Advanced Computer Architecture

Chapter 10 – Multicore, parallel, and cache coherency

Part1:

Power, multicore, the end of the free lunch, and how to program a parallel computer

Shared-memory versus distributed-memory

November 2025

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (3rd, 4th and 5th eds),* and on the lecture slides of David Patterson, John Kubiatowicz and Yujia Jin at Berkeley

What you should get from this

Parallel systems architecture is a vast topic, and we can only scratch the surface. The critical things I hope you will learn from this very brief introduction are:

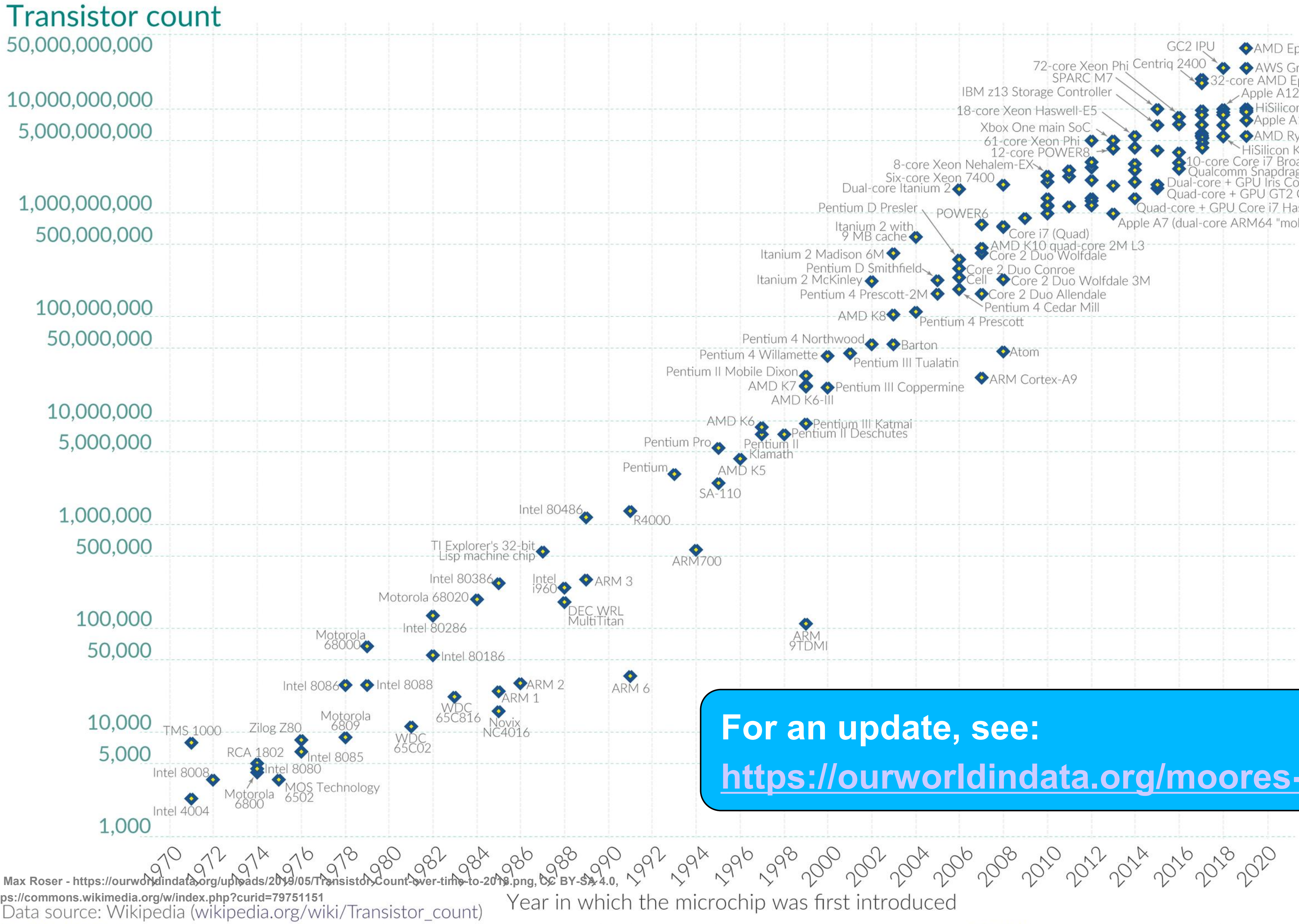
Part 1

- Why power considerations motivate multicore
- Why is shared-memory parallel programming attractive?
  - How is dynamic load-balancing implemented?
  - Why is distributed-memory parallel programming harder but more likely to yield robust performance?
- What is the cache coherency problem
  - There is a design-space of “snooping” protocols based on broadcasting invalidations and requests
- How are atomic operations and locks implemented?
  - Eg load-linked, store conditional
- What is sequential consistency?
  - Why might you prefer a memory model with weaker consistency?
- For larger systems, some kind of “directory” is needed to avoid/reduce the broadcasting

Moore's Law: The number of transistors on microchips doubles every two years

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.



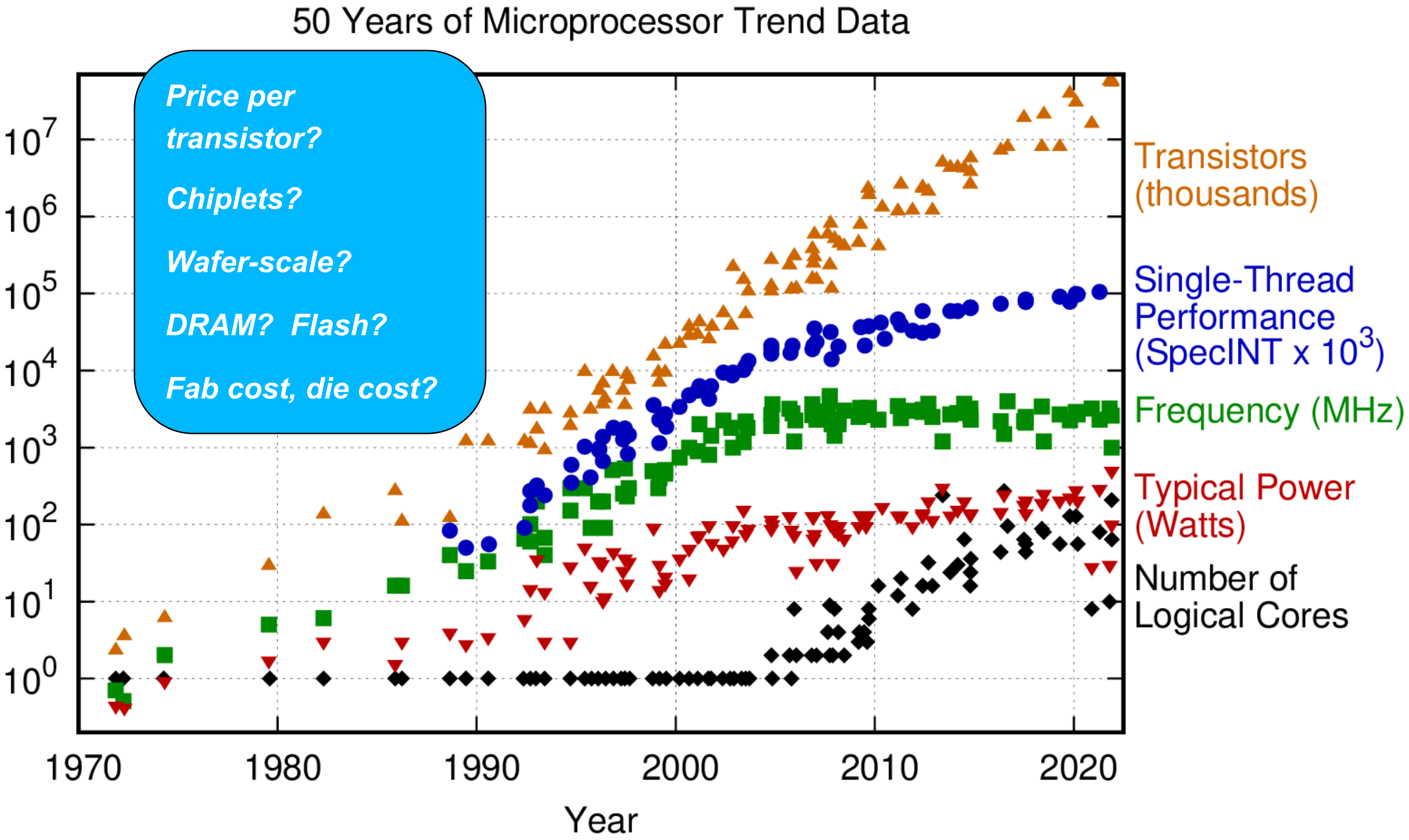
For an update, see: https://ourworldindata.org/moores-law

By Max Roser - https://ourworldindata.org/uploads/2019/05/Transistor-Count-over-time-to-2018.png, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=79751151

Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world's largest problems.

https://en.wikipedia.org/wiki/Transistor_count#/media/File:Moore's_Law_Transistor_Count_1970-2020.png



Price per transistor?

Chiplets?

Wafer-scale?

DRAM? Flash?

Fab cost, die cost?

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

https://raw.githubusercontent.com/karlrupp/microprocessor-trend-data/master/50yrs/50-years-processor-trend.png

You see this graph kind of *everywhere*. You might wonder whether this is the whole story….

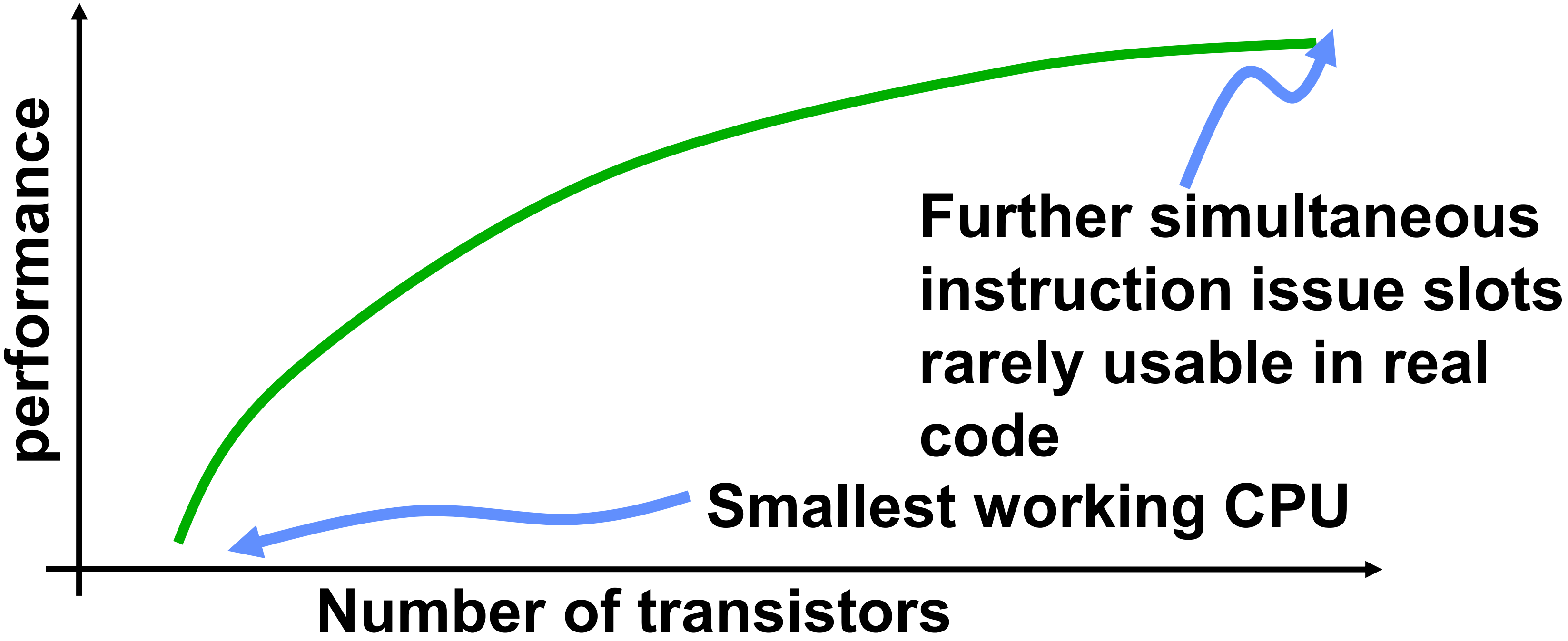
Power is the critical constraint

- Dynamic power vs static leakage
  - Dynamic: Power is consumed when signals change
  - Static: Power is consumed when gates are powered-up
  - "Dennard Scaling": dynamic power gets smaller if we make the transistors smaller
  - "the end of Dennard Scaling": static leakage starts to dominate, especially at high voltage (that is needed for high clock rate)
- Power vs clock rate
  - Power increases sharply with clock rate because
    - High static leakage due to high voltage
    - High dynamic switching
- Clock vs parallelism: *much* more efficient to use
    - Lots of parallel units, low clock rate, at low voltage

- **What can we do about power?**
- Compute fast then turn it off! (“race-to-sleep”)
- Compute just fast enough to meet deadline

- Clock gating, power gating
  - Turn units off when they’re not being used
  - Functional units
  - Whole cores...
- Dynamic voltage, clock regulation
  - Reduce clock rate dynamically
  - Reduce supply voltage as well
  - Eg when battery is low
  - Eg when CPU is not the bottleneck (*why?*)
- Run on lots of cores, each running at a slow clock rate
- Turbo mode
  - Boost clock rate when only one core is active

Why add another processor?



