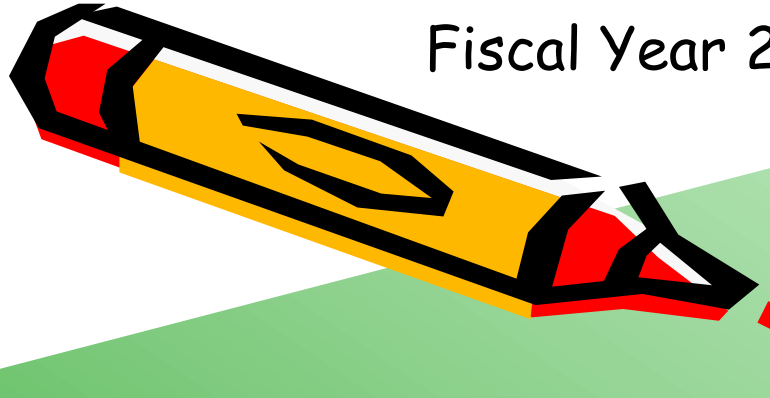
Fiscal Year 2025

Ver. 2026-01-08a



Course number: CSC.T440
School of Computing,
Graduate major in Computer Science

Computer Organization and Architecture

5. Thread Level Parallelism: Interconnection Network and Many-core Processors

www.arch.cs.titech.ac.jp/lecture/coa/
Room No. M-112(H117), Lecture (Face-to-face)
Thr 13:30-15:10

Kenji Kise, Department of Computer Science
kise[at]comp.isct.ac.jp

From multi-core era to many-core era

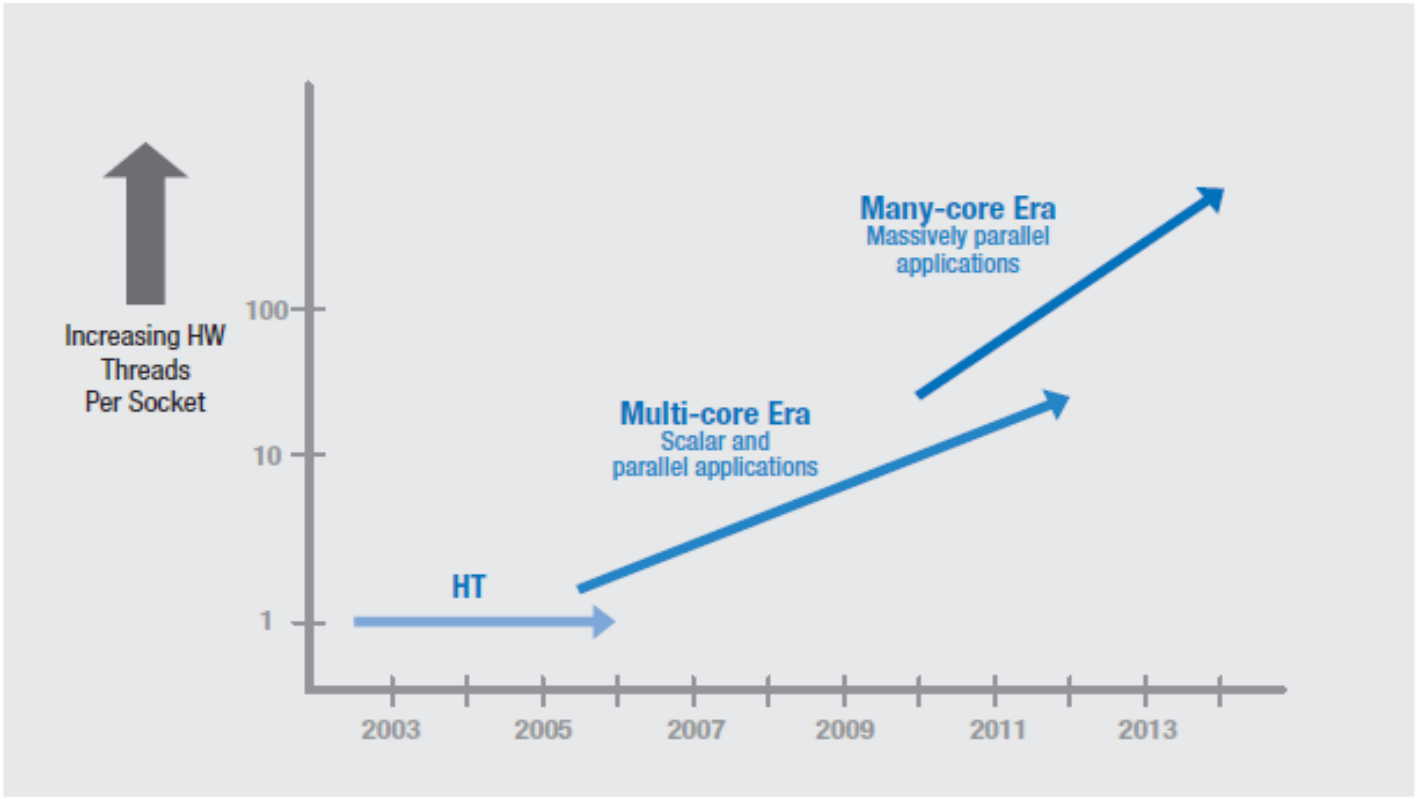


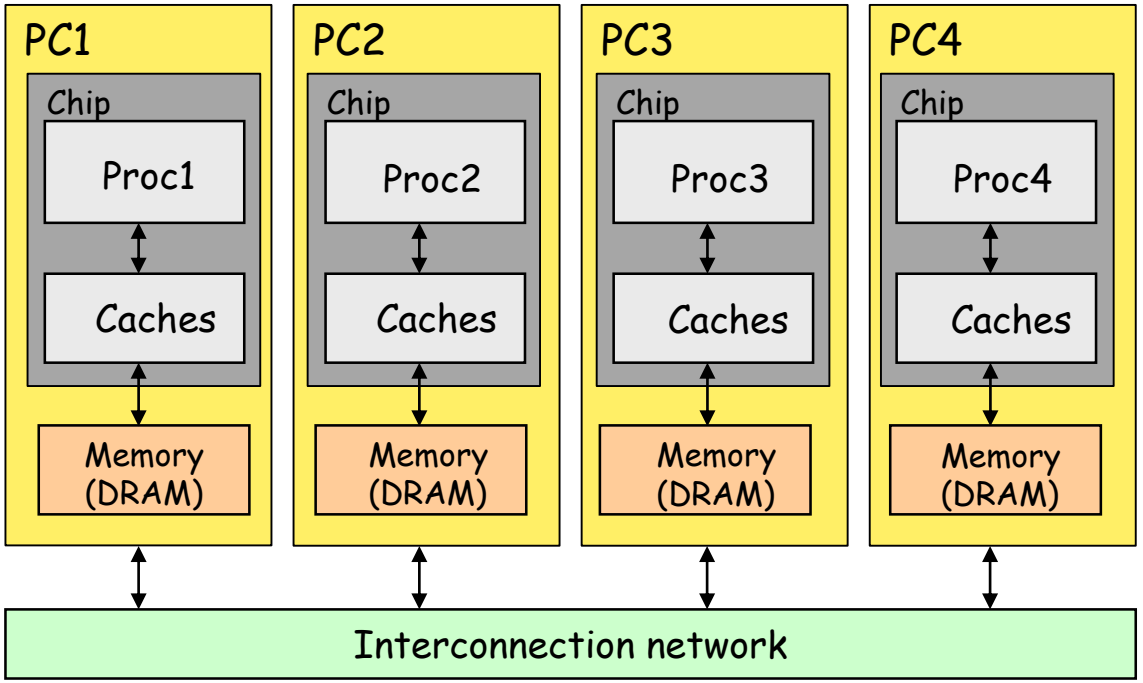
Figure 1: Current and expected eras of Intel® processor architectures

Platform 2015: Intel® Processor and Platform Evolution for the Next Decade, 2005

Distributed Memory Multi-Processor Architecture

- Multi-processor or multicore computers can be classified into distributed memory or shared memory architectures.
- A PC cluster or parallel computers for higher performance.Each memory module is associated with a processor
- Using explicit send and receive functions (message passing) to obtain the data required.
  - Who will send and receive data? How?

PC cluster



Cell Broadband Engine (2005)

- Cell Broadband Engine
  - 1 core (PPE, Power Processor Element based on a general purpose PowerPC core)
  - 8 core (SPE, Synergistic Processing Element)
    - each SPE has 256KB local memory
  - PS3, IBM Roadrunner with 12,960 CBE chips

Thie photo from PlaySation.com (Japan)

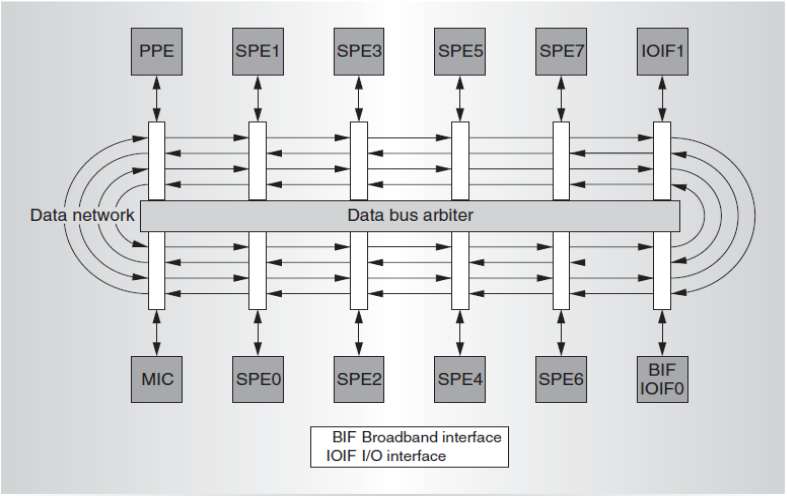


Figure 2. Element interconnect bus (EIB).

