

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 2022

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 Kubiawicz and Yujia Jin at Berkeley

What you should get from this³

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

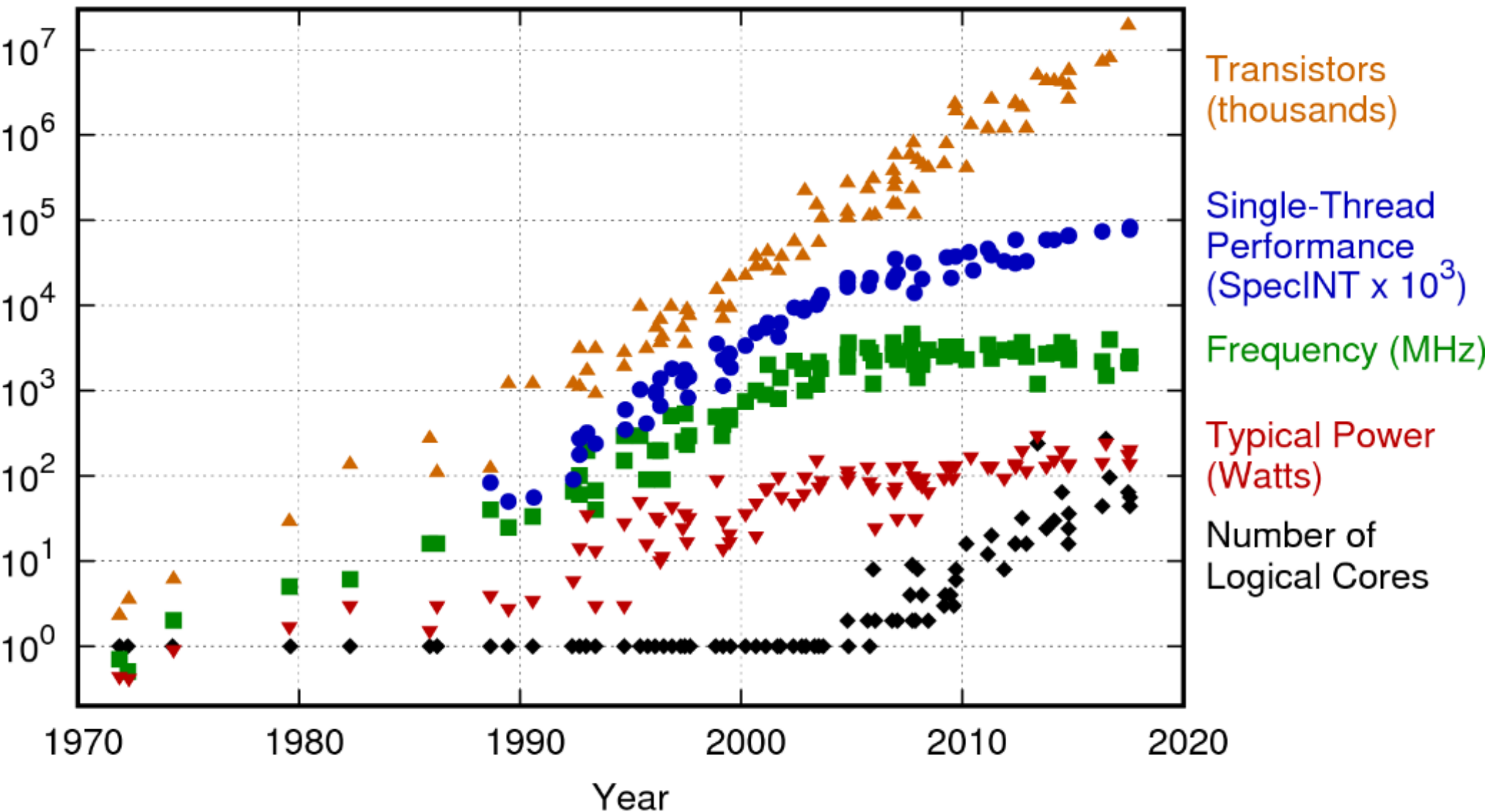
- ✚ Why power considerations motivate multicore
- ✚ Why is shared-memory parallel programming attractive?
 - ✚ How is dynamic load-balancing implemented?
 - ✚ Why is distributed-memory parallel programming harder but more likely to yield robust performance?
- ✚ What is the cache coherency problem
 - ✚ 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

5

Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at [OurWorldinData.org](https://www.ourworldindata.org). There you find more visualizations and research on this topic.