Further simultaneous instruction issue slots rarely usable in real code

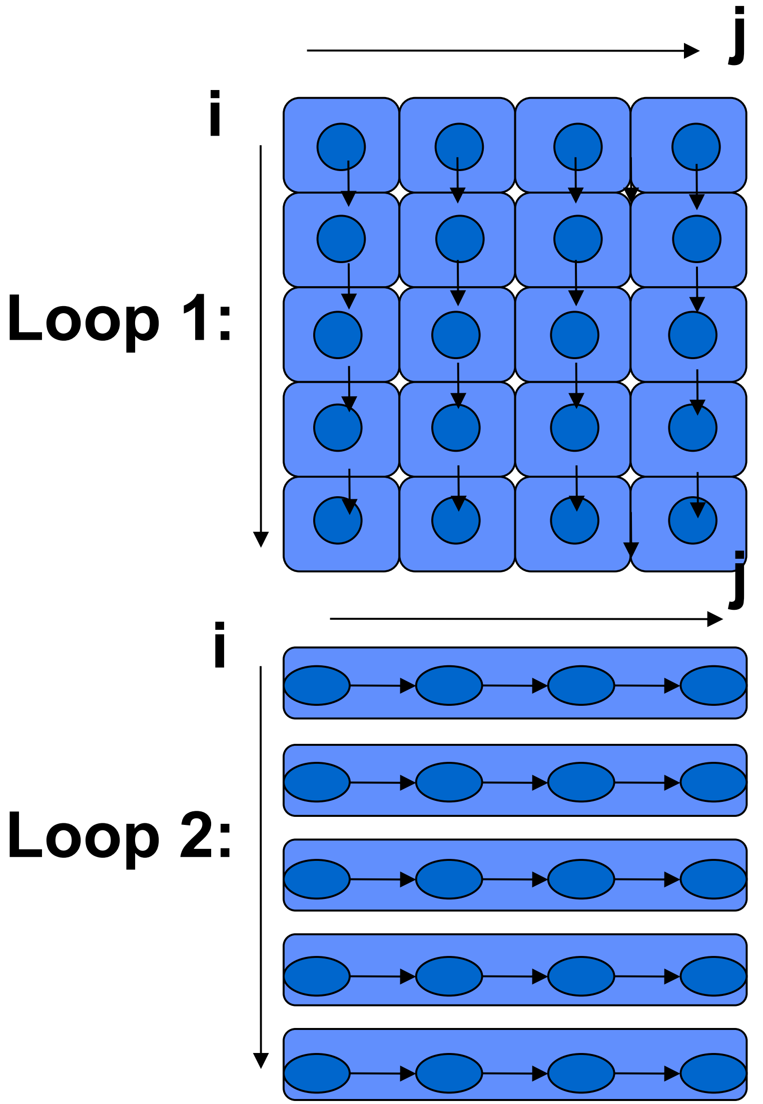
Smallest working CPU

- Increasing the complexity of a single CPU leads to diminishing returns
  - Due to lack of instruction-level parallelism
  - Too many simultaneous accesses to one register file
  - Forwarding wires between functional units too long - inter-cluster communication takes >1 cycle
    - Pollack’s rule, “performance scales as the square root of design complexity”
- More clock rate = *much* more power
- We can often do better with more cores running at a lower clock rate

How to program a parallel computer?

- Shared memory makes parallel programming much easier:

```
for(i=0; I<N; ++i)
  par_for(j=0; j<M; ++j)
   A[i,j] = (A[i-1,j] + A[i,j])*0.5;
par_for(i=0; I<N; ++i)
  for(j=0; j<M; ++j)
   B[i,j] = (A[i,j-1] + A[i,j])*0.5;
```



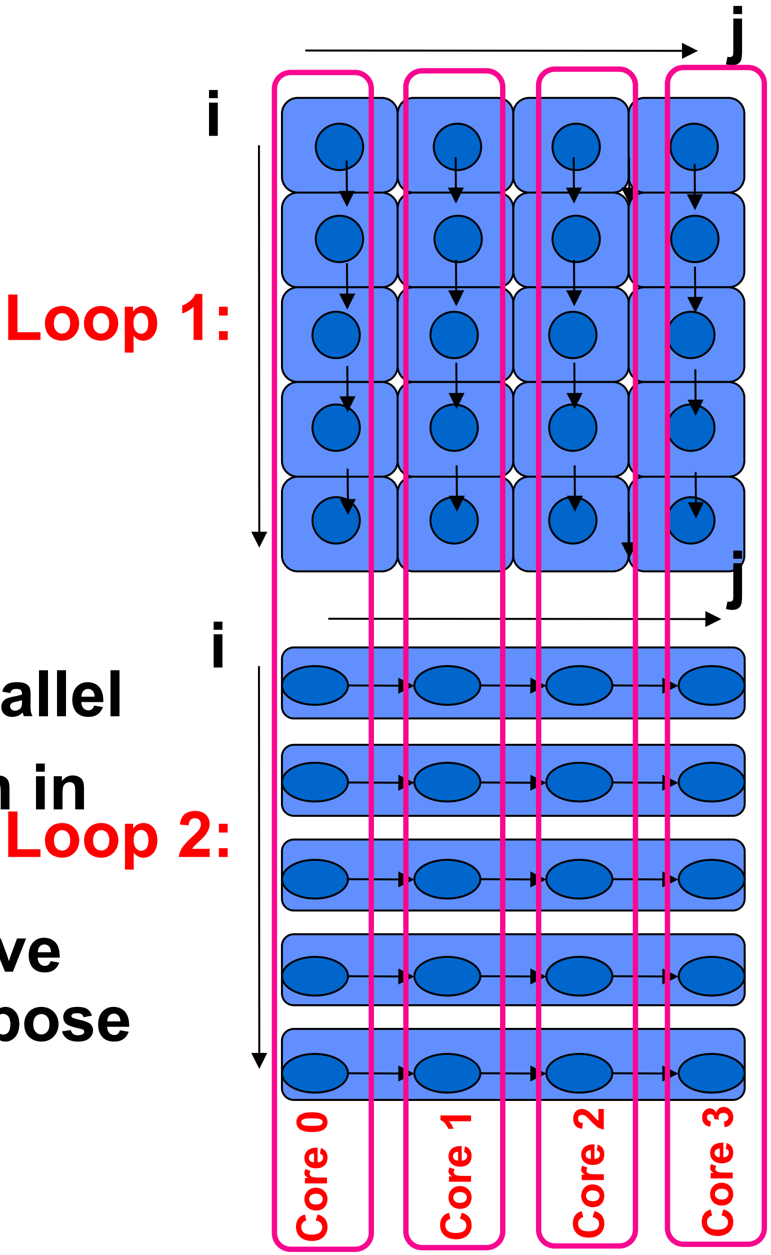
- First loop operates on rows in parallel
- Second loop operates on columns in parallel
- With distributed memory we would have to program message passing to transpose the array in between
- With shared memory… no problem!

How to program a parallel computer?

- Shared memory makes parallel programming much easier:

```
    for(i=0; I<N; ++i)
      par_for(j=0; j<M; ++j)
        A[i,j] = (A[i-1,j] + A[i,j])*0.5;
    par_for(i=0; I<N; ++i)
      for(j=0; j<M; ++j)
        B[i,j] = (A[i,j-1] + A[i,j])*0.5;
```

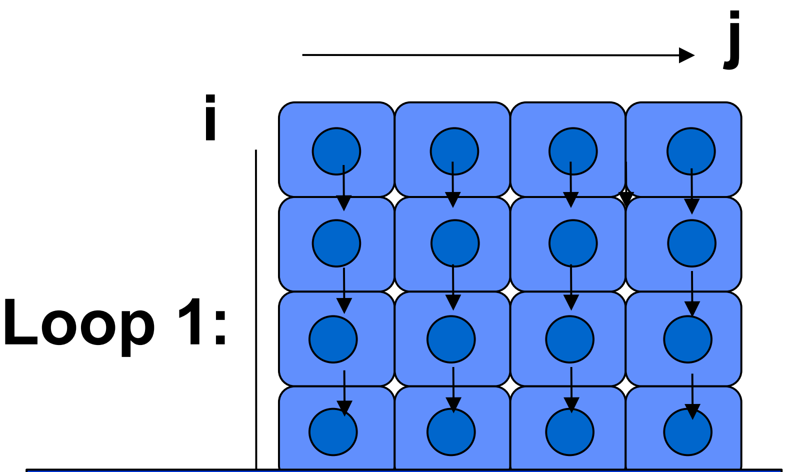
- First loop operates on each row in parallel
- Second loop operates on each column in parallel
- With distributed memory we would have to program message passing to transpose the array in between
- With shared memory… no problem!



How to program a parallel computer?

- Shared memory makes parallel programming much easier:

```
for(i=0; I<N; ++i)
  par_for(j=0; j<M; ++j)
    A[i,j] = (A[i-1,j] + A[i,j])*0.5;
par_for(i=0; I<N; ++i)
  for(j=0; j<M; ++j)
    B[i,j] = (A[i,j-1] + A[i,j])*0.5;
```



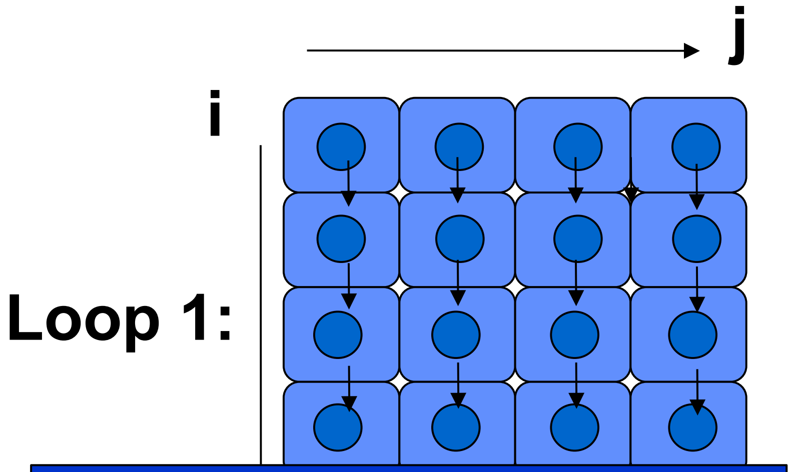
- First loop operates on rows in parallel
- Second loop operates on columns in parallel
- With distributed memory we would have to program message passing to transpose the array in between
- With shared memory… no problem!

- Shared memory is *convenient*
- Shared memory is *fast* – communicate with just a load/store

How to program a parallel computer?

- Shared memory makes parallel programming much easier:

```
    for(i=0; I<N; ++i)
      par_for(j=0; j<M; ++j)
        A[i,j] = (A[i-1,j] + A[i,j])*0.5;
    par_for(i=0; I<N; ++i)
      for(j=0; j<M; ++j)
        B[i,j] = (A[i,j-1] + A[i,j])*0.5;
```



- First loop operates on rows in parallel
- Second loop operates on columns in parallel
- With distributed memory we would have to program message passing to transpose the array in between
- With shared memory… no problem!