IEEE Micro, Cell Multiprocessor Communication Network: Built for Speed

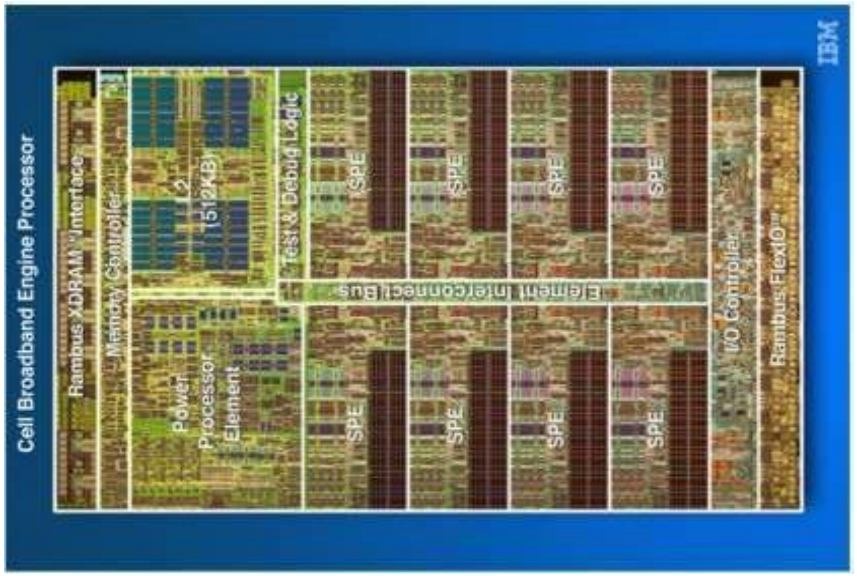
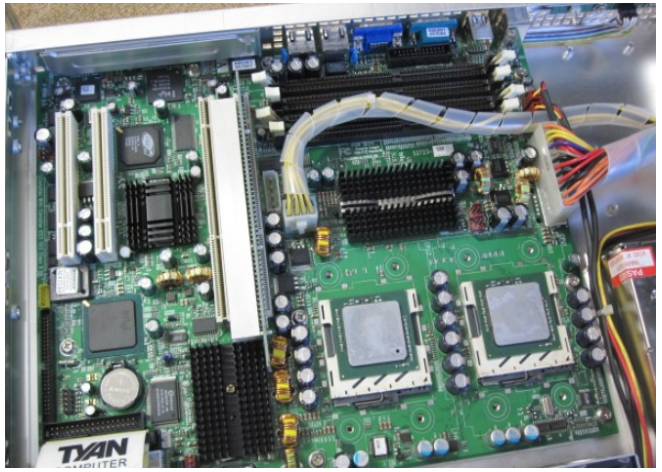
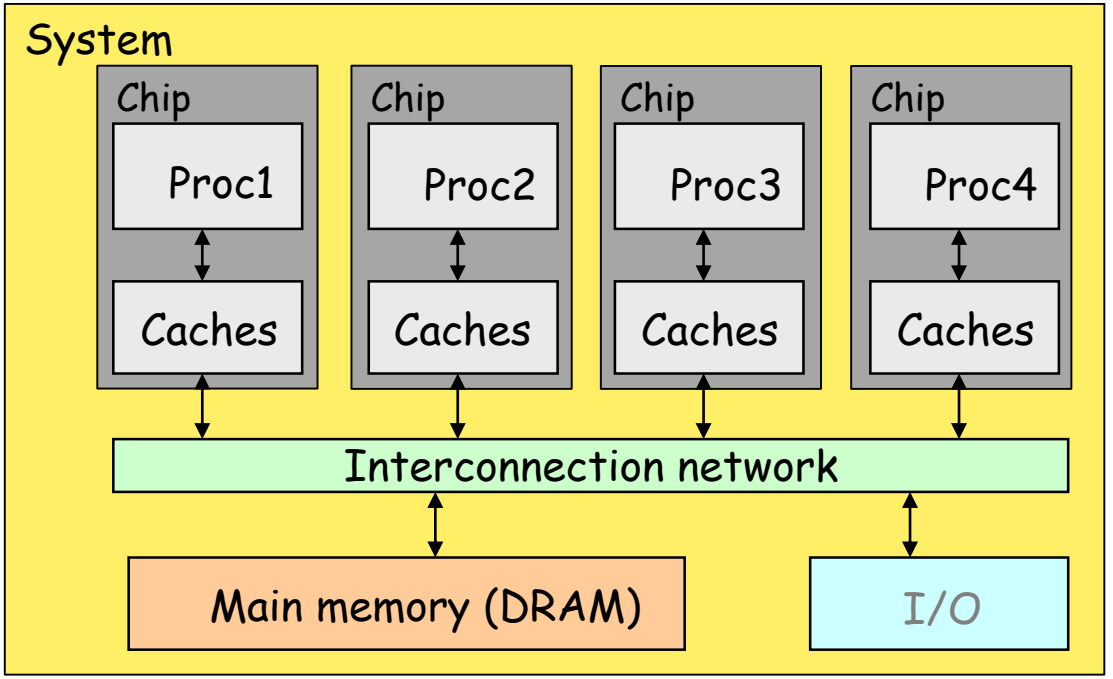


Diagram created by IBM to promote the CBEP, ©2005 from WIKIPEDIA

Shared Memory Multi-Processor Architecture

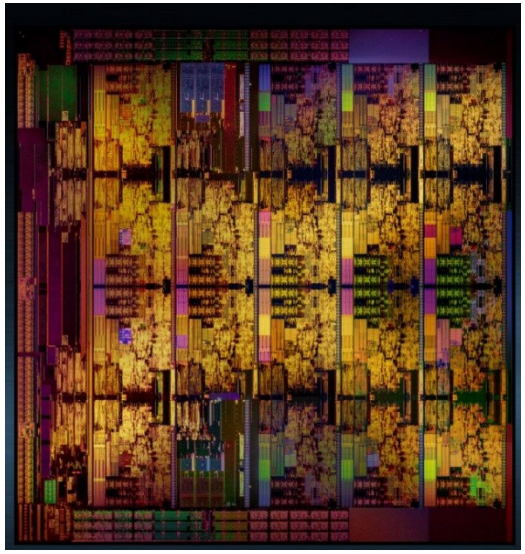
- All the processors can access the same address space of the main memory (shared memory) through an interconnection network.
- The shared memory or shared address space (SAS) is used as a means for communication between the processors.
  - What are the means to obtain the shared data?
  - What are the advantages and disadvantages of shared memory?



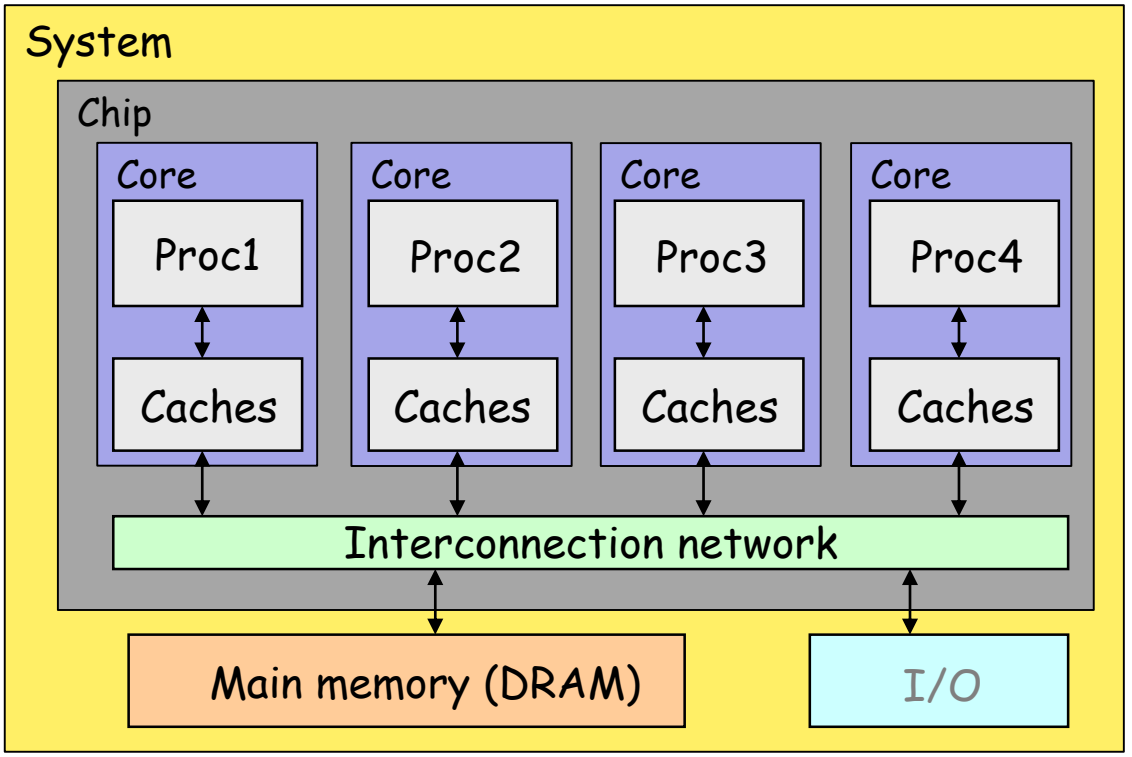


Shared memory many-core architecture

- A single-chip integrates many cores (conventional processors) and an interconnection network.
- The shared memory and shared address space (SAS) are used as a means for communication between these cores.



Intel Skylake-X, Core i9-7980XE, 2017



2022.11 AMD EPYC 9654 processor with 96 cores

- Today's high-performance chip integrates around 100 cores.

AMD EPYC™ 9004 Series Processor

All-in Feature Set support

- 12 Channels of DDR5-4800
- Up to 6TB DDR5 memory capacity
- 128 lanes PCIe® 5
- 64 lanes CXL 1.1+
- AVX-512 ISA, SMT & core frequency boost
- AMD Infinity Fabric™
- AMD Infinity Guard

Cores	AMD EPYC	Base/Boost* (up to GHz)	Default TDP (w)	cTDP (w)
96 cores	9654/P	2.40/3.70	360w	320-400w
84 cores	9634	2.25/3.70	290w	240-300w
64 cores	9554/P	3.10/3.75	360w	320-400w
64 cores	9534	2.45/3.70	280w	240-300w
48 cores	→ 9474F	3.60/4.10	360w	320-400w
	9454/P	2.75/3.80	290w	240-300w
32 cores	→ 9374F	3.85/4.30	320w	320-400w
32 cores	9354/P	3.25/3.80	280w	240-300w
32 cores	9334	2.70/3.90	210w	200-240w
24 cores	→ 9274F	4.05/4.30	320w	320-400w
	9254	2.90/4.15	200w	200-240w
	9224	2.50/3.70	200w	200-240w
16 cores	→ 9174F	4.10/4.40	320w	320-400w
	9124	3.00/3.70	200w	200-240w

The free lunch is over

- Programmers have to worry much about performance and concurrency
  - Parallel programming & multi-processor (multi-core) architectures

Free Lunch

Programmers haven't really had to worry much about performance or concurrency because of Moore's Law

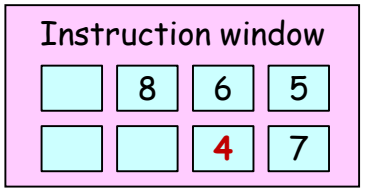
Why we did not see 4GHz processors in Market?

The traditional approach to application performance was to simply wait for the next generation of processor; most software developers did not need to invest in performance tuning, and enjoyed a "free lunch" from hardware improvements.