42 Years of Microprocessor Trend Data



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 2010-2017 by K. Rupp <https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/>

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

How to program a parallel computer?¹⁷

- Shared memory makes parallel programming much easier:

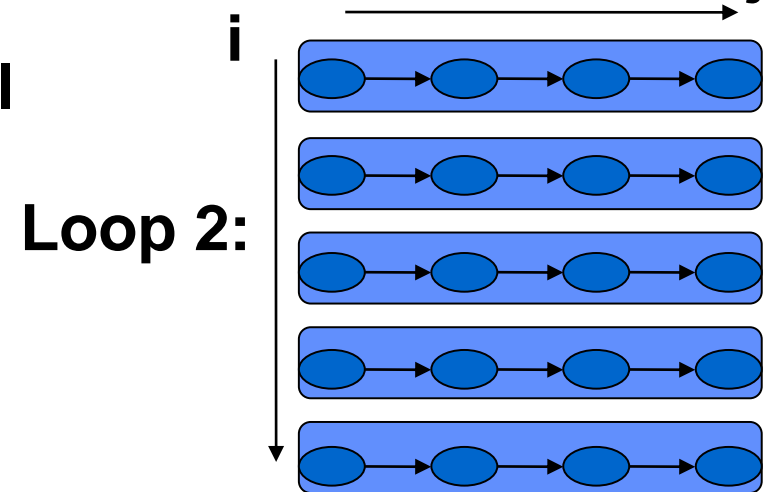
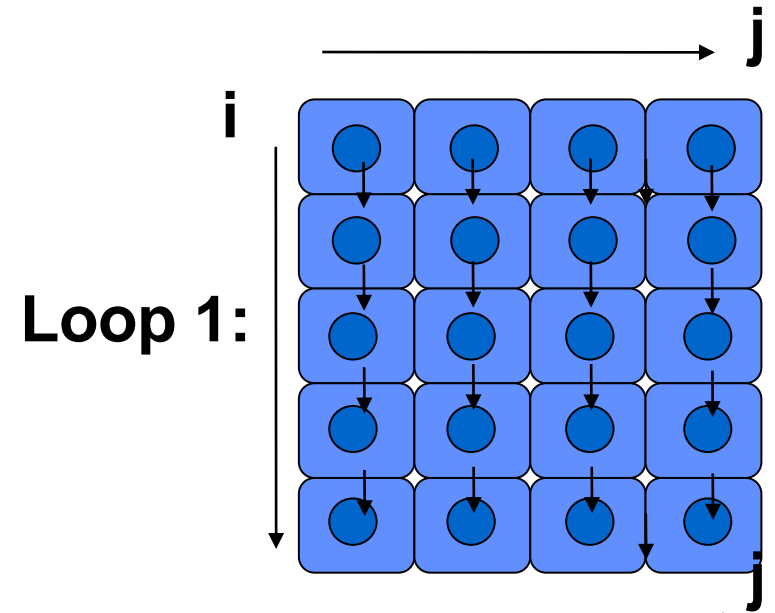
```
for(i=0; i<N; ++i)
  par_for(j=0; j<M; ++j)
    A[i,j] = (A[i-1,j] + A[i,j])*0.5;
par_for(i=0; i<N; ++i)
  for(j=0; j<M; ++j)
    A[i,j] = (A[i,j-1] + A[i,j])*0.5;
```

- 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!



How to program a parallel computer?¹⁸

Shared memory makes parallel programming much easier:

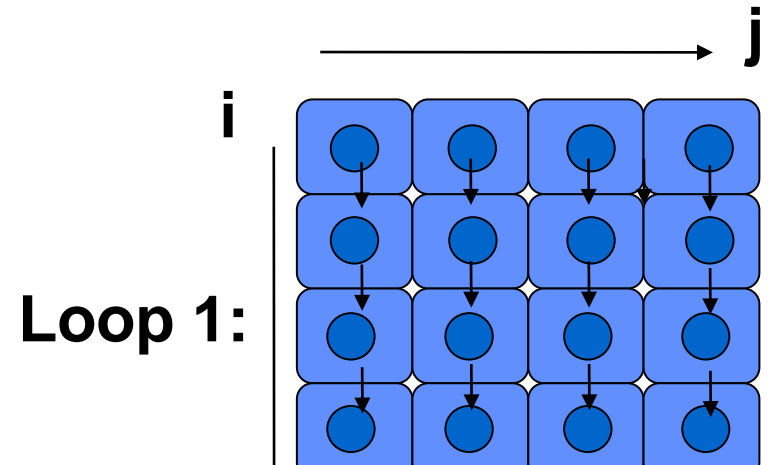
```
for(i=0; i<N; ++i)
  par_for(j=0; j<M; ++j)
    A[i,j] = (A[i-1,j] + A[i,j])*0.5;
par_for(i=0; i<N; ++i)
  for(j=0; j<M; ++j)
    A[i,j] = (A[i,j-1] + A[i,j])*0.5;
```

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!