- Shared memory is *convenient*
- Shared memory is *fast* – communicate with just a load/store
- Shared memory is a *trap*!

- Because it encourages programmers to ignore where the communication is happening

Shared-memory parallel - OpenMP

- OpenMP is a standard design for language extensions for shared-memory parallel programming
- Language bindings exist for Fortran, C, C++ and to some extent (eg research prototypes) for Java and C#
- Implementation requires compiler support – as found in GCC, clang/llvm, Intel's compilers, Microsoft Visual Studio, Apple Xcode
- Example:

```
for(i=0; I<N; ++i)
   #pragma omp parallel for
   for(j=0; j<M; ++j)
     A[i,j] = (A[i-1,j] + A[i,j])*0.5;
#pragma omp parallel for
for(i=0; I<N; ++i)
   for(j=0; j<M; ++j)
     A[i,j] = (A[i,j-1] + A[i,j])*0.5;
```

(OpenMP is just one tool for shared-memory parallel programming – there are many more, but it exposes the most important issues)

Implementing shared-memory parallel loop

```
for (i=0; i<N; i++) {
  C[i]  = A[i] + B[i];
}
```

```
if (myThreadId() == 0)
  i = 0;
barrier();
// on each thread
while (true) {
  local_i = FetchAndAdd(&i);
  if (local_i >= N) break;
  C[local_i] = A[local_i] + B[local_i];
}
barrier();
```

Barrier(): block until all threads reach this point

- "self-scheduling" loop
- FetchAndAdd() is atomic operation to get next un-executed loop iteration:

```
Int FetchAndAdd(int *i) {
  lock(i);
  r = *i;
  *i = *i+1;
  unlock(i);
  return(r);
}
```

Optimisations:

- **Work in chunks**
- **Avoid unnecessary barriers**
- **Exploit "cache affinity" from loop to loop**

There are smarter ways to implement FetchAndAdd….

Implementing Fetch-and-add

We could use locks:

```
Int FetchAndAdd(int *i) {
  lock(i);
  r = *i;
  *i = *i+1;
  unlock(i);
  return(r);
}
```

- Using locks is rather expensive (and we should discuss how they would be implemented)
- But on many processors there is support for atomic increment
- So use the GCC built-in:

  `__sync_fetch_and_add(p, inc)`

- Eg on x86 this is implemented using the “exchange and add” instruction in combination with the “lock” prefix:

  `LOCK XADDL r1 r2`

- The “lock” prefix ensures the exchange and increment are executed on a cache line which is held exclusively

Combining:

- In a large system, using FetchAndAdd() for parallel loops will lead to contention
- But FetchAndAdds can be combined in the *network*
- When two FetchAndAdd(p,1) messages meet, combine them into one FetchAndAdd(p,2) – and when it returns, pass the two values back.

More OpenMP

```
#pragma omp parallel for \
   default(shared) private(i) \
   schedule(static,chunk) \
   reduction(+:result)
for (i=0; i < n; i++)
   result = result + (a[i] * b[i]);
```

- **default(shared) private(i):**

  All variables except i are shared by all threads.

- **schedule(static,chunk):**

  Iterations of the parallel loop will be distributed in equal sized blocks to each thread in the “team”

- **reduction(+:result):**

  performs a reduction on the variables that appear in its argument list

  - **A private copy for each variable is created for each thread. At the end of the reduction, the reduction operator is applied to all private copies of the shared variable, and the final result is written to the global shared variable.**

http://www.llnl.gov/computing/tutorials/openMP/#REDUCTION

Distributed-memory parallel - MPI

- MPI (“Message-passing Interface) is a standard API for parallel programming using message passing
- Six basic calls:
  - MPI_Init - Initialize MPI
  - MPI_Comm_size - Find out how many processes there are
  - MPI_Comm_rank - Find out which process I am
  - MPI_Send - Send a message
  - MPI_Recv - Receive a message
  - MPI_Finalize - Terminate MPI

(MPI is just one tool for distributed-memory parallel programming – there are many more, but it exposes the most important issues)

- Key idea: collective operations
  - MPI_Bcast - broadcast data from the process with rank "root" to all other processes of the group
  - MPI_Reduce – combine values on all processes into a single value using the operation defined by the parameter op (eg sum)
  - MPI_AllReduce – MPI_Reduce and then broadcast so every process has the sum
- Essential advice: Single-Program, Multiple Data (SPMD)
  - Each process has a share of the data,
  - Every process *shares the same control-flow*

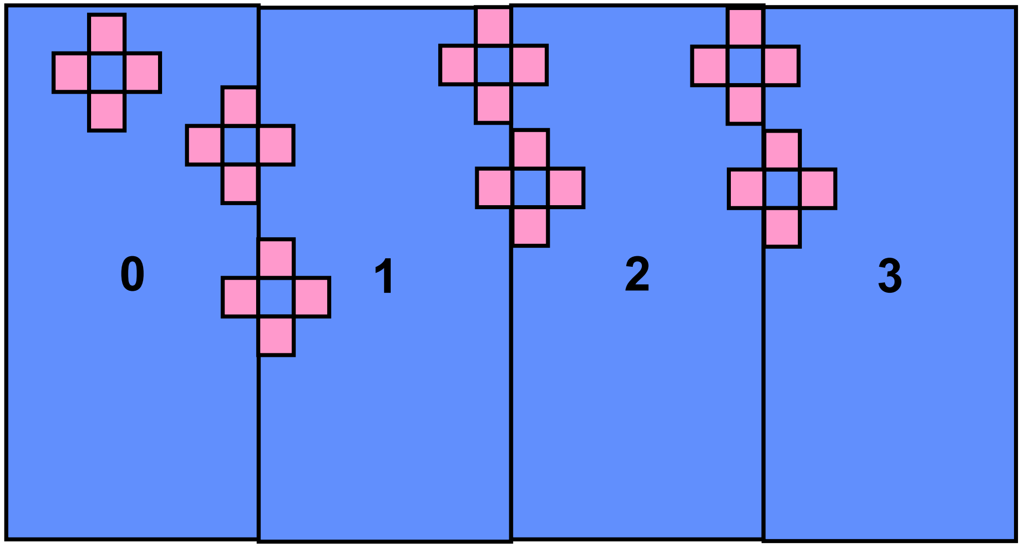
MPI Example: stencil

(**"Stencils" arise in solving differential equations, image filtering, and convolutional neural networks. There are *thousands* of research papers on efficient implementation of stencil problems!)**

- "stencil" example: each element is updated using a weighted sum of neighbour values

```
DO j=1, m
    DO i=1, n
        B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
    END DO
END DO
```

- To do this in parallel we could simply partition the outer loop
- At the strip boundaries, we need access to a column of neighbour data values
- In MPI we have to make this communication explicit



http://www.netlib.org/utk/papers/mpi-book/node51.html

Stencils in OpenMP

```
while (!converged) {
  #pragma omp parallel for private(j) collapse(2)
  for(i=0; j<N; ++j)
    for(j=0; j<M; ++j)
        B[i][j]=0.25*(A[i-1][j]+A[i+][j]+A[i][j-1]+A[i][j+1]);
  #pragma omp parallel for private(j) collapse(2)
  for(i=0; j<M; ++j)
    for(j=0; j<M; ++j)
        A[i][j] = B[i][j];
}
```

First loop nest depends on A and produces new values for A – so we have to “double-buffer” into B, and copy the new values back (after a barrier synchronisation)

- (we have omitted code to determine whether convergence has been reached)

MPI Example: stencil

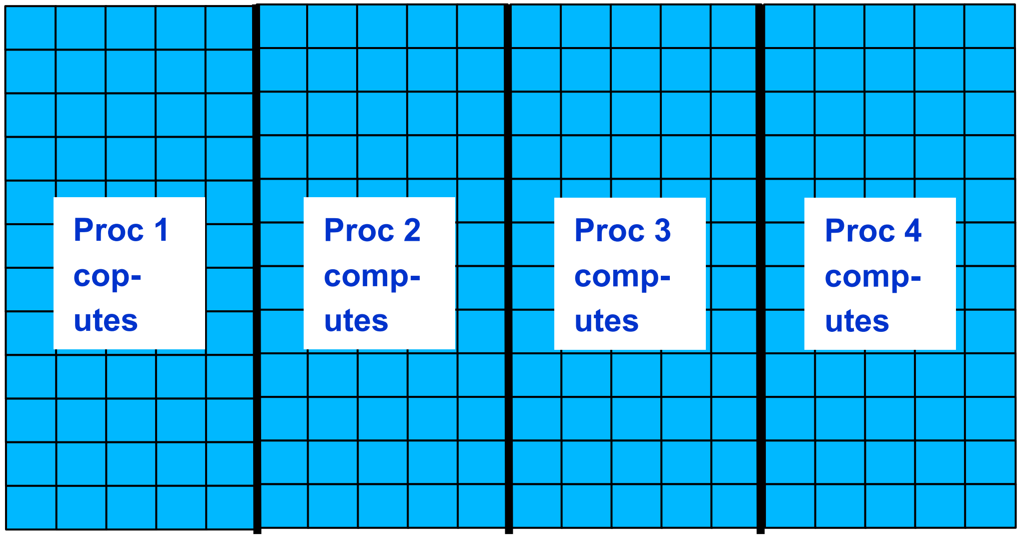
(“Stencils” arise in solving differential equations, image filtering, and convolutional neural networks. There are *thousands* of research papers on efficient implementation of stencil problems!)

- “stencil” example: each element is updated using a weighted sum of neighbour values

```
DO j=1, m
    DO i=1, n
        B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
    END DO
END DO
```

  - To do this in parallel we could simply partition the outer loop
  - At the strip boundaries, we need access to a column of neighbour data values
  - In MPI we have to make this communication explicit

Each processor computes values for its own, disjoint slice of the data



http://www.netlib.org/utk/papers/mpi-book/node51.html

MPI Example: stencil

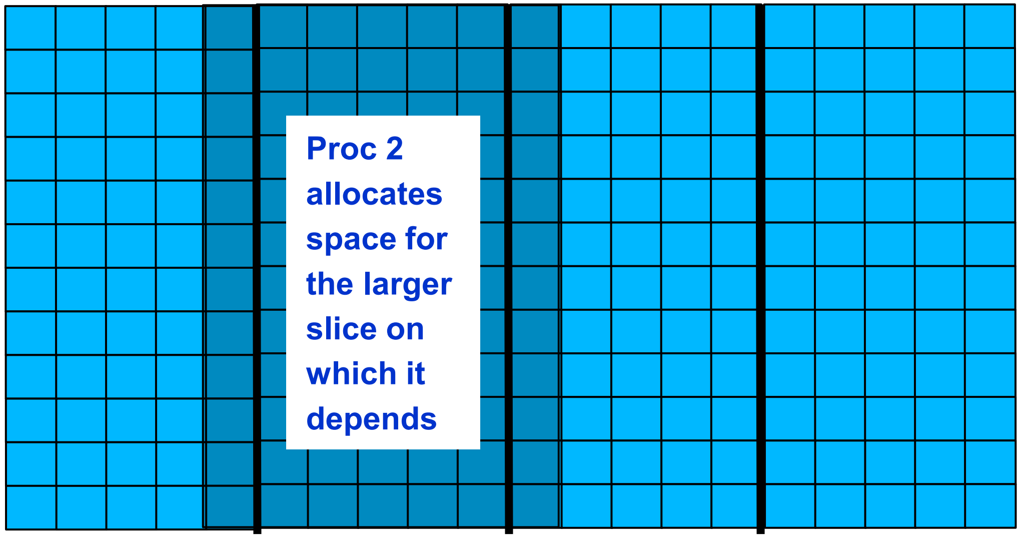
(“Stencils” arise in solving differential equations, image filtering, and convolutional neural networks. There are *thousands* of research papers on efficient implementation of stencil problems!)