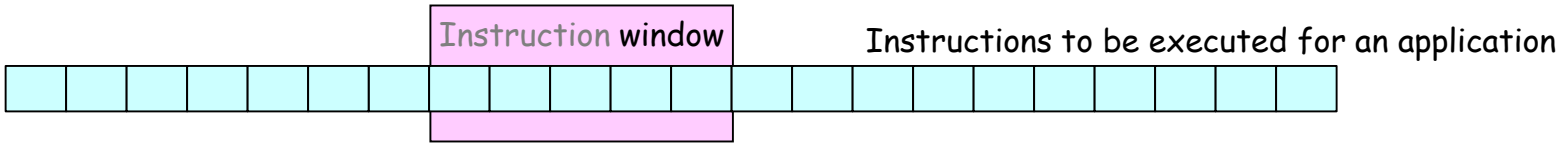
The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software by Herb Sutter, 2005

Multiprogramming

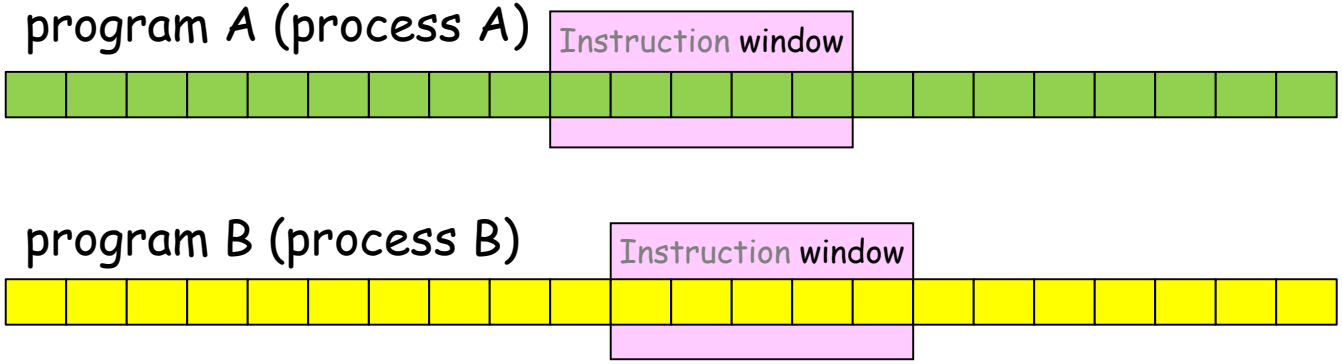
- Several independent programs (processes) run at the same time on a multi-core processor (multi-processor system).



(a)

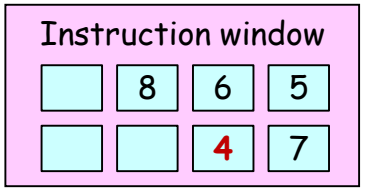


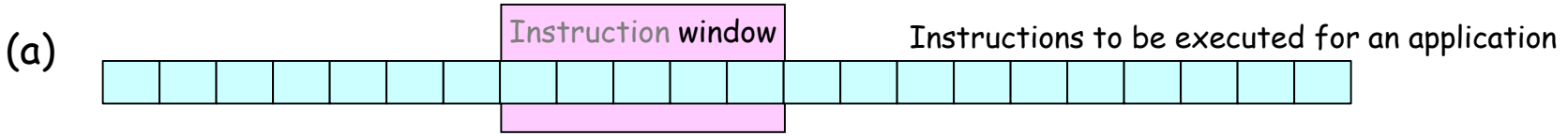
(d)

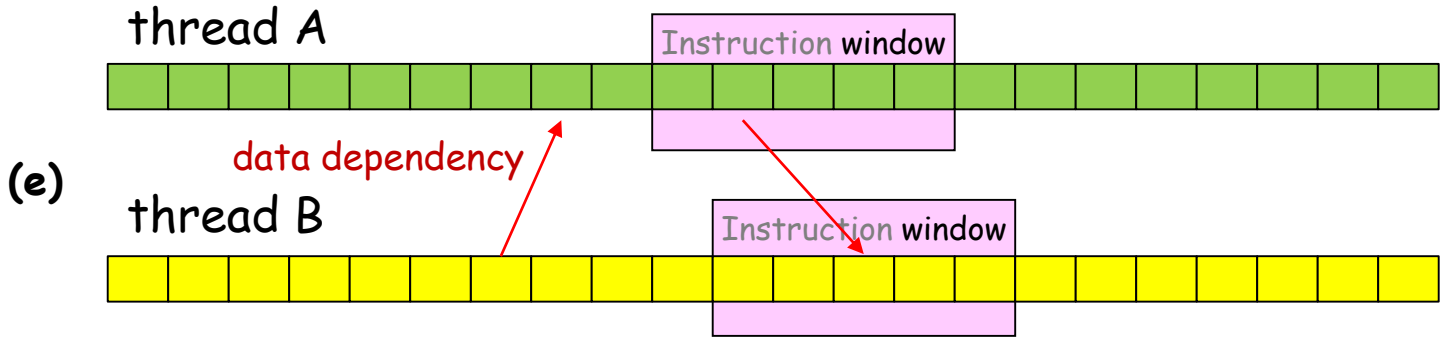


Parallel programming

- Several dependent threads run at the same time on a multi-core processor (multi-processor system).
- This is the case that is often required.







Sample of a wrong parallel program using pthread

```
% gcc main1.c -O0 -lpthread -lm -o a.out1
% ./a.out1
main: 20000000
```

Single Program Multiple Data (SPMD)

```c
#include <stdio.h>
#include <pthread.h>
#define N 10000000

int a = 0;

int func1(){
  int i;
  for(i=0; i<N; i++){a = a + 1;}
};

int func2(){
  int i;
  for(i=0; i<N; i++){a = a + 1;}
};

int main(){
  func1();
  func2();

  printf("main: %d\n", a);
  return 0;
}
```

main1.c
sequential program

```c
#include <stdio.h>
#include <pthread.h>
#define N 10000000    // ten million

int a = 0;

int func1(){
  int i;
  for(i=0; i<N; i++){a = a + 1;}
};

int func2(){
  int i;
  for(i=0; i<N; i++){a = a + 1;}
};

int main(){
  pthread_t t1, t2;
  pthread_create(&t1,NULL,(void *)func1,NULL);
  pthread_create(&t2,NULL,(void *)func2,NULL);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  printf("main: %d\n", a);
  return 0;
}
```

main2.c
parallel program with func1 and func2

```c
#include <stdio.h>
#include <pthread.h>
#define N 10000000    // ten million

int a = 0;

int func1(){
  int i;
  for(i=0; i<N; i++){a = a + 1;}
};

int main(){
  pthread_t t1, t2;
  pthread_create(&t1,NULL,(void *)func1,NULL);
  pthread_create(&t2,NULL,(void *)func1,NULL);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  printf("main: %d\n", a);
  return 0;
}
```

main3.c
parallel program with func1

Sample of some parallel programs using pthread

```
% gcc main1.c -O0 -lpthread -lm -o a.out1
% ./a.out1
main: 20000000
```

```c
#include <stdio.h>
#include <pthread.h>
#define N 10000000

int a = 0;

int func1(){
  int i;
  for(i=0; i<N; i++){a = a + 1;}
};

int func2(){
  int i;
  for(i=0; i<N; i++){a = a + 1;}
};

int main(){
  func1();
  func2();

  printf("main: %d\n", a);
  return 0;
}
```

main1.c
sequential program

```c
#include <stdio.h>
#include <pthread.h>
#define N 10000000    // ten million

int a = 0;

int func1(){
  int i;
  for(i=0; i<N; i++){a = a + 1;}
};

int main(){
  pthread_t t1, t2;
  pthread_create(&t1,NULL,(void *)func1,NULL);
  pthread_create(&t2,NULL,(void *)func1,NULL);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  printf("main: %d\n", a);
  return 0;
}
```

main3.c
parallel program with func1

```c
#include <stdio.h>
#include <pthread.h>
#define N 10000000    // ten million

int a = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

int func1(){
  int i;
  for(i=0; i<N; i++){
    pthread_mutex_lock(&m);
    a = a + 1;
    pthread_mutex_unlock(&m);
  }
};

int main(){
  pthread_t t1, t2;
  pthread_create(&t1,NULL,(void *)func1,NULL);
  pthread_create(&t2,NULL,(void *)func1,NULL);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  printf("main: %d\n", a);
  return 0;
}
```

main4.c
parallel program with func1, lock, and unlock

Sample of some parallel programs using pthread

```c
#include <stdio.h>
#include <pthread.h>
#define N 10000000    // ten million

int a = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

int func1(){
  int i;
  for(i=0; i<N; i++){
    pthread_mutex_lock(&m);
    a = a + 1;
    pthread_mutex_unlock(&m);
  }
};

int main(){
  pthread_t t1, t2;
  pthread_create(&t1,NULL,(void *)func1,NULL);
  pthread_create(&t2,NULL,(void *)func1,NULL);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  printf("main: %d\n", a);
  return 0;
}
```

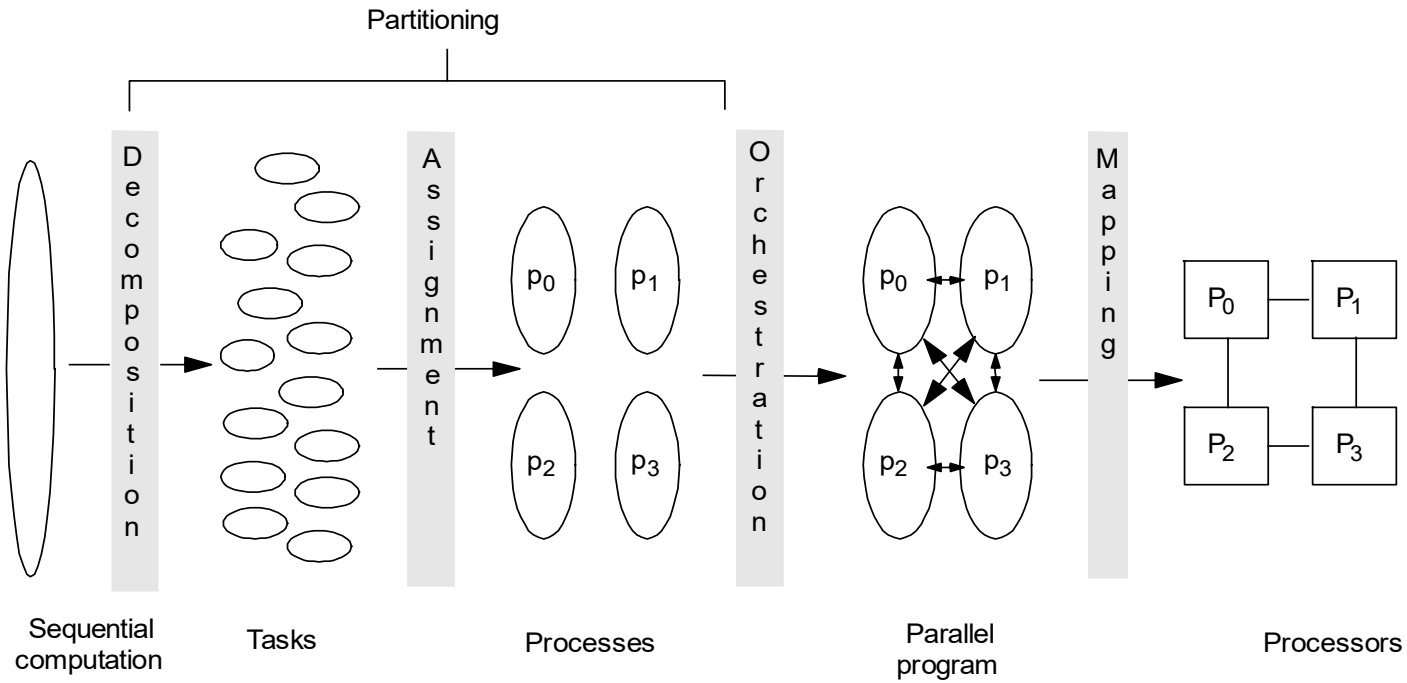
main4.c
parallel program with func1, lock, and unlock

```c
#include <stdio.h>
#include <pthread.h>
#define N 10000000    // ten million

int a = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

int func1(){
  int i;
  int my_a = 0;
  for(i=0; i<N; i++){
    my_a = my_a + 1;
  }
  pthread_mutex_lock(&m);
  a = a + my_a;
  pthread_mutex_unlock(&m);
};

int main(){
  pthread_t t1, t2;
  pthread_create(&t1,NULL,(void *)func1,NULL);
  pthread_create(&t2,NULL,(void *)func1,NULL);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  printf("main: %d\n", a);
  return 0;
}
```

