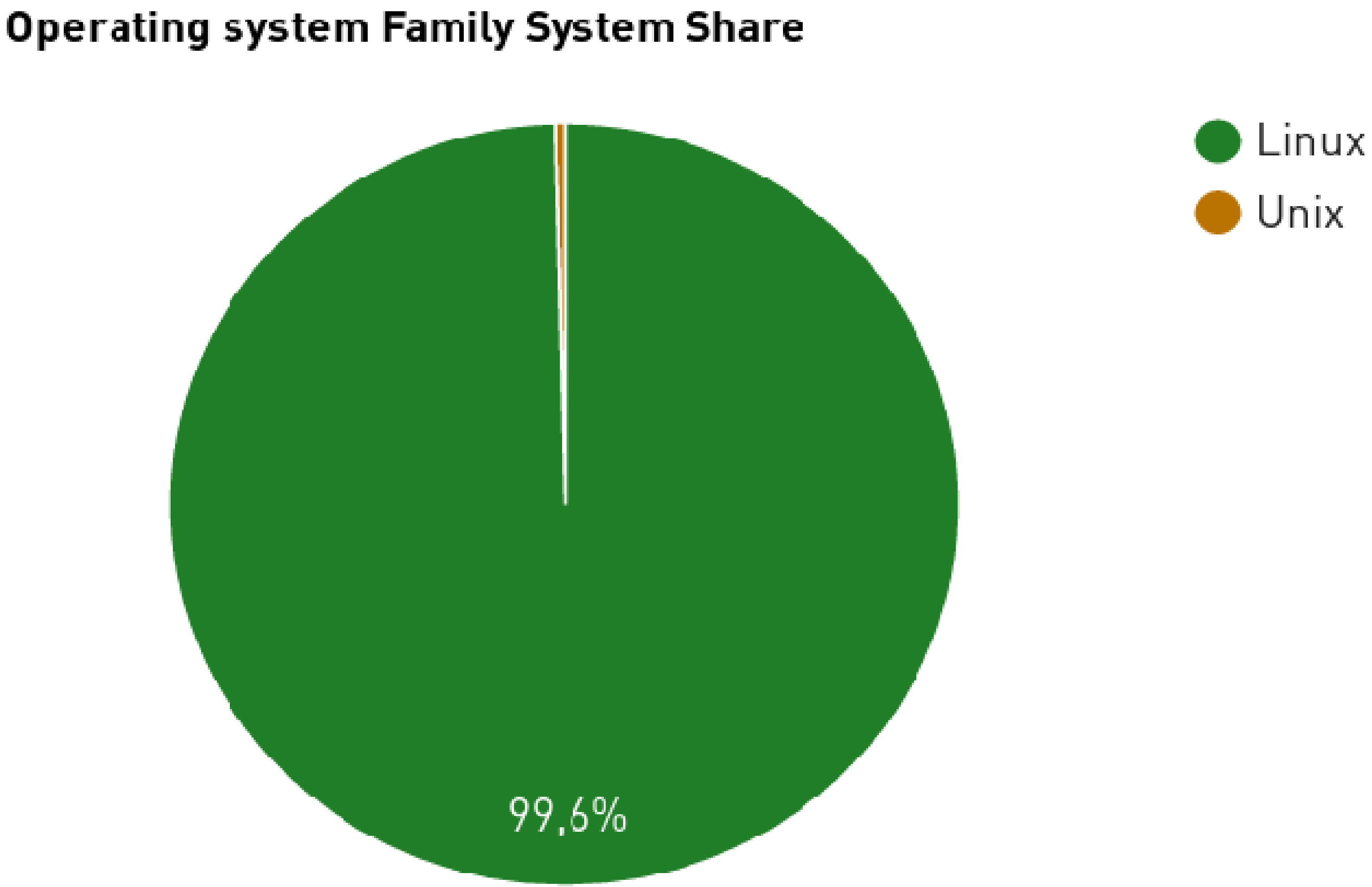
(www.top500.org)