- “stencil” example: each element is updated using a weighted sum of neighbour values

```
DO j=1, m
    DO i=1, n
        B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
    END DO
END DO
```

- To do this in parallel we could simply partition the outer loop
- At the strip boundaries, we need access to a column of neighbour data values
- In MPI we have to make this communication explicit

Each processor’s slice of work depends on a larger slice of the data



Proc 2 allocates space for the larger slice on which it depends

http://www.netlib.org/utk/papers/mpi-book/node51.html

MPI Example: stencil

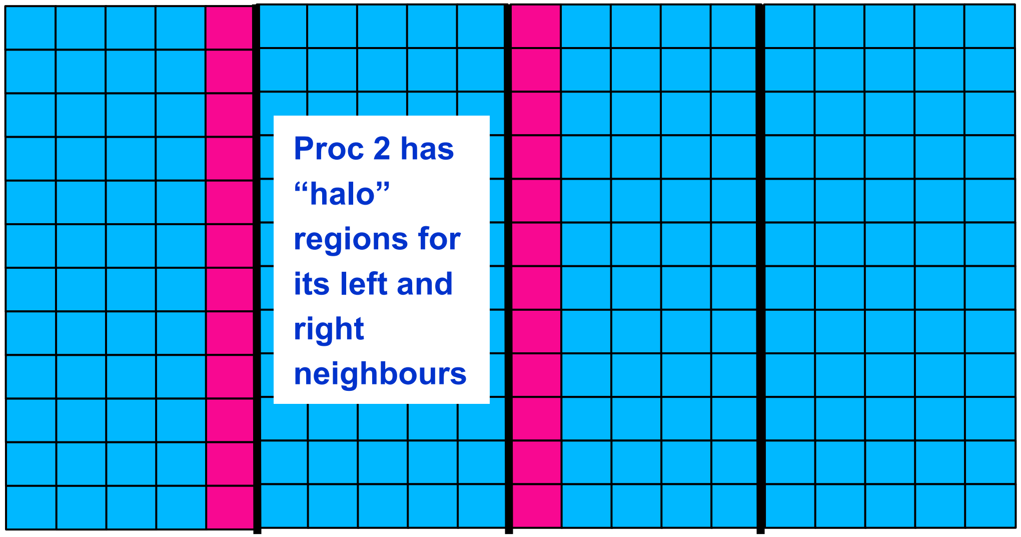
(“Stencils” arise in solving differential equations, image filtering, and convolutional neural networks. There are *thousands* of research papers on efficient implementation of stencil problems!)

- “stencil” example: each element is updated using a weighted sum of neighbour values

```
DO j=1, m
    DO i=1, n
        B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
    END DO
END DO
```

- To do this in parallel we could simply partition the outer loop
- At the strip boundaries, we need access to a column of neighbour data values
- In MPI we have to make this communication explicit

“Halo” region is allocated on each processor, updated by messages



Proc 2 has “halo” regions for its left and right neighbours

http://www.netlib.org/utk/papers/mpi-book/node51.html

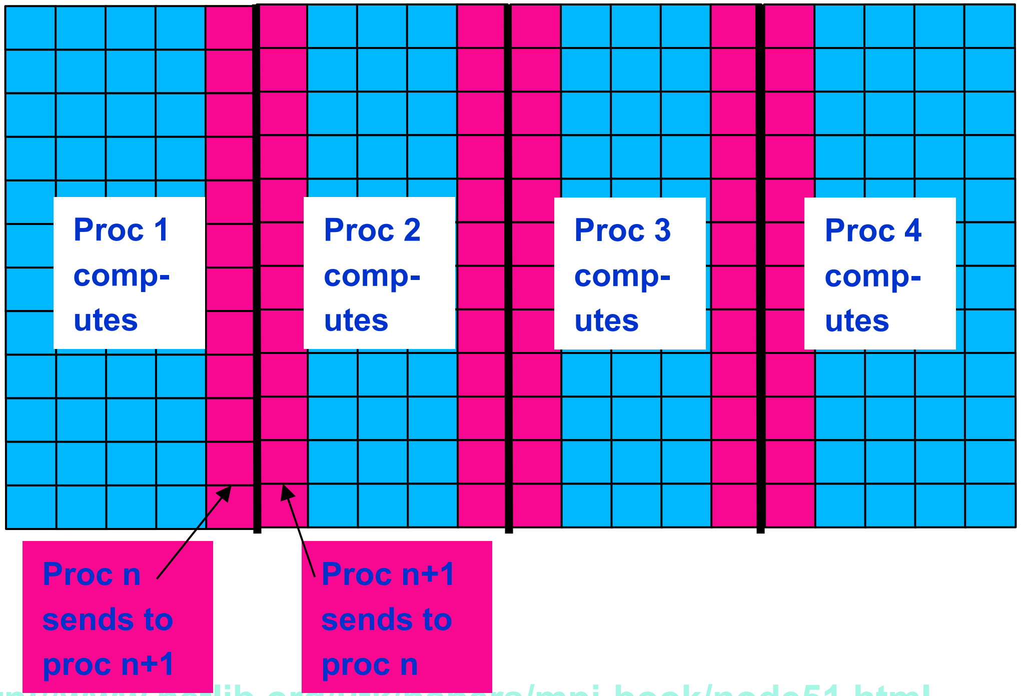
MPI Example: stencil

(“Stencils” arise in solving differential equations, image filtering, and convolutional neural networks. There are *thousands* of research papers on efficient implementation of stencil problems!)

- “stencil” example: each element is updated using a weighted sum of neighbour values

```
DO j=1, m
    DO i=1, n
        B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
    END DO
END DO
```

- To do this in parallel we could simply partition the outer loop
- At the strip boundaries, we need access to a column of neighbour data values
- In MPI we have to make this communication explicit



http://www.netlib.org/utk/papers/mpi-book/node51.html

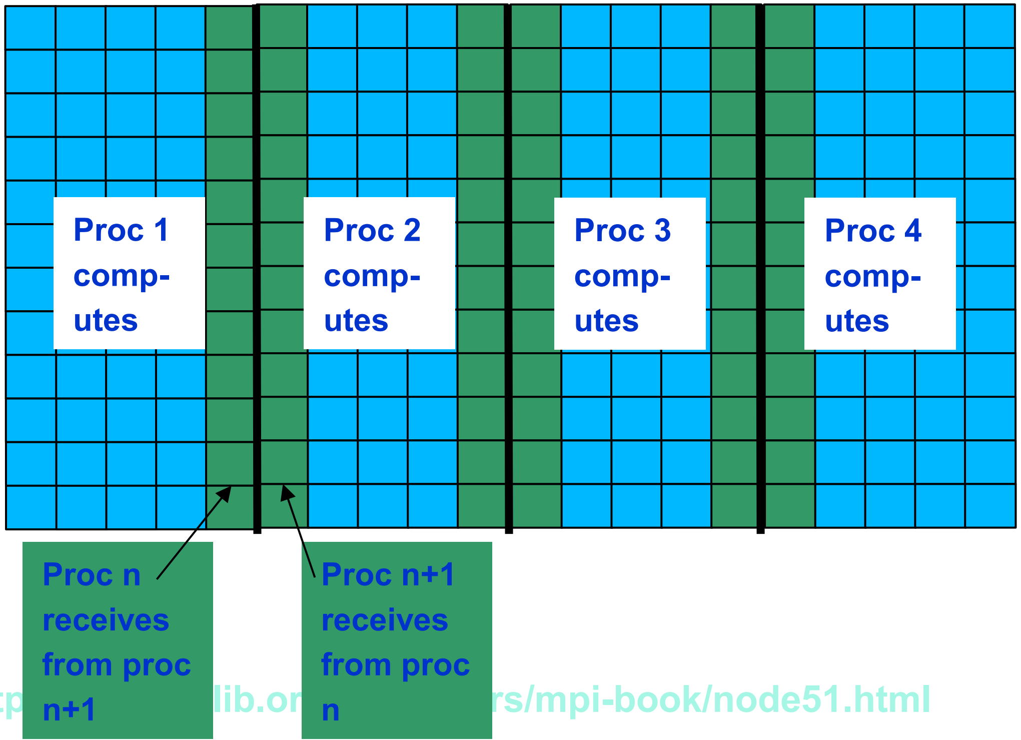
MPI Example: stencil

(“Stencils” arise in solving differential equations, image filtering, and convolutional neural networks. There are *thousands* of research papers on efficient implementation of stencil problems!)

- “stencil” example: each element is updated using a weighted sum of neighbour values

```
DO j=1, m
    DO i=1, n
        B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
    END DO
END DO
```

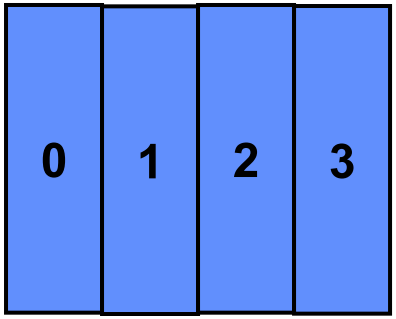
- To do this in parallel we could simply partition the outer loop
- At the strip boundaries, we need access to a column of neighbour data values
- In MPI we have to make this communication explicit



http lib.or rs/mpi-book/node51.html

MPI Example: initialisation

- SPMD
  - “Single Program, Multiple Data”
  - Each processor executes the program
  - First has to work out what part it is to play
  - “myrank” is index of this CPU
  - “comm” is MPI “communicator” – abstract index space of p processors
  - In this example, array is partitioned into slices



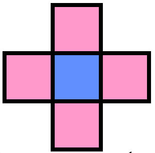
http://www.netlib.org/utk/papers/mpi-book/node51.html

```
! Compute number of processes and myrank
    CALL MPI_COMM_SIZE(comm, p, ierr)
    CALL MPI_COMM_RANK(comm, myrank, ierr)
! compute size of local block
    m = n/p
    IF (myrank.LT.(n-p*m)) THEN
                m = m+1
    END IF
! Compute neighbors
    IF (myrank.EQ.0) THEN
                left = MPI_PROC_NULL
    ELSE left = myrank - 1
    END IF
    IF (myrank.EQ.p-1)THEN
                right = MPI_PROC_NULL
    ELSE right = myrank+1
    END IF
! Allocate local arrays
    ALLOCATE (A(0:n+1,0:m+1), B(n,m))
```

(Continues on next slide)

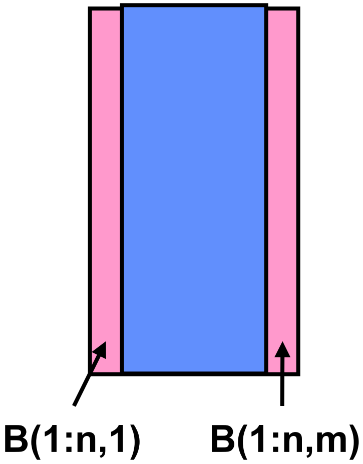
Example: Jacobi2D

- Sweep over A computing moving average of neighbouring four elements



- Compute new array B from A, then copy it back into B
- This version tries to overlap communication with computation

```
!Main Loop
DO WHILE(.NOT.converged)
       ! compute boundary iterations so they're ready to be sent right away
       DO i=1, n
              B(i,1)=0.25*(A(i-1,j)+A(i+1,j)+A(i,0)+A(i,2))
              B(i,m)=0.25*(A(i-1,m)+A(i+1,m)+A(i,m-1)+A(i,m+1))
       END DO
       ! Communicate
       CALL MPI_ISEND(B(1,1),n, MPI_REAL, left, tag, comm, req(1), ierr)
       CALL MPI_ISEND(B(1,m),n, MPI_REAL, right, tag, comm, req(2), ierr)
       CALL MPI_IRECV(A(1,0),n, MPI_REAL, left, tag, comm, req(3), ierr)
       CALL MPI_IRECV(A(1,m+1),n, MPI_REAL, right, tag, comm, req(4), ierr)
       ! Compute interior
       DO j=2, m-1
              DO i=1, n
                     B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
              END DO
       END DO
       DO j=1, m
              DO i=1, n
                     A(i,j) = B(i,j)
              END DO
       END DO
       ! Complete communication
       DO i=1, 4
              CALL MPI_WAIT(req(i), status(1.i), ierr)
       END DO
END DO
```



http://www.netlib.org/utk/papers/mpi-book/node51.html

MPI vs OpenMP

- **Which is better – OpenMP or MPI?**

MPI vs OpenMP

- **Which is better – OpenMP or MPI?**
- **OpenMP is easy!**
  - **But it hides the communication**
  - **And unintended sharing can lead to tricky bugs**

MPI vs OpenMP

- Which is better – OpenMP or MPI?