main5.c
parallel program with func1, local sum, lock, and unlock

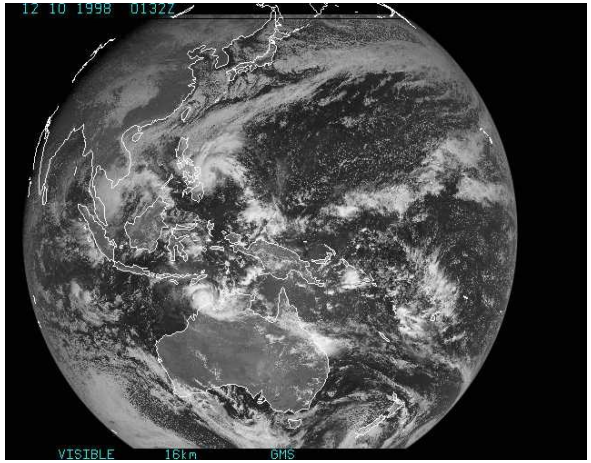
Four steps in creating a parallel program

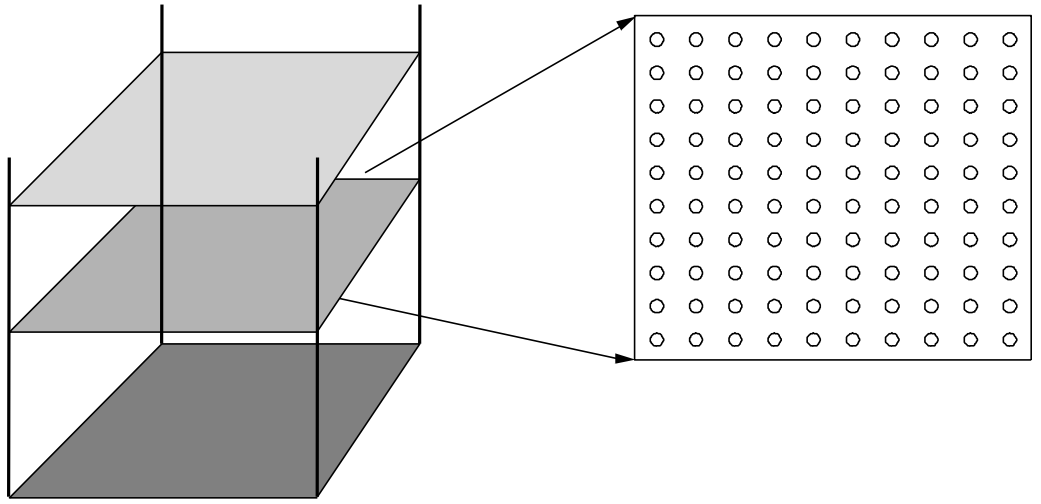
0. Preparing an optimized sequential program (baseline)
1. Decomposition of computation in tasks
2. Assignment of tasks to processes
3. Orchestration of data access, comm, synch.
4. Mapping processes to processors (cores)



Adapted from *Parallel Computer Architecture*, David E. Culler

Simulating ocean currents





(a) Cross sections

(b) Spatial discretization of a cross section

- Model as two-dimensional grids
  - Discretize in space and time
  - finer spatial and temporal resolution enables greater accuracy
- Many different computations per time step
  - Concurrency across and within grid computations
- We use one-dimensional grids for simplicity

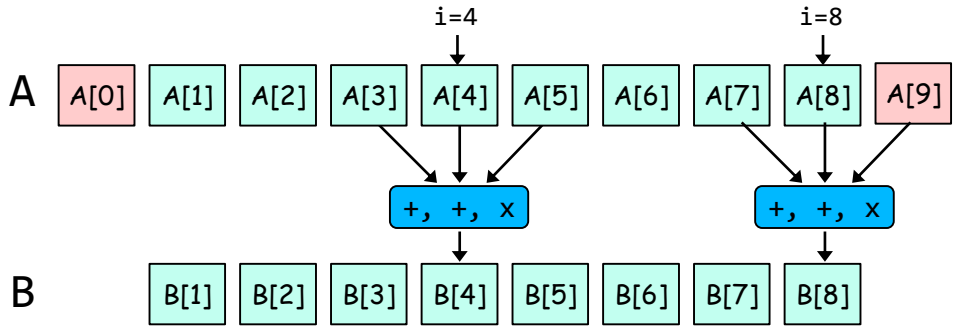
Sequential version as the baseline

- A sequential program main6.c and the execution result
- Computations in blue color are fully parallel

```
#define N 8      /* the number of grids */
#define TOL 15.0 /* tolerance parameter */
float A[N+2], B[N+2];

void solve () {
    int i, done = 0;
    while (!done) {
        float diff = 0.0;
        for (i=1; i<=N; i++) {
            B[i] = 0.333 * (A[i-1] + A[i] + A[i+1]);
            diff = diff + fabsf(B[i] - A[i]);
        }
        if (diff <TOL) done = 1;
        for (i=1; i<=N; i++) A[i] = B[i];

        for (i=0; i<=N+1; i++) printf("%6.2f ", B[i]);
        printf("| diff=%6.2f\n", diff); /* for debug */
    }
}

int main() {
    int i;
    for (i=1; i<N-1; i++) A[i] = 100+i*i; // initialize
    for (i=0; i<=N+1; i++) printf("%6.2f ", A[i]);
    printf("\n");
    solve();
}
```

main6.c sequential program

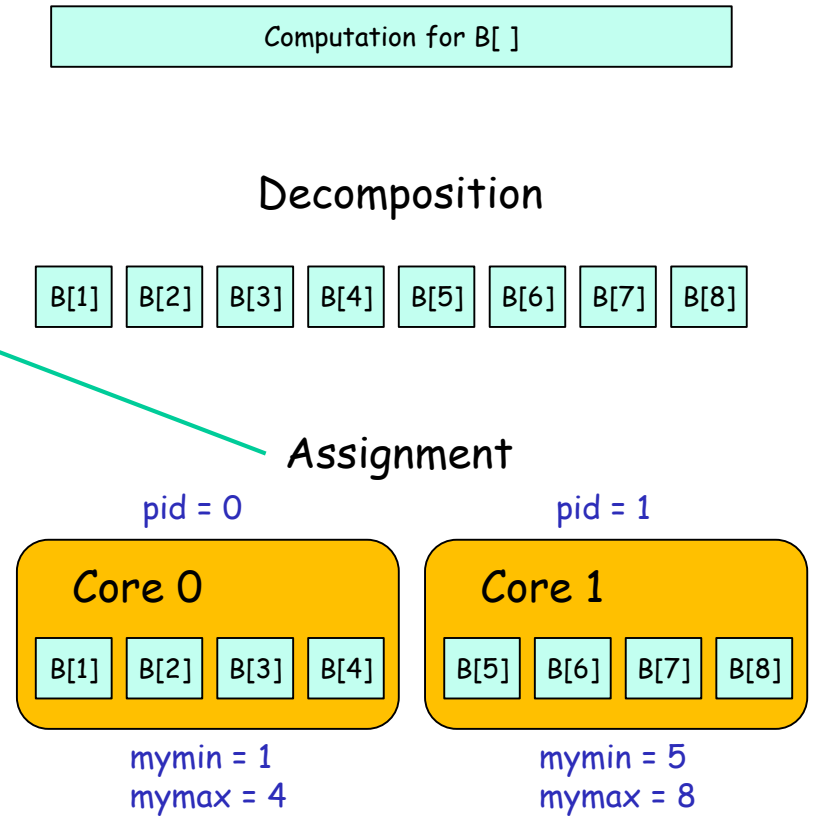
```
  0.00 101.00 104.00 109.00 116.00 125.00 136.00   0.00   0.00   0.00
  0.00  68.26 104.56 109.56 116.55 125.54  86.91  45.29   0.00   0.00 | diff=129.32
  0.00  57.55  94.03 110.11 117.10 109.56  85.83  44.02  15.08   0.00 | diff= 55.76
  0.00  50.48  87.15 106.97 112.14 104.06  79.72  48.26  19.68   0.00 | diff= 42.50
  0.00  45.83  81.45 101.99 107.62  98.54  77.27  49.17  22.63   0.00 | diff= 31.68
  0.00  42.38  76.35  96.92 102.61  94.38  74.92  49.64  23.91   0.00 | diff= 26.88
  0.00  39.54  71.81  91.87  97.87  90.55  72.91  49.44  24.49   0.00 | diff= 23.80
  0.00  37.08  67.67  87.10  93.34  87.02  70.89  48.90  24.62   0.00 | diff= 22.12
  0.00  34.88  63.89  82.62  89.06  83.67  68.87  48.09  24.48   0.00 | diff= 21.06
  0.00  32.89  60.40  78.44  85.03  80.45  66.81  47.10  24.17   0.00 | diff= 20.26
  0.00  31.07  57.19  74.55  81.23  77.35  64.72  45.98  23.73   0.00 | diff= 19.47
  0.00  29.39  54.21  70.92  77.63  74.36  62.62  44.77  23.21   0.00 | diff= 18.70
  0.00  27.84  51.46  67.52  74.23  71.47  60.52  43.49  22.64   0.00 | diff= 17.95
  0.00  26.41  48.89  64.34  71.00  68.67  58.43  42.17  22.02   0.00 | diff= 17.23
  0.00  25.07  46.50  61.35  67.94  65.97  56.37  40.84  21.38   0.00 | diff= 16.53
  0.00  23.83  44.26  58.54  65.02  63.36  54.34  39.49  20.72   0.00 | diff= 15.85
  0.00  22.68  42.17  55.88  62.24  60.85  52.34  38.14  20.05   0.00 | diff= 15.20
  0.00  21.59  40.20  53.38  59.60  58.42  50.39  36.81  19.38   0.00 | diff= 14.58
```