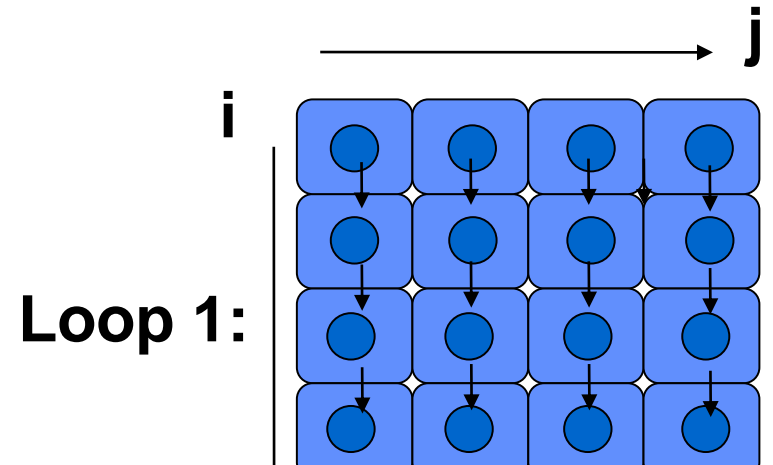
- Shared memory is *convenient*
- Shared memory is *fast* – communicate with just a load/store

How to program a parallel computer?¹⁹

- 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!



- 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 - OpenMP²⁰

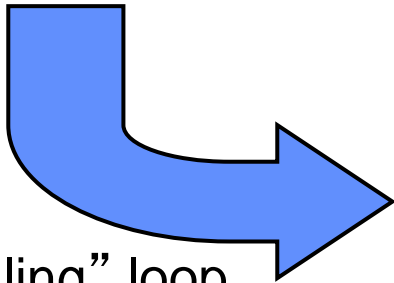
- OpenMP is a standard design for language extensions for shared-memory parallel programming
- Language bindings exist for Fortran, C, C++ and to some extent (eg research prototypes) for Java and C#
- Implementation requires compiler support – as found in GCC, clang/llvm, Intel's compilers, Microsoft 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] = 0.5*(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....

We could use locks:

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

Implementing Fetch-and-add

- ✚ 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.