- OpenMP is easy!
  - But it hides the communication
  - And unintended sharing can lead to tricky bugs

- MPI is hard work
  - You need to make data partitioning explicit
  - No hidden communication
  - Seems to require more copying of data

MPI vs OpenMP

- Which is better – OpenMP or MPI?

- OpenMP is easy!
  - But it hides the communication
  - And unintended sharing can lead to tricky bugs

- MPI is hard work
  - You need to make data partitioning explicit
  - No hidden communication
  - Seems to require more copying of data
  - It's easier to see how to reduce communication and synchronisation (?)

- Lots of research on better parallel programming models…

Ch10 part 1 summary:

- Why go multi-core?
  - Limits of instruction-level parallelism
  - Limits of SIMD parallelism
  - Parallelism at low clock rate is energy-efficient
- How to program a parallel machine?
  - Explicitly-managed threads
  - Parallel loops
  - (many alternatives – dynamic thread pool, agents etc)
  - Message-passing (“distributed memory”)
- Where is the communication?
- Where is the synchronisation?
  - Design of programming models and software tools for parallelism and locality is major research focus

Additional slides for interest and context

Supercomputers: large distributed-memory machines with fast interconnect

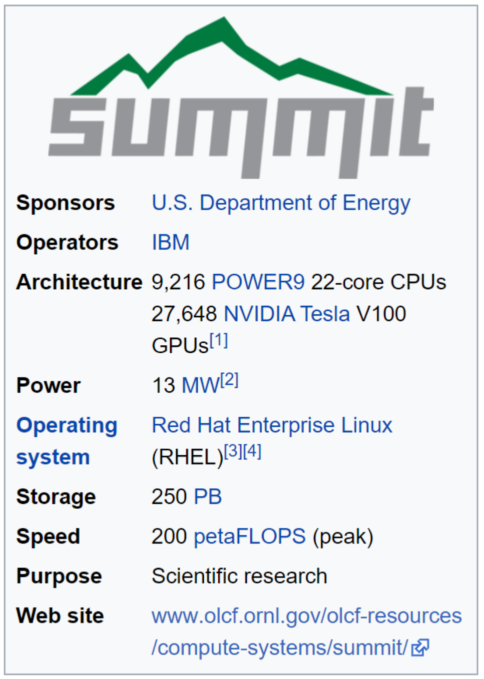
Usually (always?) programmed with MPI (and OpenMP, CUDA within each node)

Managed via batch queue

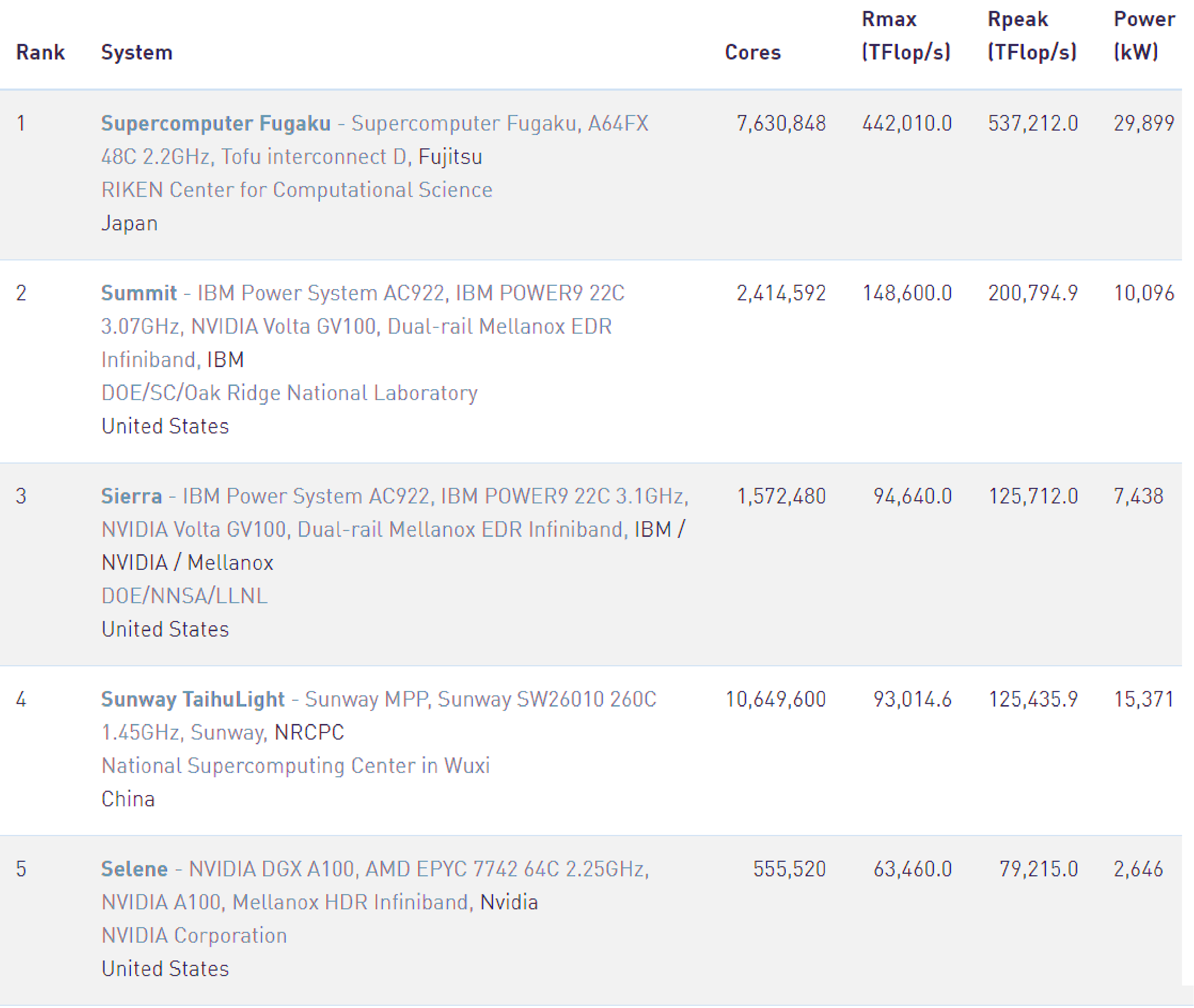
Supported by parallel filesystem

Image shows “Summit” – funded by US Dept of Energy. “Fastest computer in the world” 2018-2020. Part of 2014 $325M contract with IBM, NVIDIA and Mellanox

https://www.olcf.ornl.gov/2020/08/10/take-a-virtual-tour-of-ornls-supercomputer-center/


summit

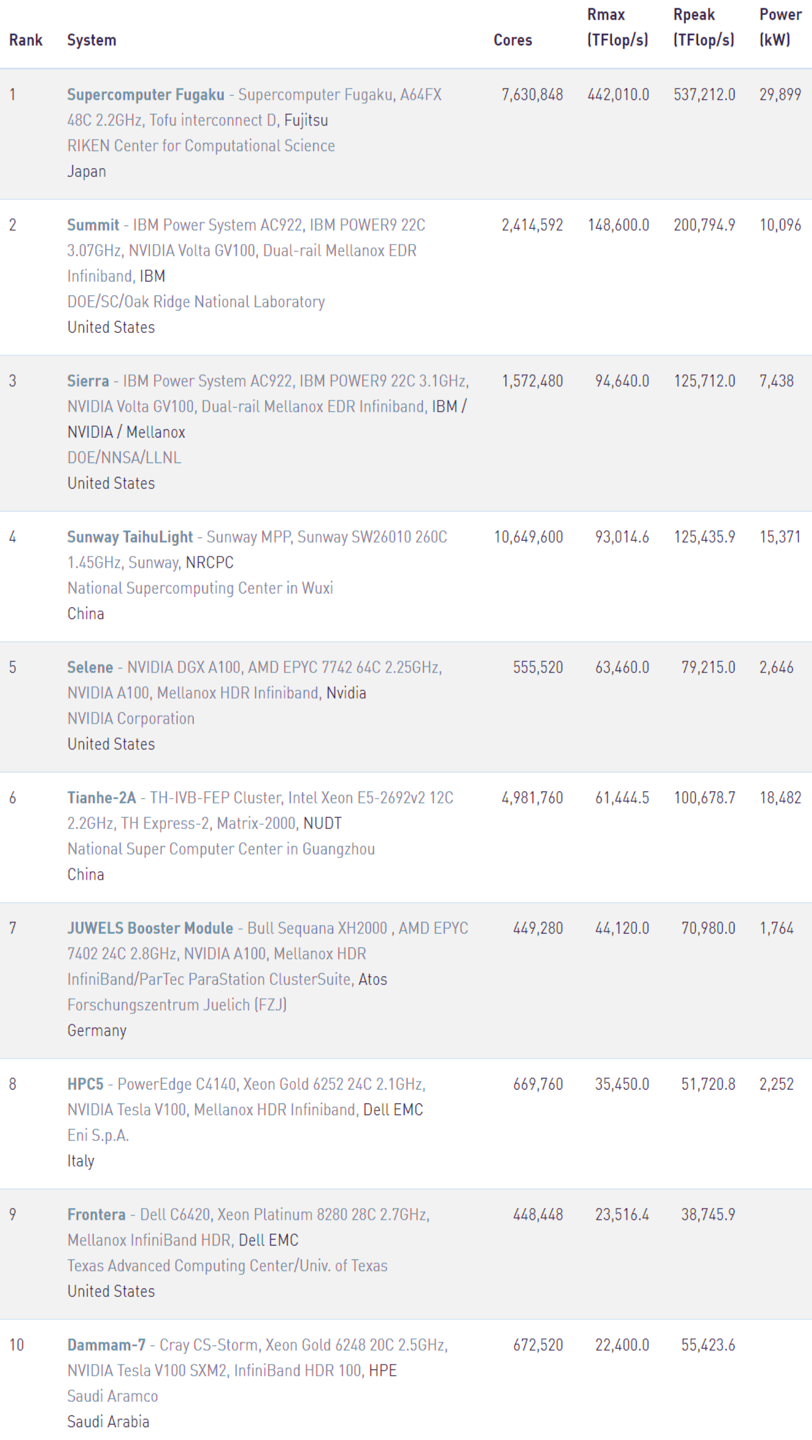
Sponsors	U.S. Department of Energy
Operators	IBM
Architecture	9,216 POWER9 22-core CPUs 27,648 NVIDIA Tesla V100 GPUs[1]
Power	13 MW[2]
Operating system	Red Hat Enterprise Linux (RHEL)[3][4]
Storage	250 PB
Speed	200 petaFLOPS (peak)
Purpose	Scientific research
Web site	www.olcf.ornl.gov/olcf-resources/compute-systems/summit/