Decomposition and assignment

- Single Program Multiple Data (SPMD)
  - Decomposition: there are eight tasks to compute B[ ]
  - Assignment: the first four tasks for core 0, and the last four tasks for core 1

```
float A[N+2], B[N+2]; /* these are in shared memory */
float diff=0.0;       /* variable  in shared memory */


void solve_pp (int pid) {
    int i, done = 0;                    /* private variables */
    int mymin = (pid==0) ? 1 : 5;       /* private variable  */
    int mymax = (pid==0) ? 4 : 8;       /* private variable  */
    while (!done) {
        float mydiff = 0;
        for (i=mymin; i<=mymax; i++) {
            B[i] = 0.333 * (A[i-1] + A[i] + A[i+1]);
            mydiff = mydiff + fabsf(B[i] - A[i]);
        }
        diff = diff + mydiff;

        if (diff<TOL) done = 1;
        if (pid==1) diff = 0.0;
        for (i=mymin; i<=mymax; i++) A[i] = B[i];
    }
}

int main() { /* solve this using two cores */
    initialize shared data A and B;
    create thread1 and call solve_pp(0);
    create thread2 and call solve_pp(1);
}
```



Orchestration

- LOCK and UNLOCK around critical section
  - Lock provides exclusive access to the locked data.
  - Set of operations we want to execute atomically
- BARRIER ensures all reach here

```
float A[N+2], B[N+2]; /* these are in shared memory */
float diff=0.0;       /* variable  in shared memory */

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_barrier_t barrier;
void solve_pp (int pid) {
    int i, done = 0;                    /* private variables */
    int mymin = (pid==0) ? 1 : 5;       /* private variable  */
    int mymax = (pid==0) ? 4 : 8;       /* private variable  */
    while (!done) {
        float mydiff = 0;
        for (i=mymin; i<=mymax; i++) {
            B[i] = 0.333 * (A[i-1] + A[i] + A[i+1]);
            mydiff = mydiff + fabsf(B[i] - A[i]);
        }
        pthread_mutex_lock(&m);
        diff = diff + mydiff;
        pthread_mutex_unlock(&m);

        pthread_barrier_wait(&barrier);  // Barrier 1
        if (diff <TOL) done = 1;
        pthread_barrier_wait(&barrier);  // Barrier 2
        if (pid==1) diff = 0.0;
        for (i=mymin; i<=mymax; i++) A[i] = B[i];
        pthread_barrier_wait(&barrier);  // Barrier 3
    }
}
```

These operations must be executed atomically

```
(1) load diff
(2) add
(3) store diff
```

After all cores update the diff, *if statement* must be executed.

```
if (diff <TOL) done = 1;
```

Parallel program after orchestration

```
% gcc main7.c -O0 -lpthread -lm -o a.out7
```

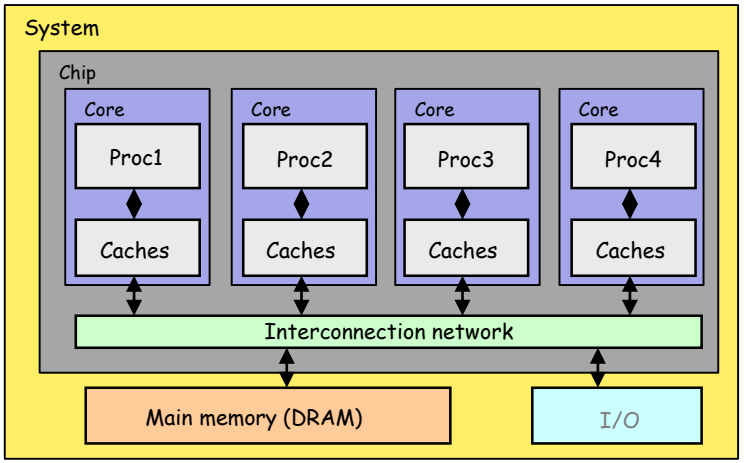
```
#include <stdio.h>
#include <math.h>
#include <pthread.h>
#define N 8           /* the number of grids */
#define TOL 15.0      /* tolerance parameter */

float A[N+2], B[N+2]; /* these are in shared memory */
float diff=0.0;       /* variable  in shared memory */
int ncores = 2;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_barrier_t barrier;

int main(){
    pthread_t t1, t2;
    int pid0 = 0;
    int pid1 = 1;
    for (int i=1; i<N-1; i++) A[i] = 100+i*i;

    pthread_barrier_init(&barrier, NULL, ncores);
    pthread_create(&t1, NULL, (void *)solve_pp, (void*)&pid0);
    pthread_create(&t2, NULL, (void *)solve_pp, (void*)&pid1);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    for (int i=0; i<=N+1; i++) printf("%6.2f ", B[i]);
    printf("\n");
    return 0;
}
```

```
void solve_pp (void *p) {
    int pid = *(int *)p;
    int i, done = 0;                    /* private variables */
    int mymin = 1 + (pid * N/ncores);   /* private variable  */
    int mymax = mymin + N/ncores - 1;   /* private variable  */
    while (!done) {
        float mydiff = 0.0;
        for (i=mymin; i<=mymax; i++) {
            B[i] = 0.333 * (A[i-1] + A[i] + A[i+1]);
            mydiff = mydiff + fabsf(B[i] - A[i]);
        }
        pthread_mutex_lock(&m);
        diff = diff + mydiff;
        pthread_mutex_unlock(&m);

        pthread_barrier_wait(&barrier);
        if (diff <TOL) done = 1;
        pthread_barrier_wait(&barrier);
        if (pid==1) diff = 0.0;
        for (i=mymin; i<=mymax; i++) A[i] = B[i];
        pthread_barrier_wait(&barrier);
    }
}
```

main7.c parallel program

Key components of many-core processors

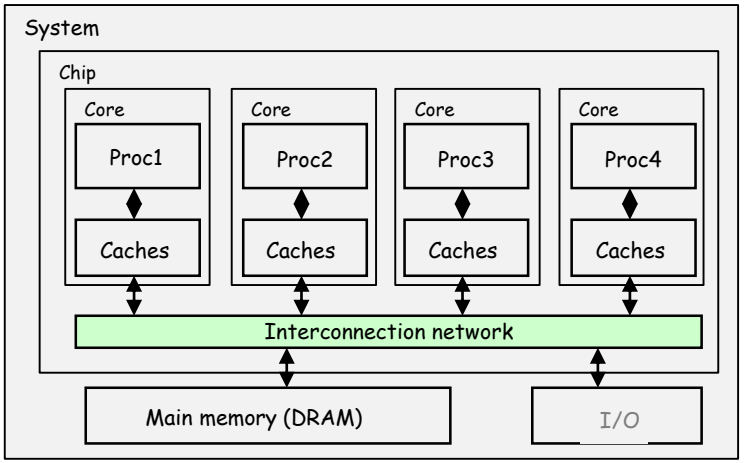
- Interconnection network
  - connecting many modules on a chip achieving high throughput and low latency
- Main memory and caches
  - Caches are used to reduce latency and to lower network traffic
  - A parallel program has private data and shared data
  - New issues are cache coherence and memory consistency
- Core
  - High-performance superscalar processor providing a hardware mechanism to support thread synchronization (lock, unlock, barrier)



Shared memory many-core architecture

Key components of many-core processors

- Interconnection network
  - connecting many modules on a chip achieving high throughput and low latency
- Main memory and caches
  - Caches are used to reduce latency and to lower network traffic
  - A parallel program has private data and shared data
  - New issues are cache coherence and memory consistency
- Core
  - High-performance superscalar processor providing a hardware mechanism to support thread synchronization (lock, unlock, barrier)



Shared memory many-core architecture

On-Chip Interconnection network requirements

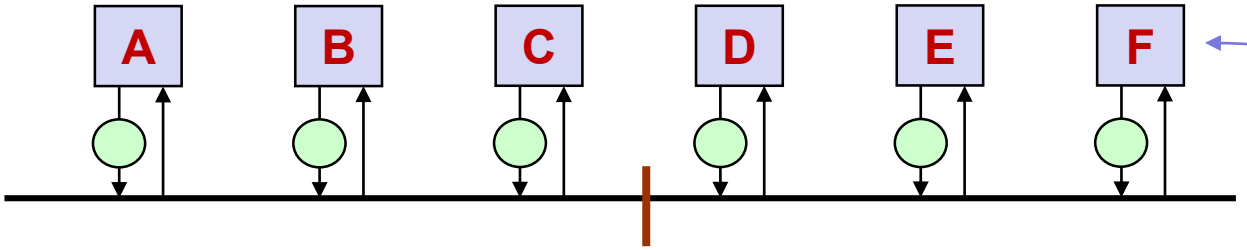
- Connecting many modules on a chip achieving high throughput and low latency
  - Topology
    - the number of ports, links, switches (HW resources)
    - bus, ring bus, tree, fat-tree, crossbar, mesh, torus
  - Circuit switching / packet switching
    - Centralized control / distributed control with FIFO and flow control (scalability)
  - Routing
    - deadlock free, livelock free
    - in-order data delivery / out-of-order delivery
    - adaptive routing, XY-dimension order routing
  - Network-on-chip (NoC) router architecture

Performance metrics of interconnection network

- Network cost
  - number of links on a switch to connect to the network (plus one link to connect to the processor)
  - width in bits per link, length of link
- Network bandwidth (NB)
  - represents the best case
  - bandwidth of each link x number of links
- Bisection bandwidth (BB)
  - represents the worst case
  - divide the machine in two parts, each with half the nodes and sum the bandwidth of the links that cross the dividing line

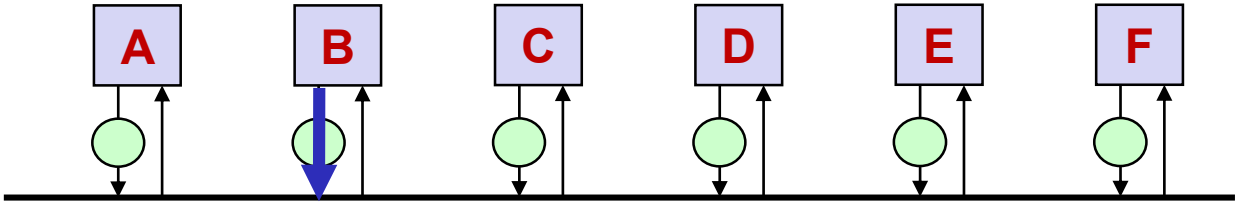
Bus Network

- N cores (□), N switch (○), 1 link (the bus)
- Only 1 simultaneous transfer at a time
  - NB (best case) = link (bus) bandwidth x 1
  - BB (worst case) = link (bus) bandwidth x 1
- All processors can snoop the bus