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*

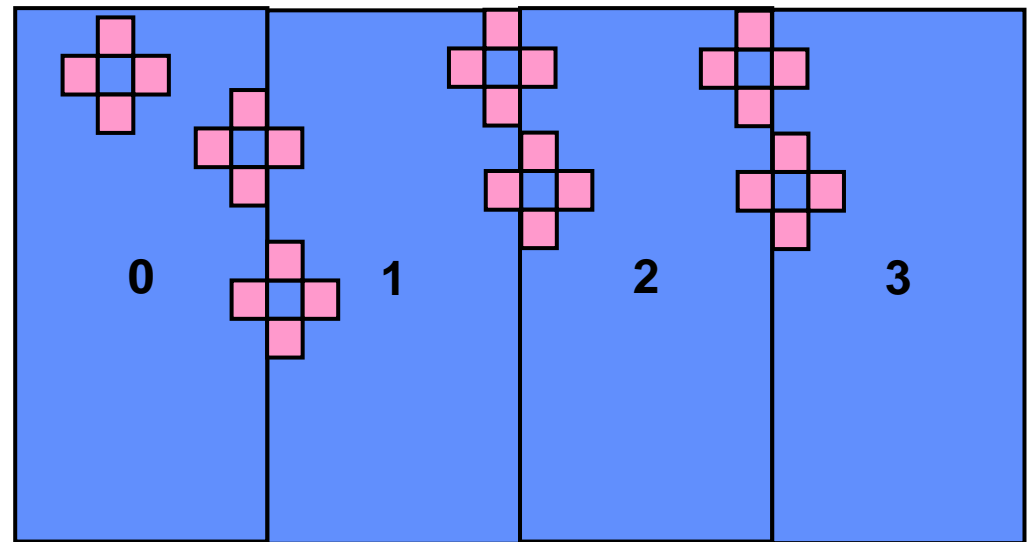
MPI Example: stencil

“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
```

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

- ➡ 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



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

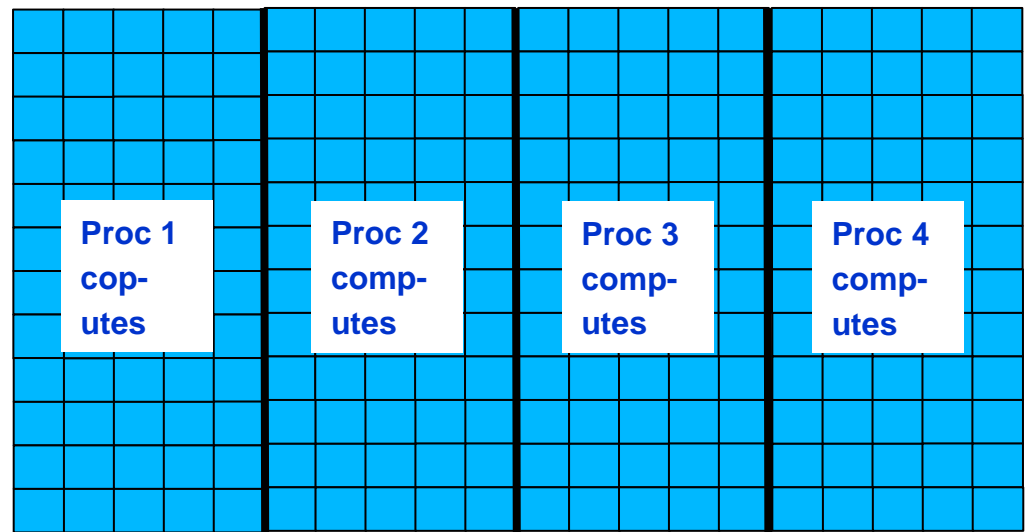
“stencil” example: each element is updated using a weighted sum of neighbour values

```
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
```

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

- ➡ 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



MPI Example: stencil

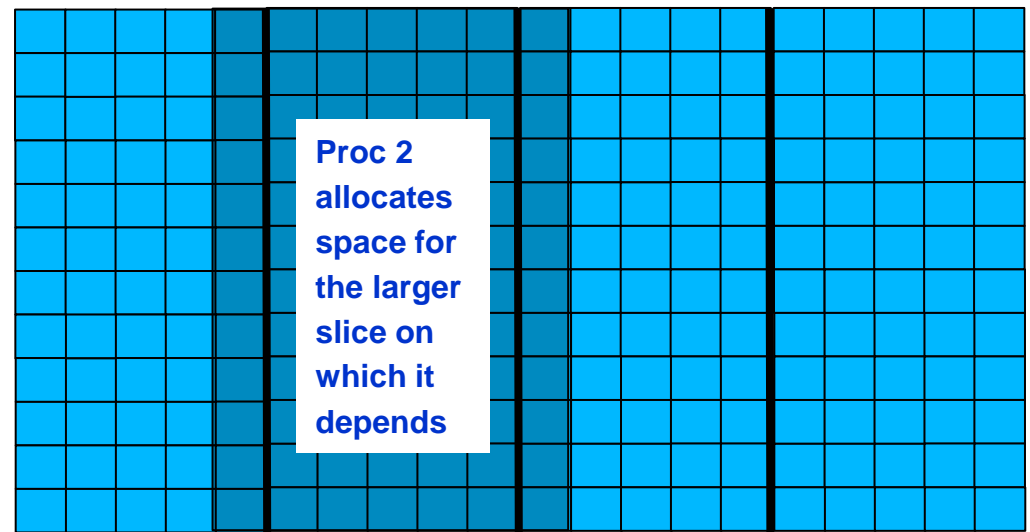
“stencil” example: each element is updated using a weighted sum of neighbour values

```
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
```

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

- ➡ 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



MPI Example: stencil

“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
```