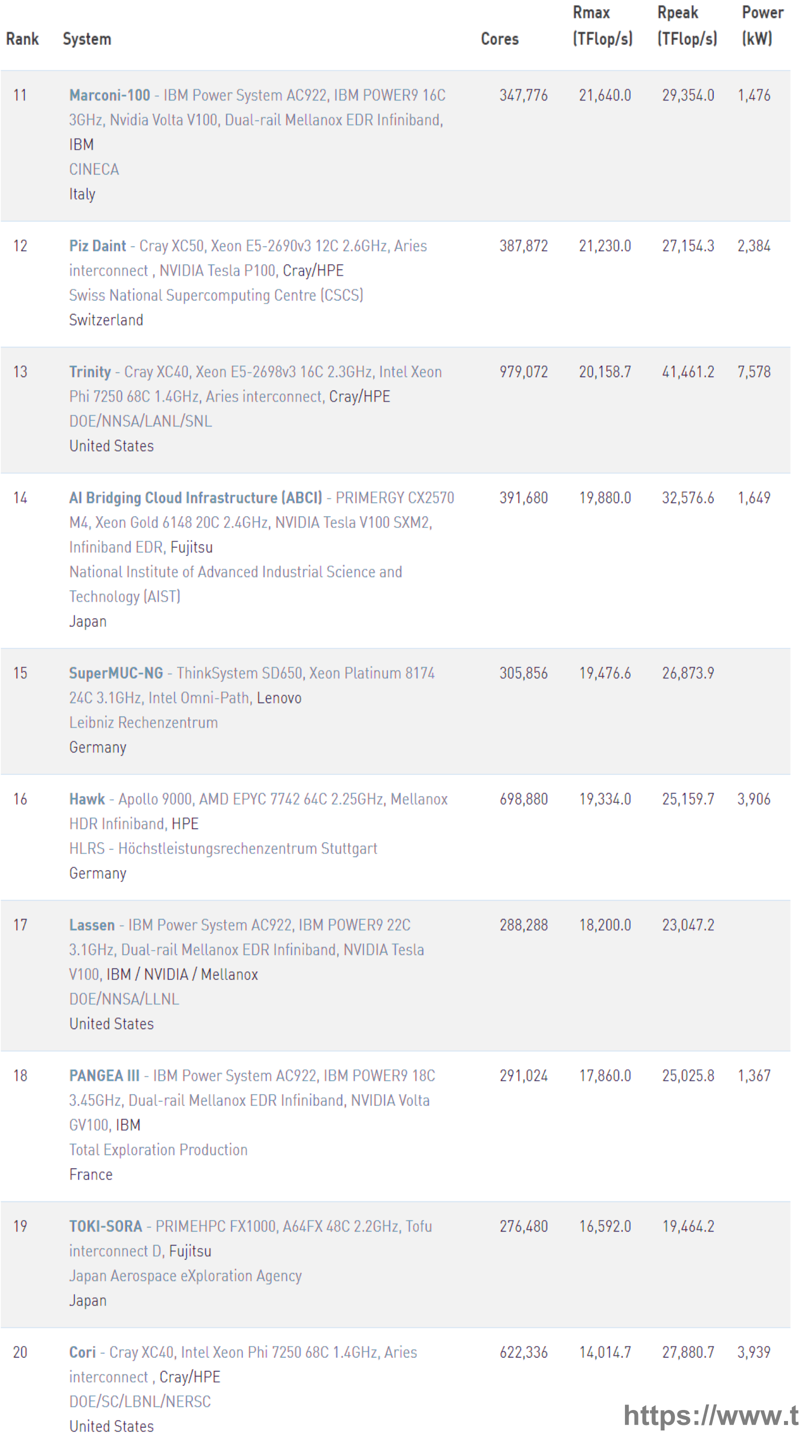
Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, 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
3	**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
4	**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
5	**Selene** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646

https://www.top500.org/lists/top500/list/2020/11/

- **TOP500 List (Nov 2020)**
- **Rmax and Rpeak values are in Gflops**
- **ranked by their performance on the LINPACK Benchmark.**
- **“to solve a dense system of linear equations. For the TOP500, we used that version of the benchmark that allows the user to scale the size of the problem and to optimize the software in order to achieve the best performance for a given machine”**



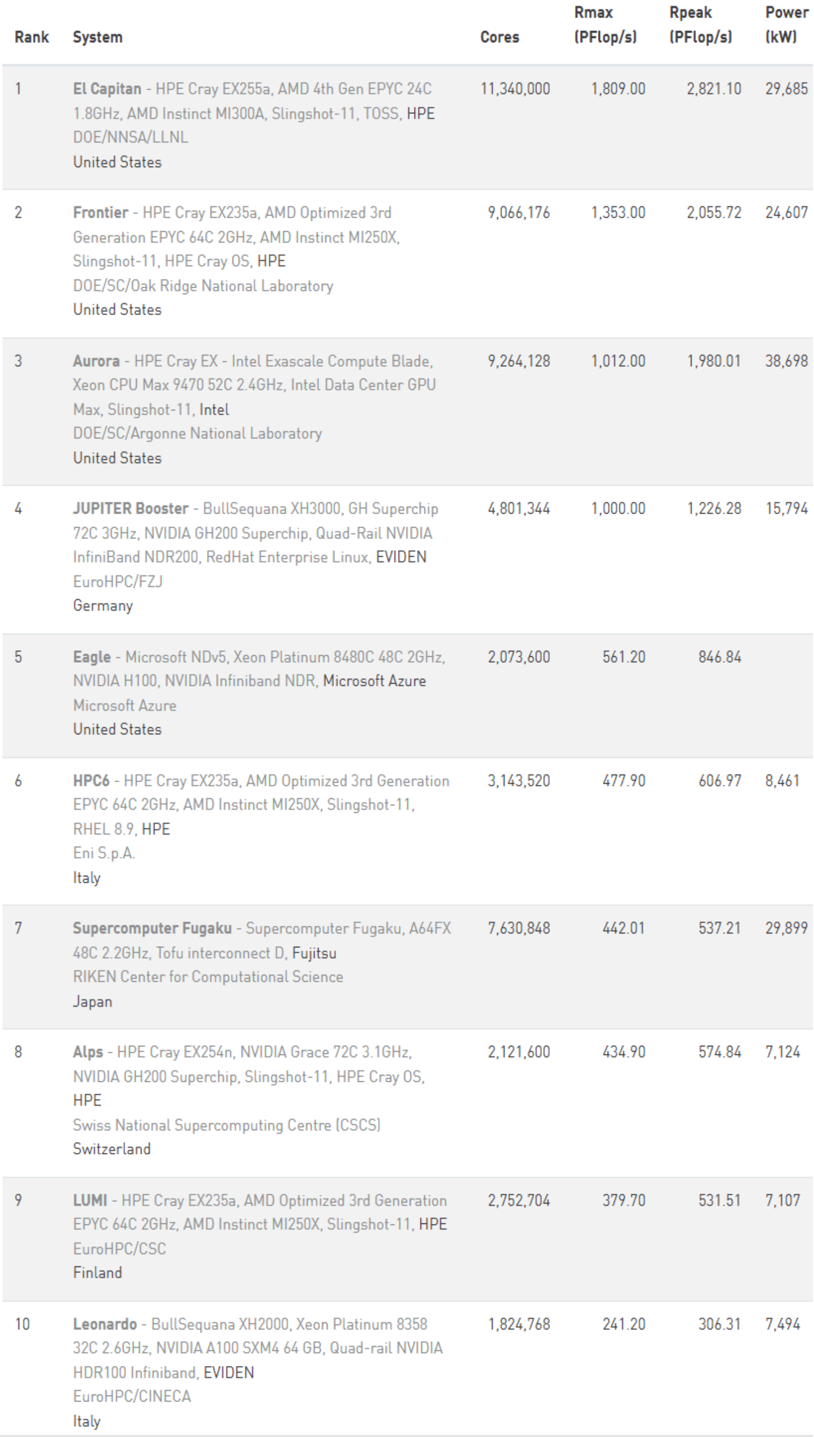
Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, 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
3	**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
4	**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
5	**Selene** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646
6	**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
7	**JUWELS Booster Module** - Bull Sequana XH2000 , AMD EPYC 7402 24C 2.8GHz, NVIDIA A100, Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, Atos Forschungszentrum Juelich (FZJ) Germany	449,280	44,120.0	70,980.0	1,764
8	**HPC5** - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A. Italy	669,760	35,450.0	51,720.8	2,252
9	**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	
10	**Dammam-7** - Cray CS-Storm, Xeon Gold 6248 20C 2.5GHz, NVIDIA Tesla V100 SXM2, InfiniBand HDR 100, HPE Saudi Aramco Saudi Arabia	672,520	22,400.0	55,423.6	

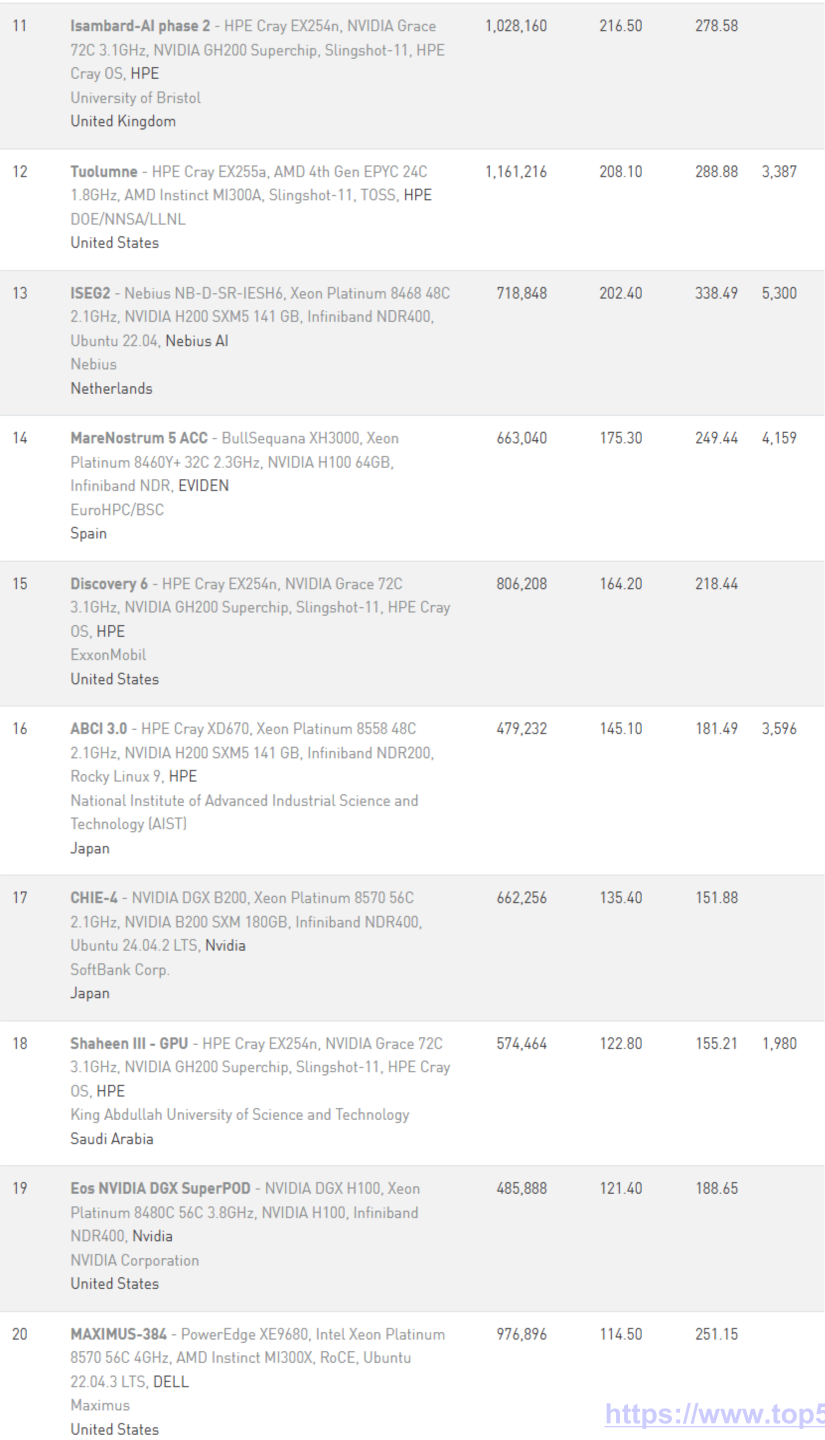


Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
11	**Marconi-100** - IBM Power System AC922, IBM POWER9 16C 3GHz, Nvidia Volta V100, Dual-rail Mellanox EDR Infiniband, IBM CINECA Italy	347,776	21,640.0	29,354.0	1,476
12	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100, Cray/HPE Swiss National Supercomputing Centre (CSCS) Switzerland	387,872	21,230.0	27,154.3	2,384
13	**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
14	**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
15	**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	
16	**Hawk** - Apollo 9000, AMD EPYC 7742 64C 2.25GHz, Mellanox HDR Infiniband, HPE HLRS - Höchstleistungsrechenzentrum Stuttgart Germany	698,880	19,334.0	25,159.7	3,906
17	**Lassen** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Tesla V100, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	288,288	18,200.0	23,047.2	
18	**PANGEA III** - IBM Power System AC922, IBM POWER9 18C 3.45GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Volta GV100, IBM Total Exploration Production France	291,024	17,860.0	25,025.8	1,367
19	**TOKI-SORA** - PRIMEHPC FX1000, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu Japan Aerospace eXploration Agency Japan	276,480	16,592.0	19,464.2	
20	**Cori** - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray/HPE DOE/SC/LBNL/NERSC United States	622,336	14,014.7	27,880.7	3,939