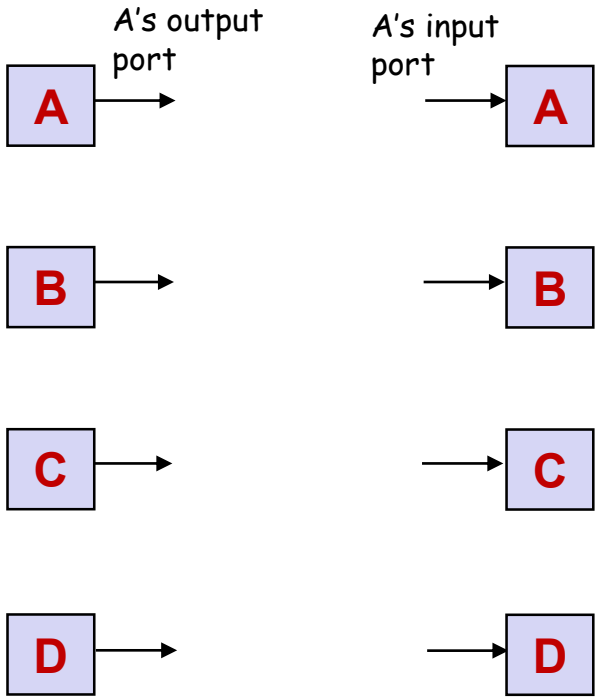
Core or processor node

The case where core B sends a packet to someone



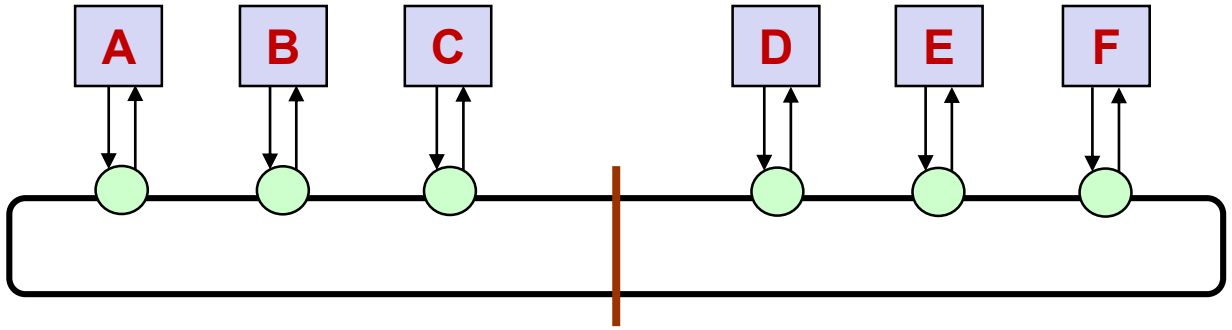
Exercise 1

- Bus Network with multiplexer (mux)
- one N-input mux for N cores
- Draw the bus network organization of 4 cores using a 4-input mux.

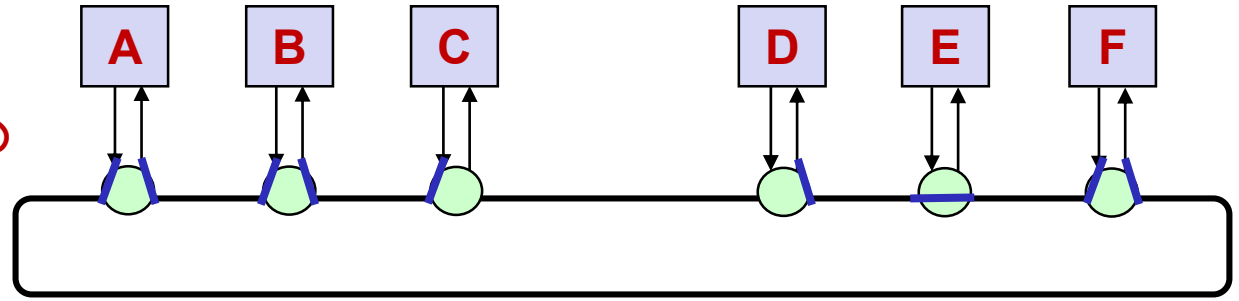


Ring Network

- N cores, N switches, 2 links/switch, N links
- N simultaneous transfers
  - NB (best case) = link bandwidth x N
  - BB (worst case) = link bandwidth x 2
- If a link is as fast as a bus, the ring is only twice as fast as a bus in the worst case, but is N times faster in the best case



The case where
A -> F, B->A, C->B, F->D



Cell Broadband Engine (2005)

- Cell Broadband Engine
  - 1 core (PPE, Power Processor Element based on a general purpose PowerPC core)
  - 8 core (SPE, Synergistic Processing Element)
    - each SPE has 256KB local memory
  - PS3, IBM Roadrunner with 12,960 CBE chips

Thie photo from PlaySation.com (Japan)

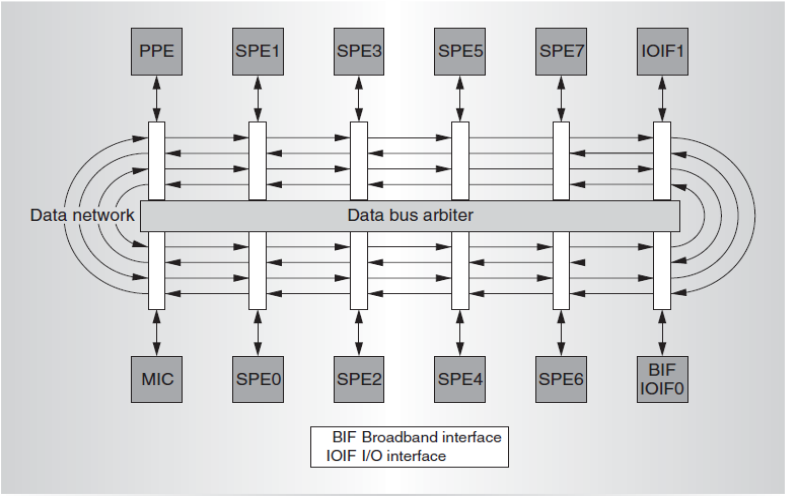


Figure 2. Element interconnect bus (EIB).

IEEE Micro, Cell Multiprocessor Communication Network: Built for Speed

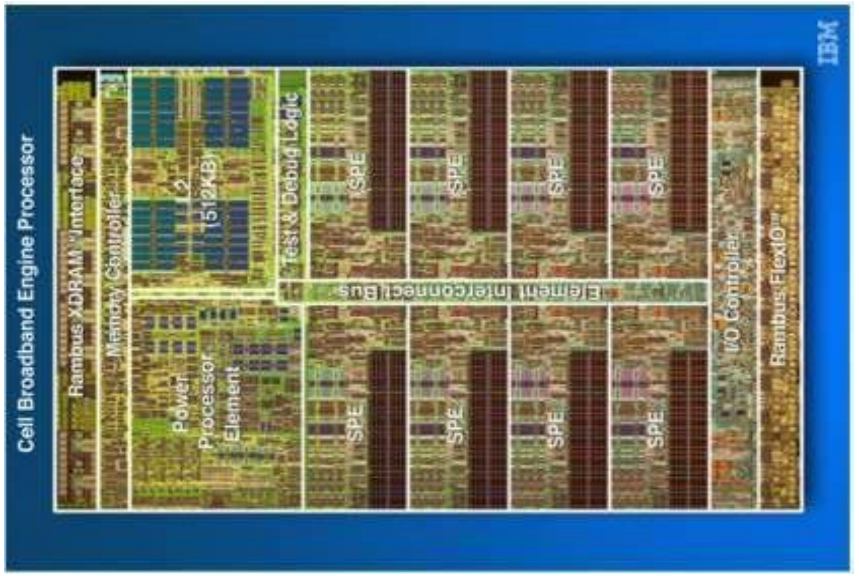


Diagram created by IBM to promote the CBEP, ©2005 from WIKIPEDIA

Intel Xeon Phi (2012)

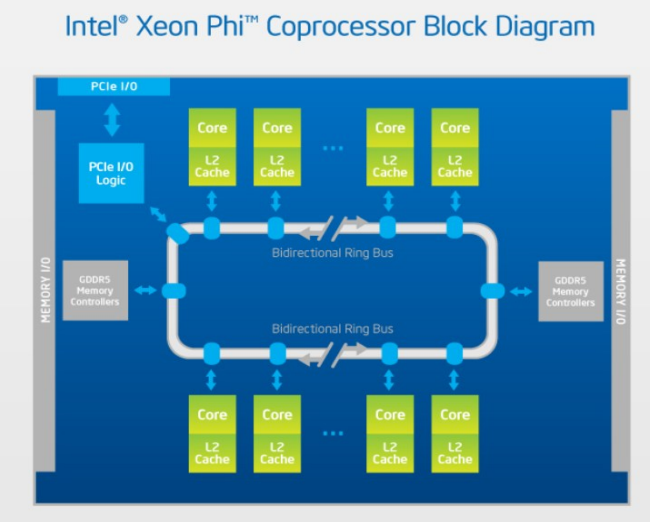
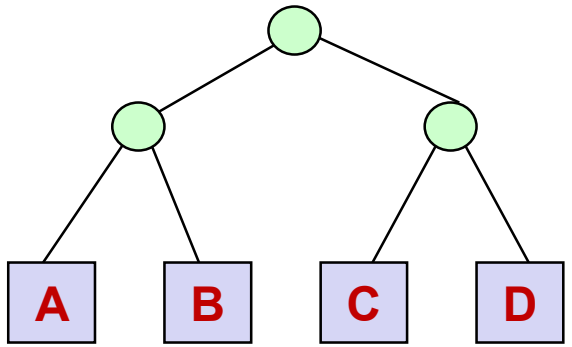


Table 2. Intel® Xeon Phi™ Product Family Specifications

PRODUCT NUMBER	FORM FACTOR &, THERMAL SOLUTION[4]	BOARD TDP (WATTS)	NUMBER OF CORES	FREQUENCY (GHz)	PEAK DOUBLE PRECISION PERFORMANCE (GFLOP)	PEAK MEMORY BANDWIDTH (GB/s)	MEMORY CAPACITY (GB)	INTEL® TURBO BOOST TECHNOLOGY
3120P	PCIe, Passive	300	57	1.1	1003	240	6	N/A
3120A	PCIe, Active	300	57	1.1	1003	240	6	N/A
5110P	PCIe, Passive	225	60	1.053	1011	320	8	N/A
5120D	Dense form factor, None	245	60	1.053	1011	352	8	N/A
7110P	PCIe, Passive	300	61	1.238	1208	352	16	Peak turbo frequency: 1.33 GHz
7120X	PCIe, None	300	61	1.238	1208	352	16	

Fat Tree (1)

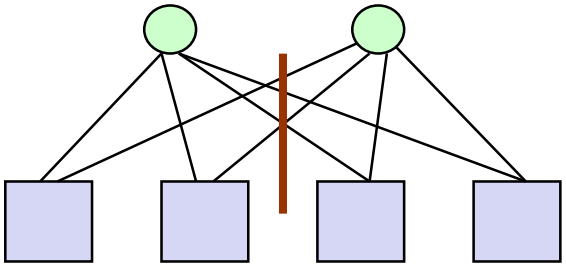
- Trees are good structures. People in CS use them all the time.
- Any time A wants to send to C, it ties up the upper links, so that B can't send to D.
  - The bisection bandwidth on a tree is horrible just 1 link, at all times
- The solution is to 'thicken' the upper links.
  - More links as the tree gets thicker increases the bisection bandwidth