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

- ➡ 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



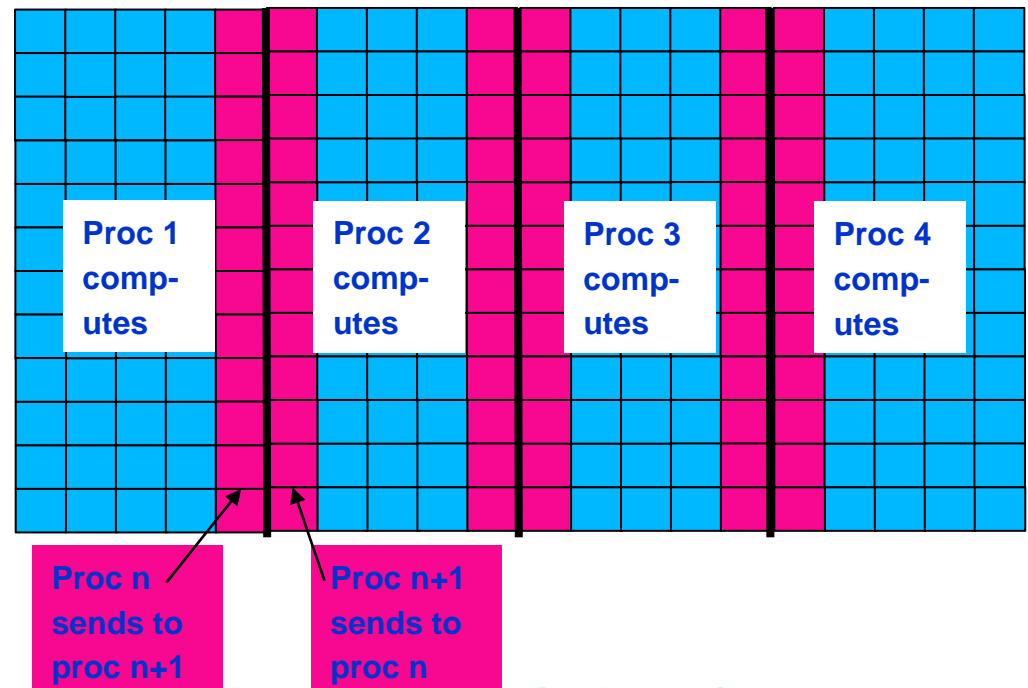
MPI Example: stencil

“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
```

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

- ➡ 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



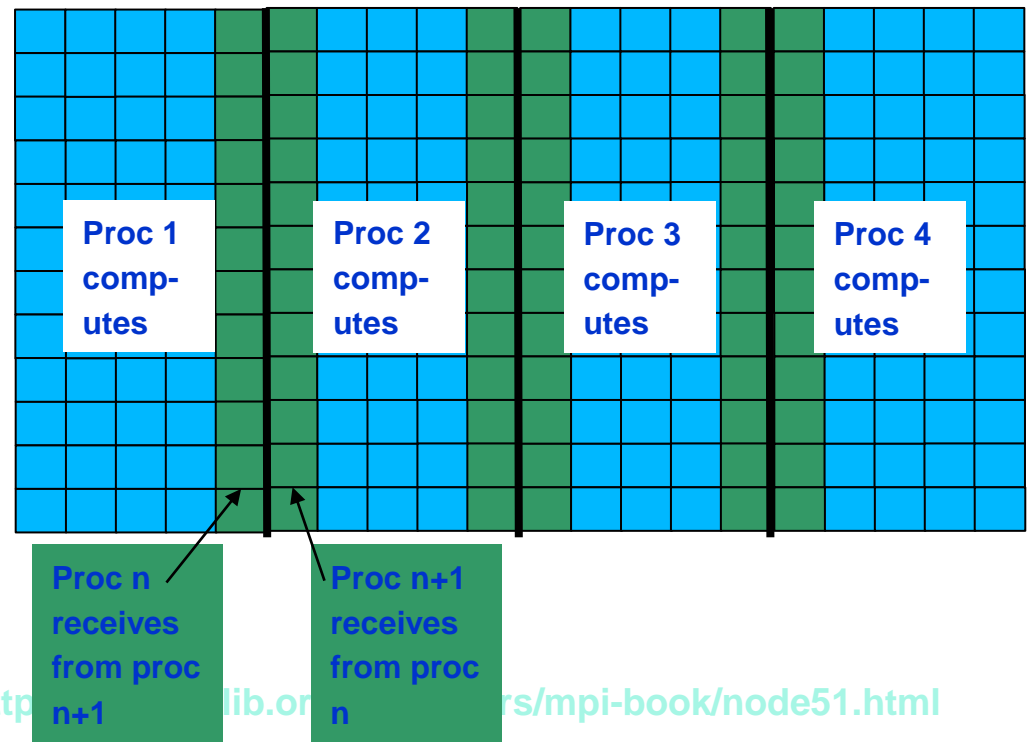
MPI Example: stencil

“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
```