- **TOP500 List (Nov 2020)**
- **Rmax and Rpeak values are in Gflops**
- **ranked by Rmax - performance on the LINPACK Benchmark**
- **“to solve a dense system of linear equations. For the TOP500, we used that version of the benchmark that allows the user to scale the size of the problem and to optimize the software in order to achieve the best performance for a given machine”**

https://www.top500.org/lists/top500/list/2020/11/





Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**El Capitan** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	11,340,000	1,809.00	2,821.10	29,685
2	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, HPE DOE/SC/Oak Ridge National Laboratory United States	9,066,176	1,353.00	2,055.72	24,607
3	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
4	**JUPITER Booster** - BullSequana XH3000, GH Superchip 72C 3GHz, NVIDIA GH200 Superchip, Quad-Rail NVIDIA InfiniBand NDR200, RedHat Enterprise Linux, EVIDEN EuroHPC/FZJ Germany	4,801,344	1,000.00	1,226.28	15,794
5	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
6	**HPC6** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, RHEL 8.9, HPE Eni S.p.A. Italy	3,143,520	477.90	606.97	8,461
7	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
8	**Alps** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Cray OS, HPE Swiss National Supercomputing Centre (CSCS) Switzerland	2,121,600	434.90	574.84	7,124
9	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107
10	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, EVIDEN EuroHPC/CINECA Italy	1,824,768	241.20	306.31	7,494
11	**Isambard-AI phase 2** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Cray OS, HPE University of Bristol United Kingdom	1,028,160	216.50	278.58	
12	**Tuolumne** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	1,161,216	208.10	288.88	3,387
13	**ISEG2** - Nebius NB-D-SR-IESH6, Xeon Platinum 8468 48C 2.1GHz, NVIDIA H200 SXM5 141 GB, Infiniband NDR400, Ubuntu 22.04, Nebius AI Nebius Netherlands	718,848	202.40	338.49	5,300
14	**MareNostrum 5 ACC** - BullSequana XH3000, Xeon Platinum 8460Y+ 32C 2.3GHz, NVIDIA H100 64GB, Infiniband NDR, EVIDEN EuroHPC/BSC Spain	663,040	175.30	249.44	4,159
15	**Discovery 6** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Cray OS, HPE ExxonMobil United States	806,208	164.20	218.44	
16	**ABCI 3.0** - HPE Cray XD670, Xeon Platinum 8558 48C 2.1GHz, NVIDIA H200 SXM5 141 GB, Infiniband NDR200, Rocky Linux 9, HPE National Institute of Advanced Industrial Science and Technology (AIST) Japan	479,232	145.10	181.49	3,596
17	**CHIE-4** - NVIDIA DGX B200, Xeon Platinum 8570 56C 2.1GHz, NVIDIA B200 SXM 180GB, Infiniband NDR400, Ubuntu 24.04.2 LTS, Nvidia SoftBank Corp. Japan	662,256	135.40	151.88	
18	**Shaheen III - GPU** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Cray OS, HPE King Abdullah University of Science and Technology Saudi Arabia	574,464	122.80	155.21	1,980
19	**Eos NVIDIA DGX SuperPOD** - NVIDIA DGX H100, Xeon Platinum 8480C 56C 3.8GHz, NVIDIA H100, Infiniband NDR400, Nvidia NVIDIA Corporation United States	485,888	121.40	188.65	
20	**MAXIMUS-384** - PowerEdge XE9680, Intel Xeon Platinum 8570 56C 4GHz, AMD Instinct MI300X, RoCE, Ubuntu 22.04.3 LTS, DELL Maximus United States	976,896	114.50	251.15	

- **TOP500 List (Nov 2025)**
- **Rmax and Rpeak values are in Gflops**
- **ranked by Rmax - performance on the LINPACK Benchmark**
- **“to solve a dense system of linear equations. For the TOP500, we used that version of the benchmark that allows the user to scale the size of the problem and to optimize the software in order to achieve the best performance for a given machine”**
- **https://www.top500.org/statistics/sublist/**

https://www.top500.org/lists/top500/list/2025/11/

Google datacentres

https://datacenterfrontier.com/inside-a-google-data-center-2020-version/

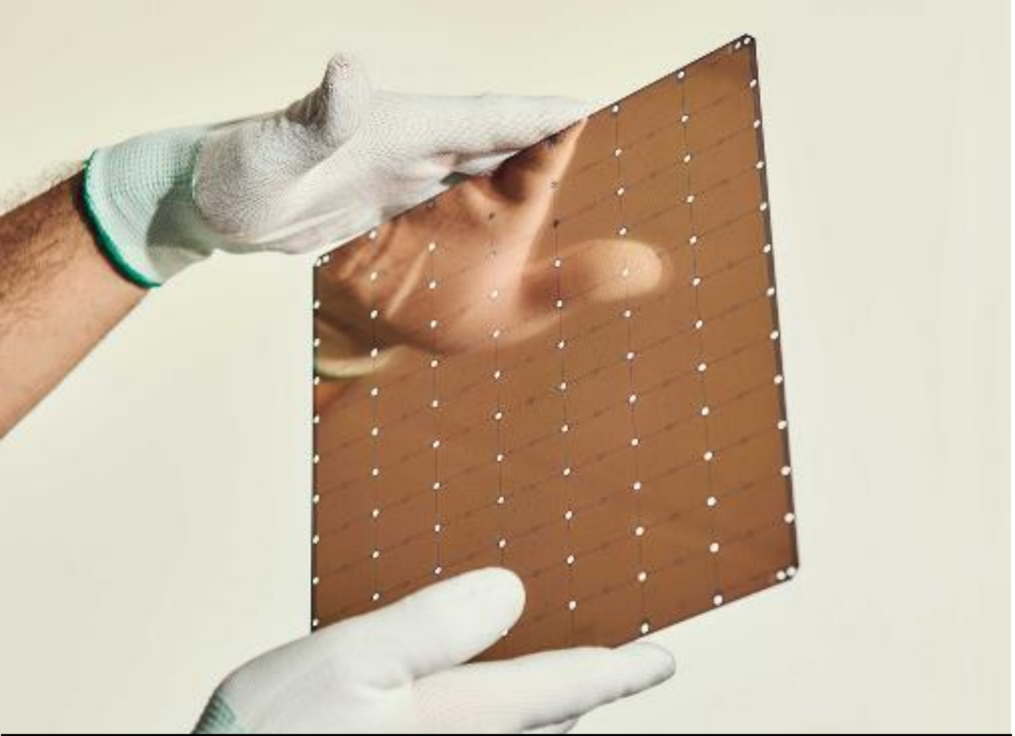
Google Data Center, Council Bluffs Iowa. Image Credit: Caddavis.photography from United States, CC BY 2.0 https://creativecommons.org/licenses/by/2.0, via Wikimedia Commons

What are parallel computers used for?

http://www.coloandcloud.com/wp-content/uploads/2011/12/Quincy-Data-Center-Facilities-1024x634.jpg

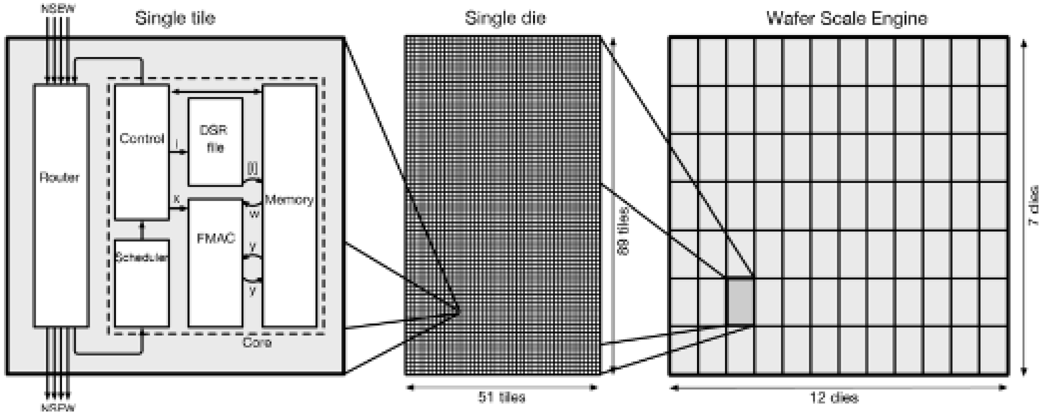
Kolos datacentre, at Ballangen (Norway), inside the Arctic circle. ~~Not yet built~~ – was planned to expand to 600,000m^2 and 1,000MW, using cheapest electricity in Europe

https://kolos.com/



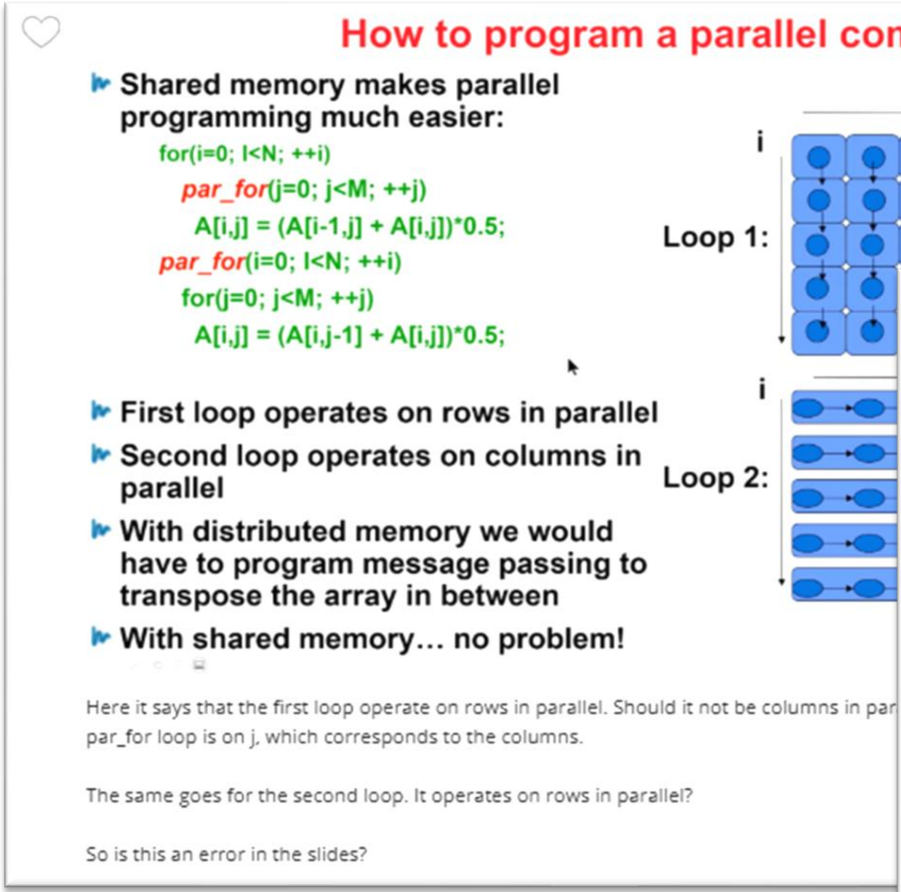
Cerebras CS-1

- 1.2 trillion transistors (cf largest GPUs, FPGAs, Graphcore etc ca. 30 billion)
- Ca.400,000 processor cores
- Ca.18GB SRAM
- TDP ca.17KW
- SRAM-to-core bandwidth “9 petabytes/s”
- Claimed 0.86PFLOPS (partially reduced precision floating point) on stencil CFD application



https://www.cerebras.net/beyond-ai-for-wafer-scale-compute-setting-records-in-computational-fluid-dynamics/

Student question: Sequential composition of parallel loops



How to program a parallel computer?