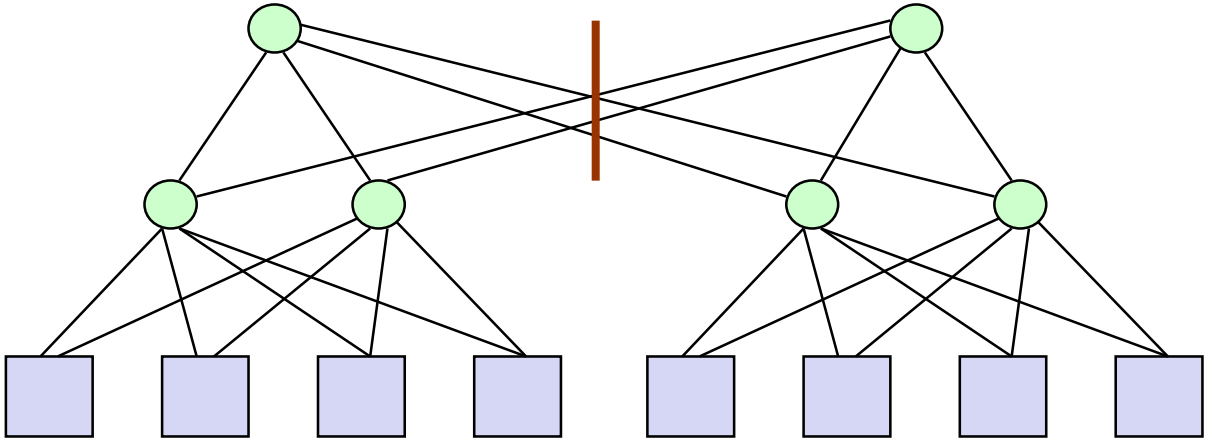
Tree, N = 4

Fat Tree (2)

- N cores, log(N-1) x logN switches, 2 up + 4 down = 6 links/switch, N x logN links
- N simultaneous transfers
  - NB = link bandwidth x N log N
  - BB = link bandwidth x 4



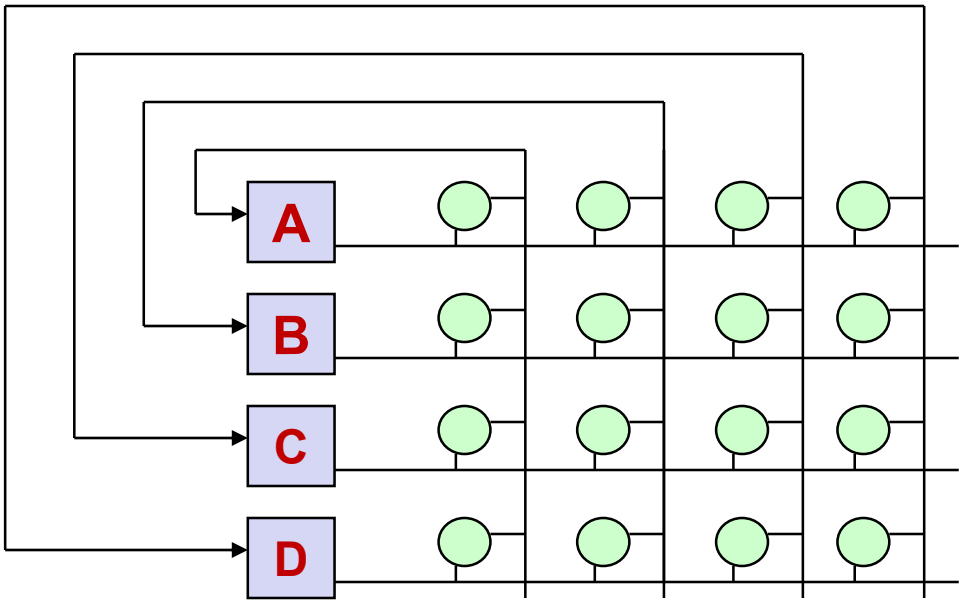
Fat Tree, N = 4

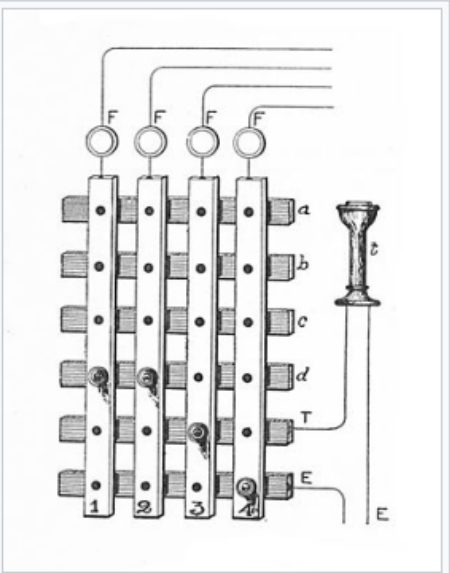


Fat Tree, N = 8

Crossbar (Xbar) Network

- N cores, N^2 switches (unidirectional), 2 links/switch, N^2 links
- N simultaneous transfers
  - NB = link bandwidth x N (best case)
  - BB = link bandwidth x N (worst case)



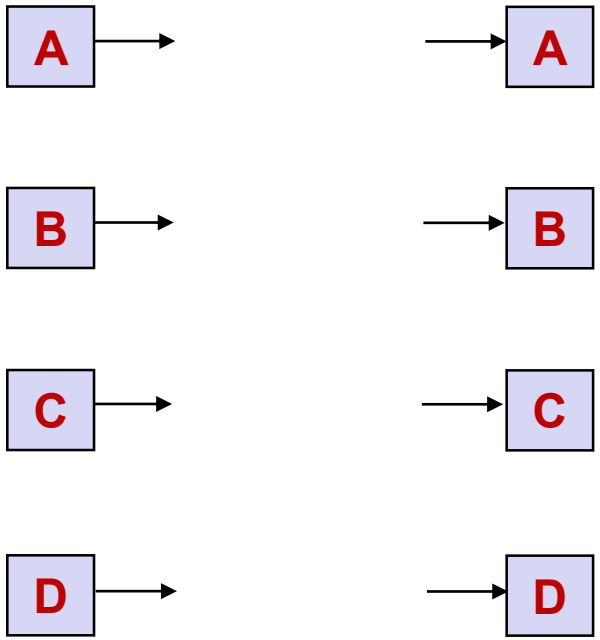


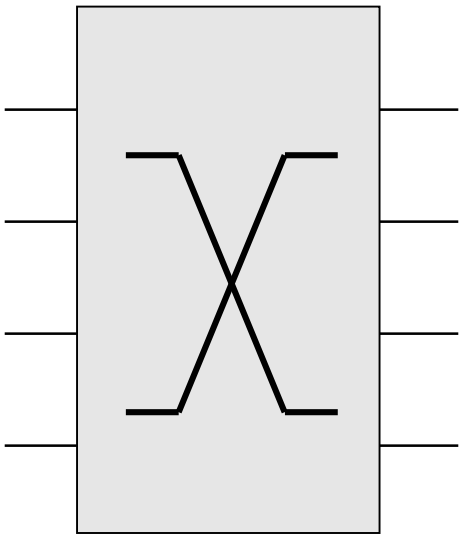
Crossbar telephone exchange of1903 for four subscribers (vertical bars), having four cross-bar talking circuits (horizontal bars), and one bar to connect the operator (T). The lowest cross-bar connects idle stations to ground to enable the signaling indicators (F). The switch is operated manually with metal pins that create a connection between the horizontally and vertically arranged bars.[1]

Wikipedia

Exercise 2

- Crossbar Network with multiplexer (mux)
- N N-input mux for N cores
- Draw the crossbar network organization of 4 cores using four 4-input muxs.

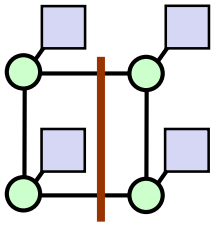




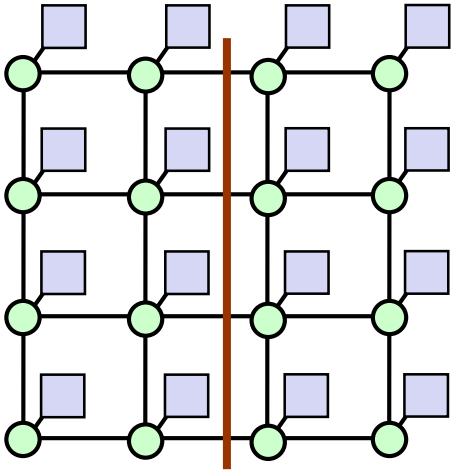
A symbol of Xbar

Mesh Network

- N cores, N switches, 5 links/switch
- N simultaneous transfers
  - NB = link bandwidth x N (best case)
  - BB = link bandwidth x $N^{1/2}$ (worst case)