TOP500 06/2017 Statistics

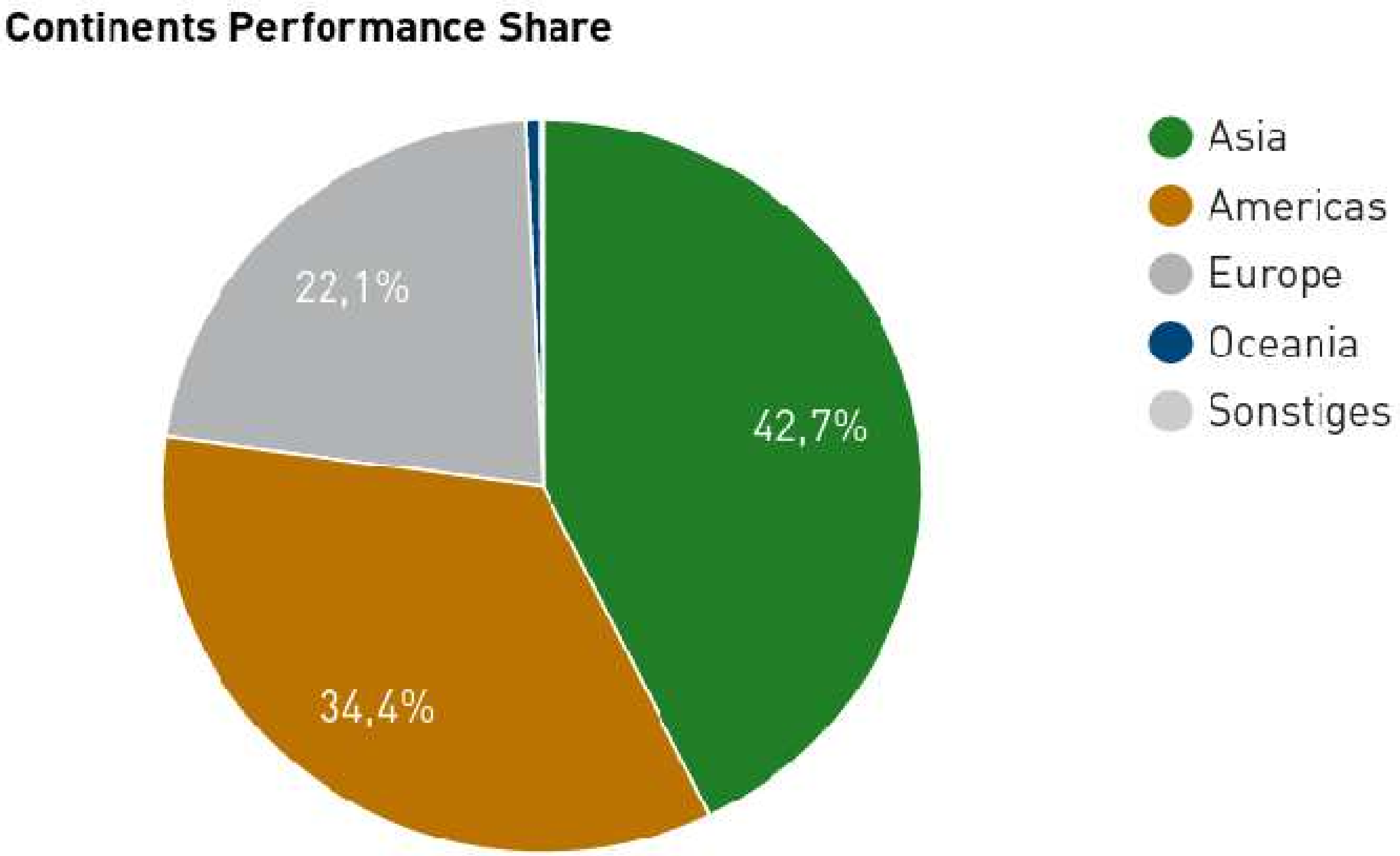


(www.top500.org)