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

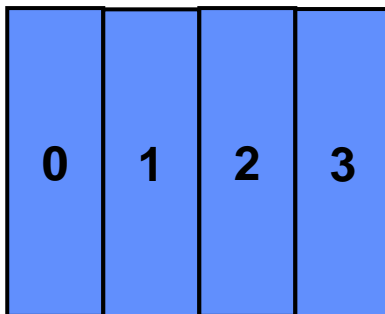
- ➡ To do this in parallel we could simply partition the outer loop
- ➡ At the strip boundaries, we need access to a column of neighbour data values
- ➡ In MPI we have to make this communication explicit



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



! 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

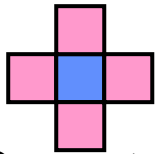
! Allocate local arrays

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



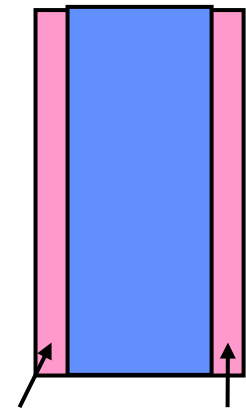
- ➡ Compute new array B from A, then copy it back into B

- ➡ 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

```



B(1:n,1) B(1:n,m)

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**



	
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/ 

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/>

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

- **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)	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	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
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	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
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	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
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	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
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	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	
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	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
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	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	
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	18	PANGAEA 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
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		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	
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		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”**



What are parallel computers used for?

QUINCY DATA CENTERS

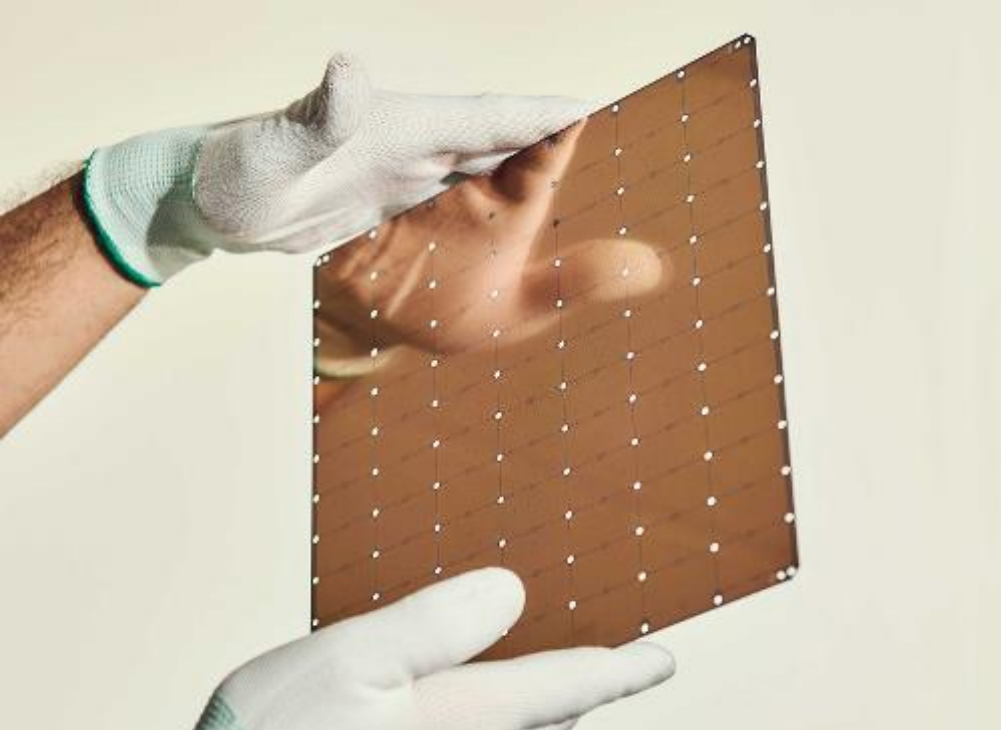


COLOANDCLOUD.COM



Kolos

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



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