- Shared memory makes parallel programming much easier:

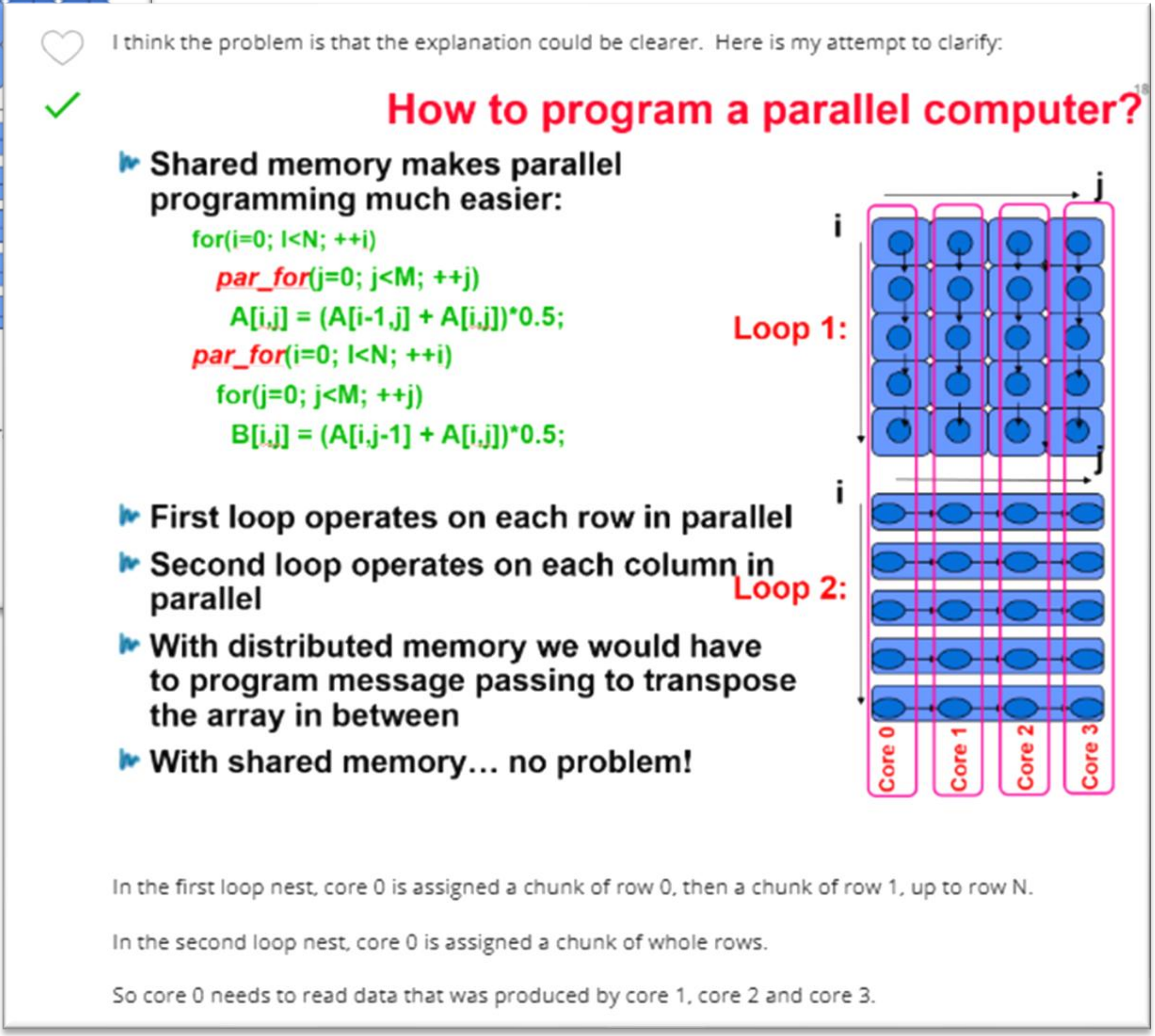
```
for(i=0; I<N; ++i)
  par_for(j=0; j<M; ++j)
    A[i,j] = (A[i-1,j] + A[i,j])*0.5;
par_for(i=0; I<N; ++i)
  for(j=0; j<M; ++j)
    A[i,j] = (A[i,j-1] + A[i,j])*0.5;
```

- First loop operates on rows in parallel
- Second loop operates on columns in parallel
- With distributed memory we would have to program message passing to transpose the array in between
- With shared memory… no problem!

Here it says that the first loop operate on rows in parallel. Should it not be columns in par
par_for loop is on j, which corresponds to the columns.

The same goes for the second loop. It operates on rows in parallel?

So is this an error in the slides?



I think the problem is that the explanation could be clearer. Here is my attempt to clarify:

How to program a parallel computer?

- Shared memory makes parallel programming much easier:

```
for(i=0; I<N; ++i)
  par_for(j=0; j<M; ++j)
    A[i,j] = (A[i-1,j] + A[i,j])*0.5;
par_for(i=0; I<N; ++i)
  for(j=0; j<M; ++j)
    B[i,j] = (A[i,j-1] + A[i,j])*0.5;
```

- First loop operates on each row in parallel
- Second loop operates on each column in parallel
- With distributed memory we would have to program message passing to transpose the array in between
- With shared memory… no problem!

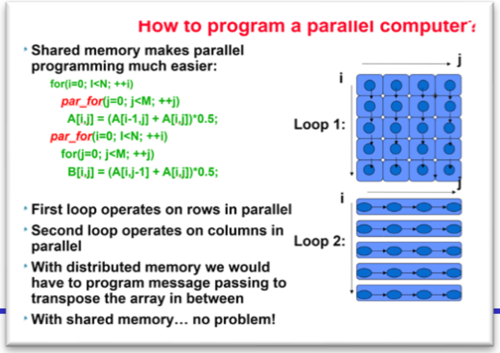
In the first loop nest, core 0 is assigned a chunk of row 0, then a chunk of row 1, up to row N.

In the second loop nest, core 0 is assigned a chunk of whole rows.

So core 0 needs to read data that was produced by core 1, core 2 and core 3.

Student question: Cache misses due to successive parallel loops

Q: you mention that the accesses to A[i,j-1] and A[i,j] in the last line of the second loop will cause cache misses. Please could you elaborate on this? Do we get cache misses because of the data that was allocated into the cache during the execution of the first loop?



How to program a parallel computer?

- Shared memory makes parallel programming much easier:

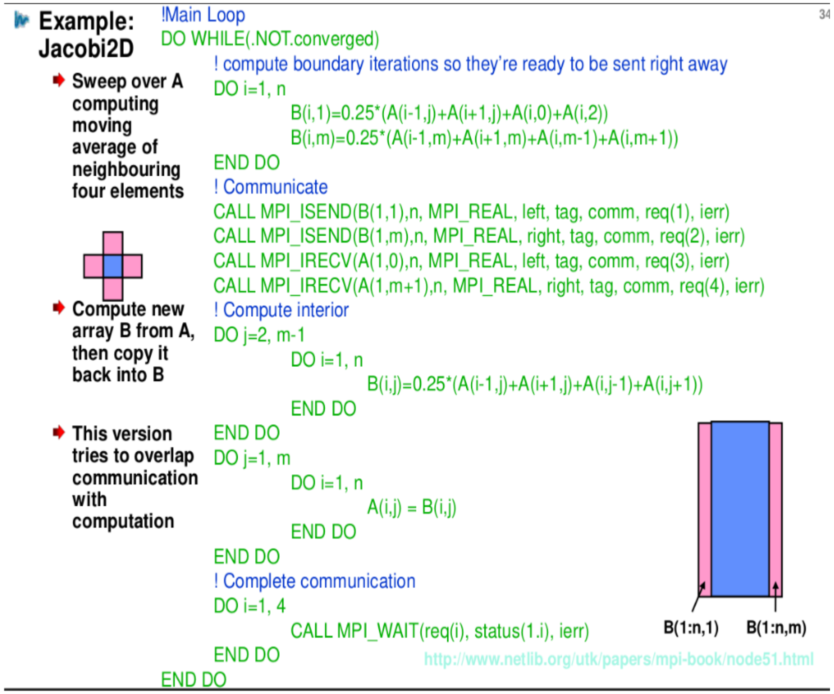
```
for(i=0; I<N; ++i)
  par_for(j=0; j<M; ++j)
    A[i,j] = (A[i-1,j] + A[i,j])*0.5;
par_for(i=0; I<N; ++i)
  for(j=0; j<M; ++j)
    B[i,j] = (A[i,j-1] + A[i,j])*0.5;
```

- First loop operates on rows in parallel
- Second loop operates on columns in parallel
- With distributed memory we would have to program message passing to transpose the array in between
- With shared memory… no problem!

- The first loop nest assigns to array A; the second one reads from it.
- Let's suppose that the first "par_for" loop runs on four cores - if M=100, then core0 might get iterations j=0:24, core1: 25:49, core3: 50:74, core 3: 75:99.
- When core0 executes the store instructions for the assignment "A[i,j] = (A[i-1,j] + A[i,j])*0.5;", it acquires ownership of the cache line on which A[i,j] falls. In fact core0 is going to acquire all the cache lines on which elements A[i,0:24] lie. Core1 will acquire A[i,24:49], etc.
- Now consider the second loop nest. This time we parallelise over i - so if N=10, core0 will get iterations i=0:2, core1: 3:5, core2: 6:8, core3: 9:11.
- Now core0 is going to read all 100 elements of each of the rows of A that it needs - that is, A[i,0:99]. So is core1, ditto core2, core3. So core0 will broadcast read requests for the whole row, and all the snooping cache controllers will be involved in providing this data.
- The same will happen with the other cores - there will be a storm of read requests.

Incidentally, if you were programming this with MPI, you could use an MPI_Broadcast() operation to achieve this effect much more efficiently. You might wonder whether there is some way to achieve the effect of such a broadcast in a cache coherency protocol; see Sarah Talbot's PhD work, *Using proxies to reduce controller contention in large shared-memory multiprocessors* https://link.springer.com/content/pdf/10.1007/BFb0024734.pdf

Student question: Neighbour exchange in stencil loop



Example: Jacobi2D

- Sweep over A computing moving average of neighbouring four elements
- Compute new array B from A, then copy it back into B
- This version tries to overlap communication with computation

```
!Main Loop
DO WHILE(.NOT.converged)
    ! compute boundary iterations so they're ready to be sent right away
    DO i=1, n
            B(i,1)=0.25*(A(i-1,j)+A(i+1,j)+A(i,0)+A(i,2))
            B(i,m)=0.25*(A(i-1,m)+A(i+1,m)+A(i,m-1)+A(i,m+1))
    END DO
    ! Communicate
    CALL MPI_ISEND(B(1,1),n, MPI_REAL, left, tag, comm, req(1), ierr)
    CALL MPI_ISEND(B(1,m),n, MPI_REAL, right, tag, comm, req(2), ierr)
    CALL MPI_IRECV(A(1,0),n, MPI_REAL, left, tag, comm, req(3), ierr)
    CALL MPI_IRECV(A(1,m+1),n, MPI_REAL, right, tag, comm, req(4), ierr)
    ! Compute interior
    DO j=2, m-1
            DO i=1, n
                    B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
            END DO
    END DO
    DO j=1, m
            DO i=1, n
                    A(i,j) = B(i,j)
            END DO
    END DO
    ! Complete communication
    DO i=1, 4
            CALL MPI_WAIT(req(i), status(1.i), ierr)
    END DO
END DO
```

http://www.netlib.org/utk/papers/mpi-book/node51.html

We have already calculated the values on the boundary. I think the job is done. Why do we still need to send the values (`CALL MPI_ISEND((B(1,1)...`) on the boundary and receive the values for calculating the boundary (`CALL MPI_IRECV(A(1,0...`)?

TL;DR: we split the domain into strips; each processor looks after a strip. There's a dependence - a data flow, between each processor and its left- and right-neighbours.

In detail:
We compute the values on the boundary in the first "DO i=1,n" loop. This computes values for the left column (B(1:n,1). ((And in the second line it computes the values for the right column, B(1:n,m).))
The calculation of the left boundary uses A(1:n,0) and A(1:n,2) - the values of the columns to the left and the right of the column B(1:n,1).

So the thing to understand here is that the subdomain that this processor is reading has five regions:
A(1:n,0) // read from the left neighbour processor
A(1:n,1) // computed by this processor *using* A(1:n,0) above [and then *sent* to the left neighbour]
A(1:n,2:m-1) // computed by this processor using only data computed on this processor
A(1:n,m) // computed by this processor using A{1:n,m+1) below [and then *sent* to the right neighbour]
A(1:n,m+1) // read from the right neighbour processor