TOP500 06/2017 Statistics

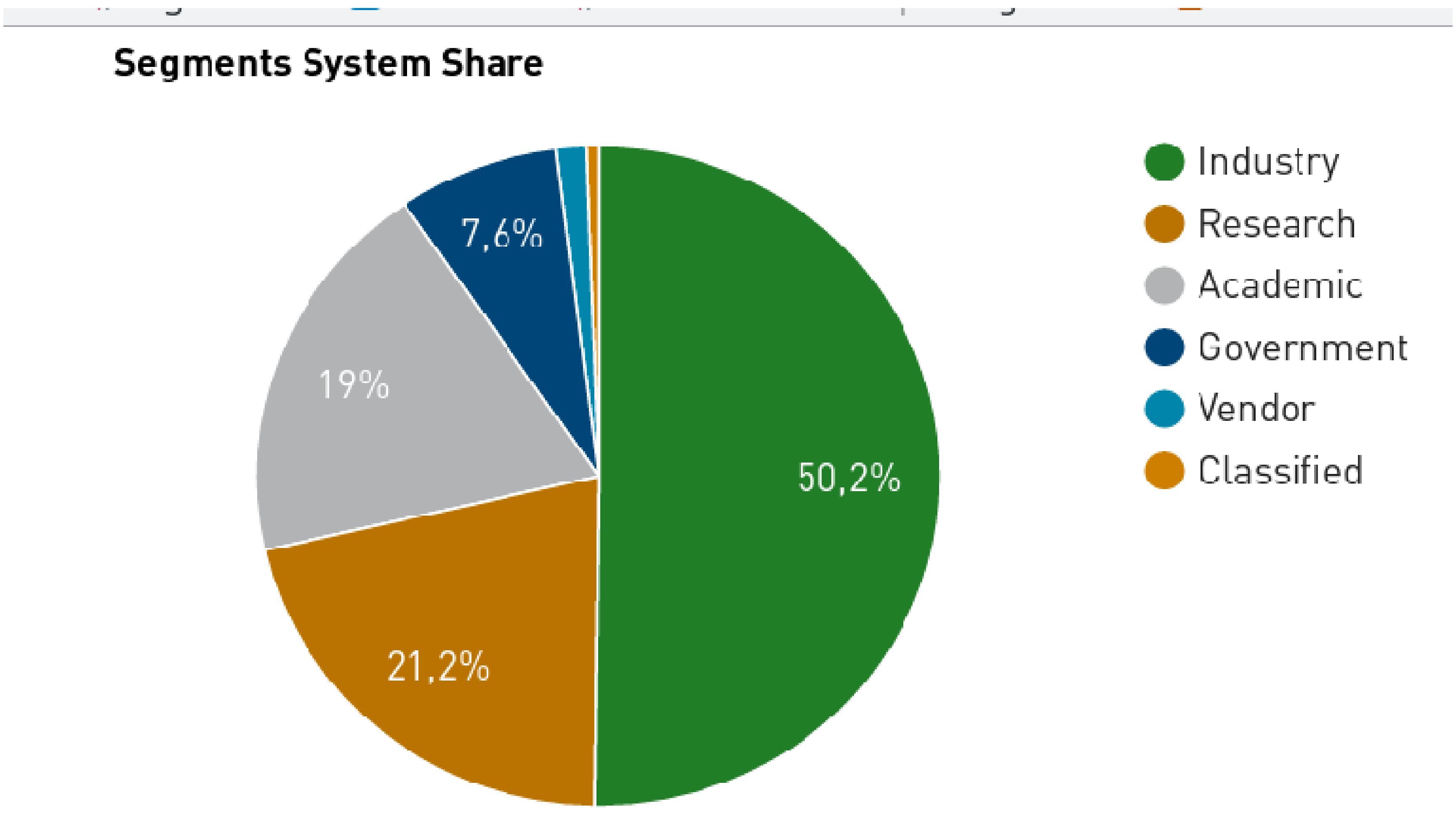


TOP500 06/2017 Statistics



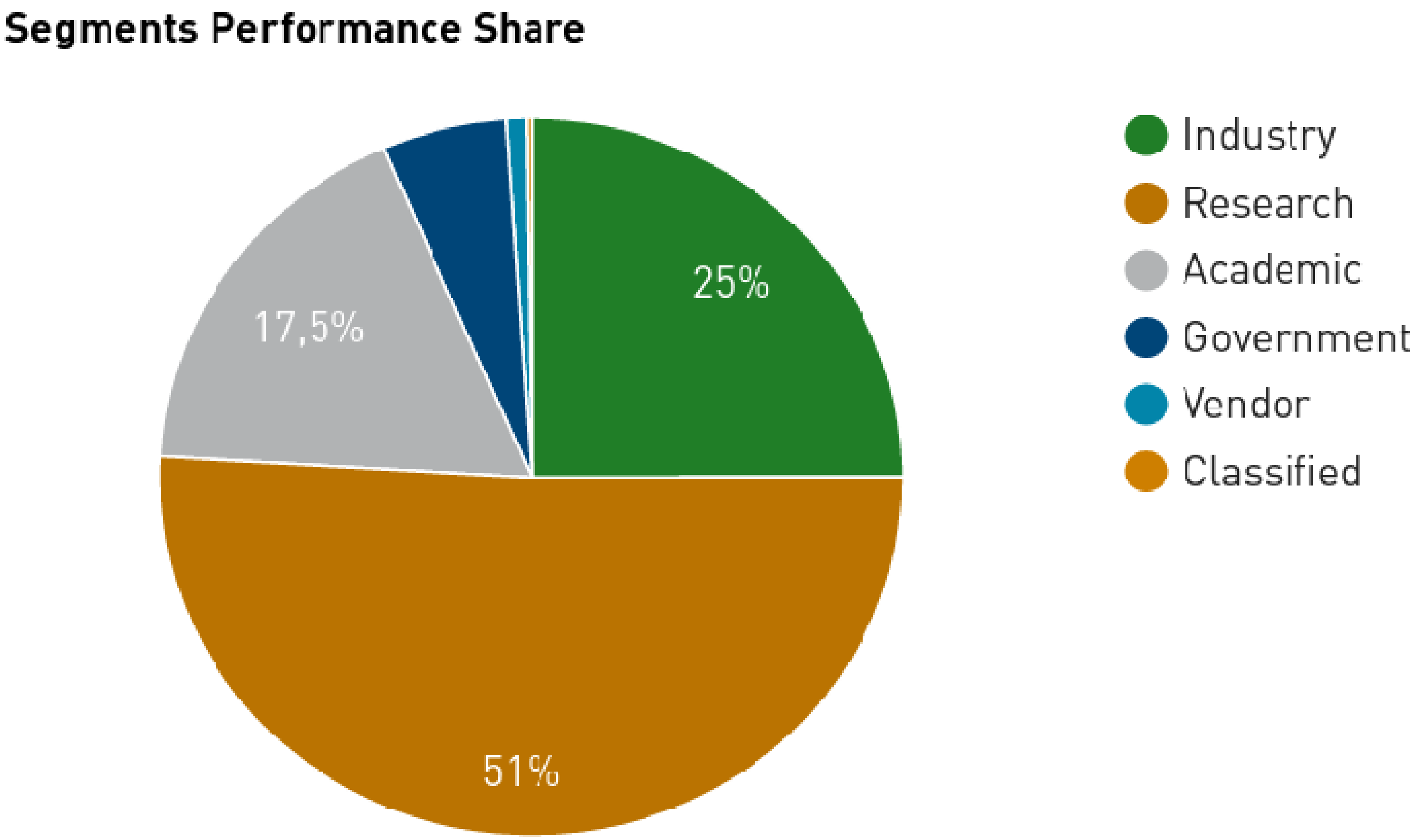
(www.top500.org)

TOP500 06/2017 Statistics



(www.top500.org)

TOP500 06/2017 Statistics



(www.top500.org)

OpenMP

See also www.openmp.org. OpenMP 1.0 is from 1998, MPI (MPI 1994). OpenMP is the de facto standard for shared memory applications in C, C++, Fortran whereas MPI is the de facto standard for distributed memory applications. OpenMP is portable and relatively simple, perfect for multicore (for the moment; new parallel languages will come up [see Juelich]). Note that MPI gives good performance for distributed AND shared memory machines. But it is more complicated.

OpenMP

- New compiler directives
- Runtime library
- Environmental variables

OpenMP uses thread-based parallelism. What is a thread?

- A thread is a lightweight process=an instance of the programm + data
- Each thread has its own ID $(0, \ldots, (N-1))$ and flow control
- Can shared data but also have private data
- All communication is done via the shared data

OpenMP Programming Modell

- All threads have access to the same **globally shared** memory
- Data can be shared or private (shared visible for all, private visible for owner only)
- Data transfer is transparent to the programmer
- Synchronization takes place but is mostly implicit

Fork and Join Model Master thread $\rightarrow$ (parallel region) Parallel threads $1, \ldots, n \rightarrow$ Synchronization

Today's compilers can do automatic parallelization of loops, BUT may fail

- because autoparallelization may not able to determine whether there is data dependance or not (situation is better in Fortran than in C).
- because granularity is not high enough: Compiler will parallelize on lowest level but should do so on the highest level. State of the art speedup using auto-parallelization is typical 4-8 at max.

Here, the programmer can help using explicit parallelization using OpenMP.

Some OpenMP directives

`omp parallel`

When a thread reaches a PARALLEL directive it creates a team of N threads and becomes the master of the team. The master is member of the team and has thread number 0.

The number of threads N is determined at runtime from `OMP_NUM_THREADS`; all threads execute the next statement or the next block if the statement is a $\{\ \}$-block. After the statement, the threads join back into one. There is an implied barrier at the end of the block.

C:

```
  #pragma omp parallel
  {
    // Code inside this region runs in parallel.
    printf("Hello!\n");
  }
```

Fortran:

```
!$OMP PARALLEL
       PRINT *,'Hello'
!$OMP END PARALLEL
```

omp parallel

```
#pragma omp parallel
{
  // Code inside this region runs in parallel.
  printf("Hello!\n");
}
```

This example prints the text “`Hello!`” followed by a newline as many times as there are threads in the team. For a quad-core system, it will output.

```
Hello!
Hello!
Hello!
Hello!
```

Note that there is overhead in creating and destroying a team. That is why it is often best to parallelize the outermost loop.

`omp parallel private`

- Variables (except for the automatic variables local to the block) are by default shared by the whole thread team.
- Sometimes this is not the desired behavior. In that case the PRIVATE clause can be used.
- All variables in its list are private to each thread, i.e. there exist N copies of them. The private variables are uninitialized!
- But see also later `firstprivate` and `lastprivate`.

omp parallel private (C)

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main ()
  {
  int nthreads, tid;
  /* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel private(nthreads, tid)
    {
    /* Obtain thread number */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
    /* Only master thread does this */
    if (tid == 0)
      {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
      }
    }  /* All threads join master thread and disband */
}
```

`omp parallel private` (Fortran)

```
      PROGRAM HELLO
      INTEGER NTHREADS, TID, OMP_GET_NUM_THREADS,
     +        OMP_GET_THREAD_NUM

C     Fork a team of threads giving them their own copies of variables
!$OMP PARALLEL PRIVATE(NTHREADS, TID)

C     Obtain thread number
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Hello World from thread = ', TID

C     Only master thread does this
      IF (TID .EQ. 0) THEN
        NTHREADS = OMP_GET_NUM_THREADS()
        PRINT *, 'Number of threads = ', NTHREADS
      END IF

C     All threads join master thread and disband
!$OMP END PARALLEL
      END
```

`omp parallel private` (Output)

Output:

```
Hello World from thread = 1
Hello World from thread = 3
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 2
```

`omp barrier`

- Threads can be synchronized by using an OpenMP barrier.
- The barrier clause causes threads encountering the barrier to wait until all the other threads in the same team have encountered the barrier.
- A barrier is implied after: `end parallel`, `end for`; `end sections`, `end single`.

`omp parallel for`

- The PARALLEL FOR directive splits the for-loop so that each thread in the current team handles a different portion of the loop.

- All portions are roughly of the same size.

- Note that (in C) some restrictions apply for the for-loop, e.g. the number of iterations of the loop must be known at runtime at the entry of the loop.

- There is an implicit **barrier** at the end of the loop.

- The loop variable is by default `private` whereas all others are `shared` by default. It is therefore not necessary (but maybe good style) to specify the loop variable in the `private` list.

Add vectors in parallel in C

```
rheinbach@quattro:~$ cat omp_add.c
#include <stdio.h>
#include <omp.h>

int main()
  {
  int n=8;
  int a[]={1,2,3,4,5,6,7,8};
  int b[]={1,2,3,4,5,6,7,8};
  int c[]={0,0,0,0,0,0,0,0};
  int i,num;
  int nthreads;

#pragma omp parallel for shared (n,a,b,c) private(i,num)
  for(i=0;i<n;i++)
    {
    c[i]=a[i]+b[i];
    }
#pragma omp parallel
    {
    num = omp_get_num_threads();
    printf("Num threads: %d\n",num);
    }

#pragma omp master
    {
    printf("\n");
    for(i=0;i<n;i++)
      {
      printf("%d ",c[i]);
      }
    }
  printf("\n");
}
```

How to Log in, Edit, Compile and Run

```
rheinbach@quattro:~$ g++ -fopenmp omp_add.c
rheinbach@quattro:~$ ./a.out
```

Output

```
Num threads: 4
Num threads: 4
Num threads: 4
Num threads: 4

2 4 6 8 10 12 14 16
```

Add vectors in parallel (FORTRAN)

In Fortran the directive is PARALLEL DO:

```
!$OMP PARALLEL DO
!$OMP& SHARED(A,B,C) PRIVATE(I)
      DO I = 1, N
         C(I) = A(I) + B(I)
      ENDDO
```

Changing the number of threads

The number of threads can be changed by setting the environment variable `OMP_NUM_THREADS`. A typical scalability example:

```
rheinbach@quattro:~$ export OMP_NUM_THREADS=1
rheinbach@quattro:~$ time ./a.out
real    0m7.022s
user    0m7.020s
sys     0m0.000s

rheinbach@quattro:~$ export OMP_NUM_THREADS=2
rheinbach@quattro:~$ time ./a.out
real    0m3.489s
user    0m6.136s
sys     0m0.000s

rheinbach@quattro:~$ export OMP_NUM_THREADS=4
rheinbach@quattro:~$ time ./a.out
real    0m1.913s
user    0m6.036s
sys     0m0.000s
```

omp parallel reduction

- The REDUCTION clause has the form `REDUCTION(op:variable,op:variable,...)`
- It performs a reduction operation on variables in its list using the given operator. They must be shared in the enclosing block.
- A private copy of the variables with the same name is created at entry of the block. At the end of the block the reduction operation is performed to the private variables and the result is written into a shared variable of the same name.
- If the result is automatically initialized to an appropriate value (see below). You may also initialize it to any other value that 0.

C:

```
#pragma omp parallel for private(i) reduction(+:result)
  for (i=0; i < n; i++)
    {
    result = result + (a[i] * b[i]);
    }
```

omp parallel reduction

FORTRAN:

```
!$OMP  PARALLEL DO PRIVATE(I)
!$OMP& REDUCTION(+:RESULT)

       DO I = 1, N
         RESULT = RESULT + (A(I) * B(I))
       ENDDO

!$OMP  END PARALLEL DO NOWAIT
```

The operator in the REDUCTION list may be

Operator	Initialization value
+ - | ^ ||	0
* &&	1
&	~0

How to do compile and run!

```
ssh login@login01.hrz.tu-freiberg.de
pico loop.c
gcc -fopenmp loop.c
export OMP_NUM_THREADS=4
time ./a.out
```

See also:

http://bisqwit.iki.fi/story/howto/openmp/

omp single and omp master

- The single clause specifies that the given block is executed by only one thread.
- It is unspecified which thread executes the block!
- Other threads skip the statement/block and wait at the implicit barrier at the end of the block.
- In the master clause the block is run by the master thread, and there is no implied barrier at the end: other threads skip the construct without waiting.

```
#pragma omp parallel
{
  ParallelPart1();
  #pragma omp single
  {
    SequentialPart();
  }
  ParallelPart2();
}
```

Race Conditions/Thread Safety

- If several threads write different values to a shared variable at the “same time” the result is undefined.
- Such a situation is referred to as race condition (“Wettlaufsituation”). Race conditions are simply incorrect parallel algorithms.
- Such a race condition arises if in the vector scalar product the `reduction` clause is omitted.
- The concurrent write to a variable can also be prevented by the `critical` clause. Note that the C++ STL is NOT THREAD SAVE. Concurrent access to containers is not allowed other than const access.

omp flush

After a write to a shared variable due to optimizations the result may still remain in a register. So a different thread will read the wrong value from the memory. A `flush` enforces that a consistent view of the memory at a point in the program.

- To ensure that both variables are flushed before the `if` clause a barrier would have to be added after the flush.

- `flush` is always needed after a write before a read by another thread but flush is implied after:

  - ⋆ `barrier`
  - ⋆ `critical`
  - ⋆ `end critical`
  - ⋆ `parallel`
  - ⋆ `end parallel`
  - ⋆ `end sections`
  - ⋆ `end single`

omp flush (Example)

In this example when reaching the flush directive the result will be written to the shared memory. Therefore in this example none of the threads will enter the if-block (if they flush at the same time) or one of them will (if it reaches the if-clause before the other thread flushes):

```
/* a=0 and b=0 */
/* Thread 1 */
b=1;
#pragma omp flush(a,b)
if(a==0)
{
...
}

/* Thread 2 */
#pragma omp flush(a,b)
if(b==0)
{
...
}
```

Memory Cells and Registers

- Variables can be stored in “**memory cells**”, i.e., they reside in the main memory, i.e., access is **slow** and typically uses caches.

- Variables can also reside in internal memory of processors called “registers”, i.e., access is then immediate. But the number of registers is very limited, e.g., **16 registers** in your x86-64/AMD64 computer.

- *Remark: C has the (ancient) keyword* `register` *which is recommendation to the compiler to store the variable in a register for fast access, e.g., as loop index. Today obsolete since compilers will optimize for themselves and may even ignore* `register`*. W.r.t. language they will still honor that the adress & of a register variable cannot be taken.*