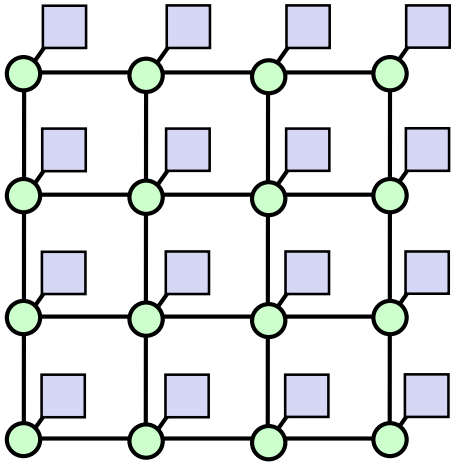


N = 4

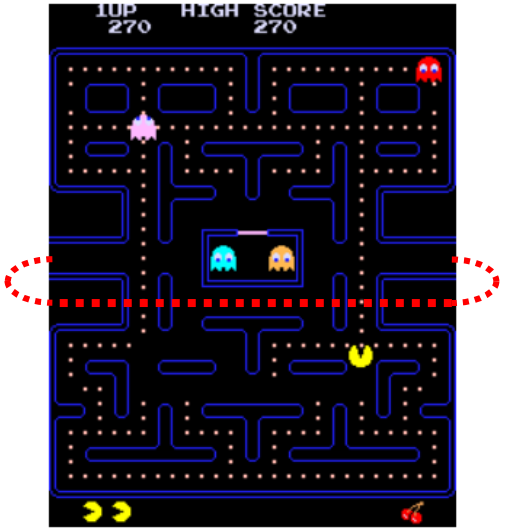


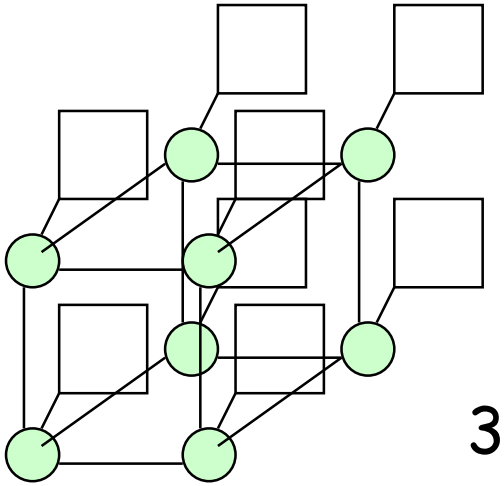
N = 16

2D and 3D Mesh / Torus Network

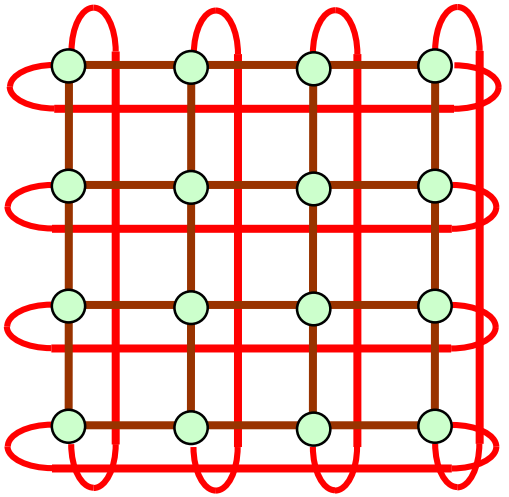


2D Mesh





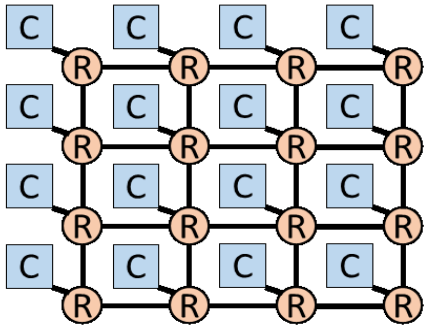
3D Mesh



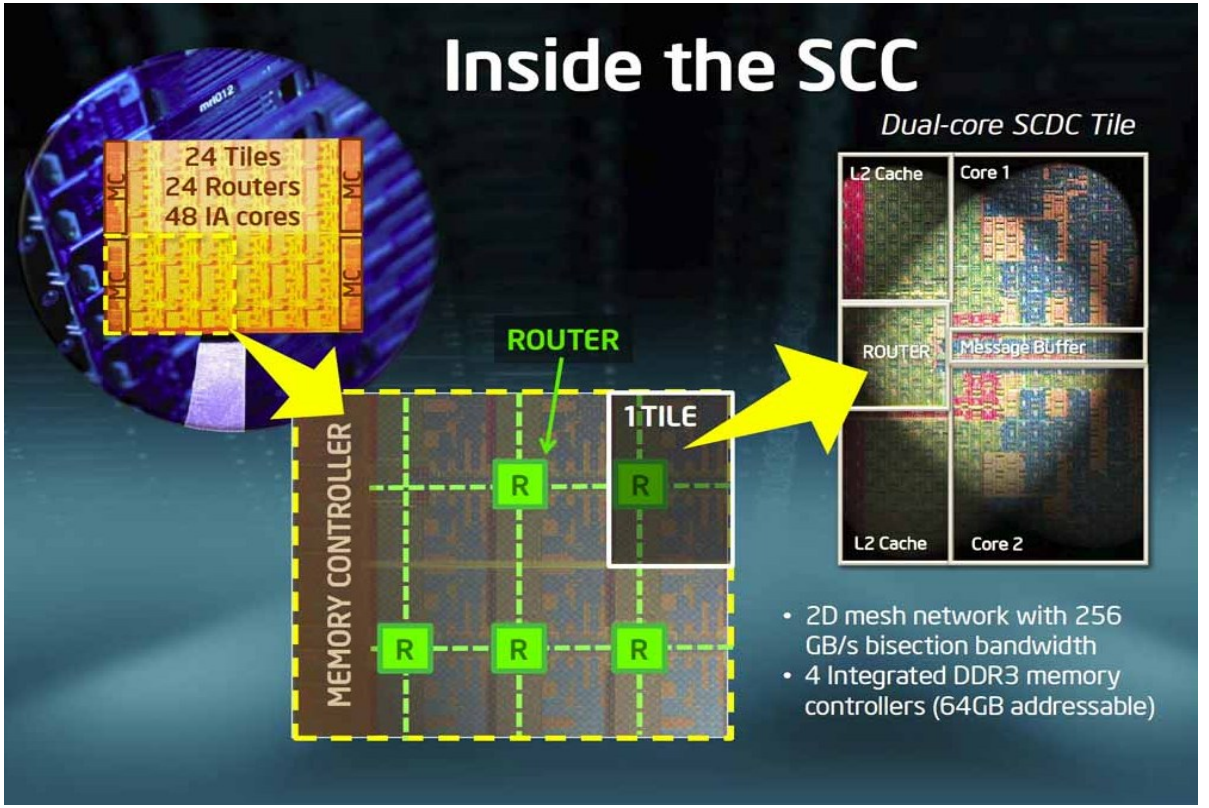
Torus

Intel Single-Chip Cloud Computer (2009)

- To research multi-core processors and parallel processing.

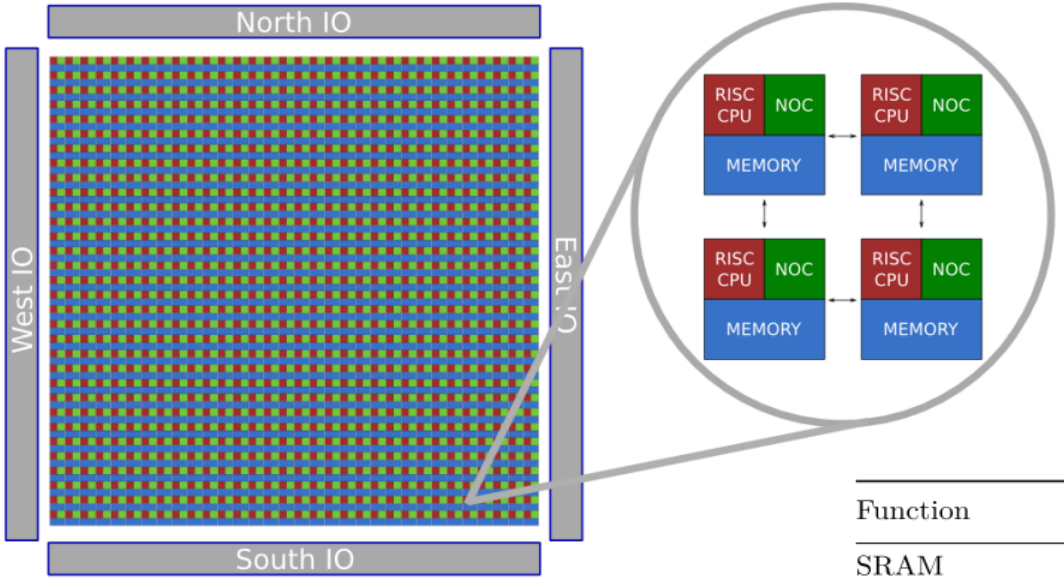


A many-core architecture with 2D Mesh NoC



Intel Single-Chip Cloud Computer (48 Core)

Epiphany-V: A 1024 core 64-bit RISC SoC (2016)



Summary of Epiphany-V features:

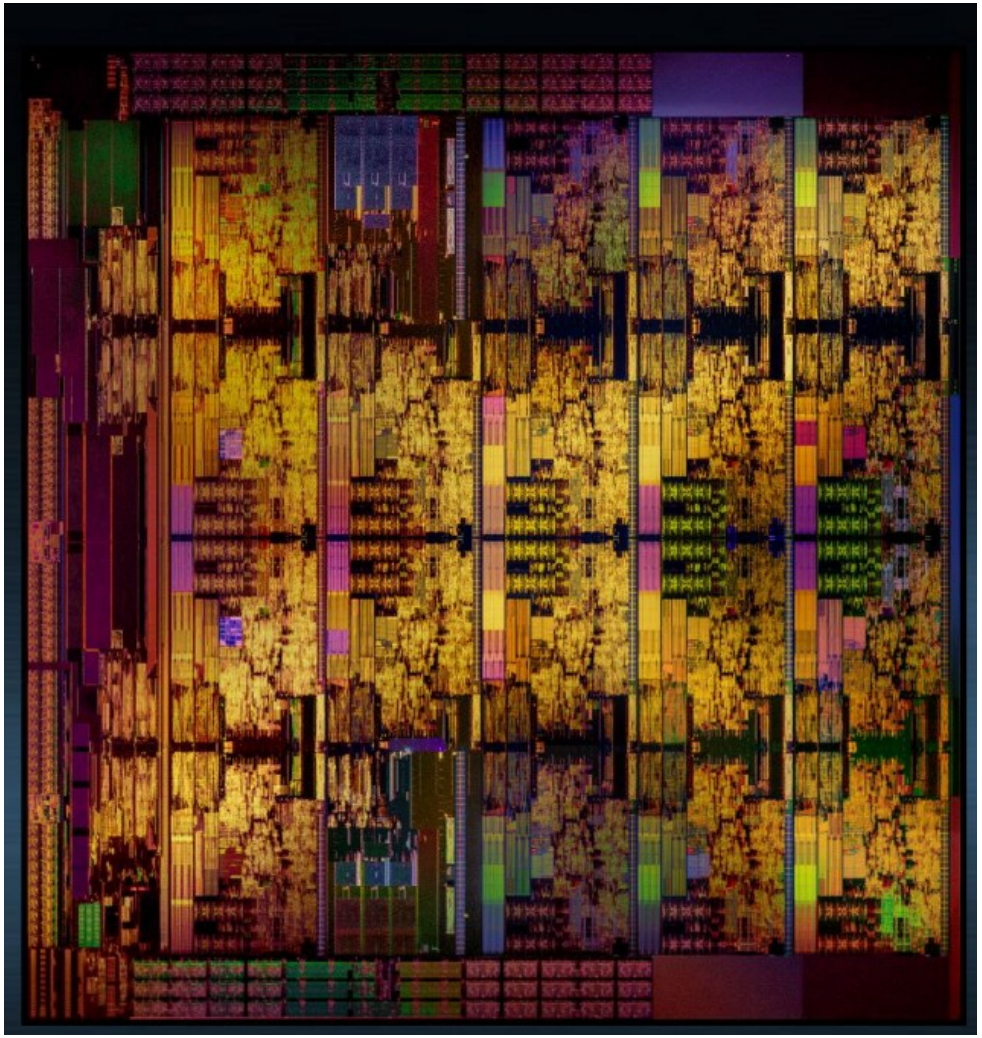
- 1024 64-bit RISC processors
- 64-bit memory architecture
- 64/32-bit IEEE floating point support
- 64MB of distributed on-chip memory
- 1024 programmable I/O signals
- Three 136-bit wide 2D mesh NOCs
- 2052 Independent Power Domains
- Support for up to 1 billion shared memory processors
- Binary compatibility with Epiphany III/IV chips
- Custom ISA extensions for deep learning, communication, and cryptography

Function	Value (mm^2)	Share of Total Die Area
SRAM	62.4	53.3%
Register File	15.1	12.9%
FPU	11.8	10.1%
NOC	12.1	10.3%
IO Logic	6.5	5.6%
"Other" Core Stuff	5.1	4.4%
IO Pads	3.9	3.3%
Always on Logic	0.66	0.6%

Table 5: Epiphany-V Area Breakdown

Intel Skylake-X, Core i9-7980XE (2017)

- 18 core
- 2D mesh topology

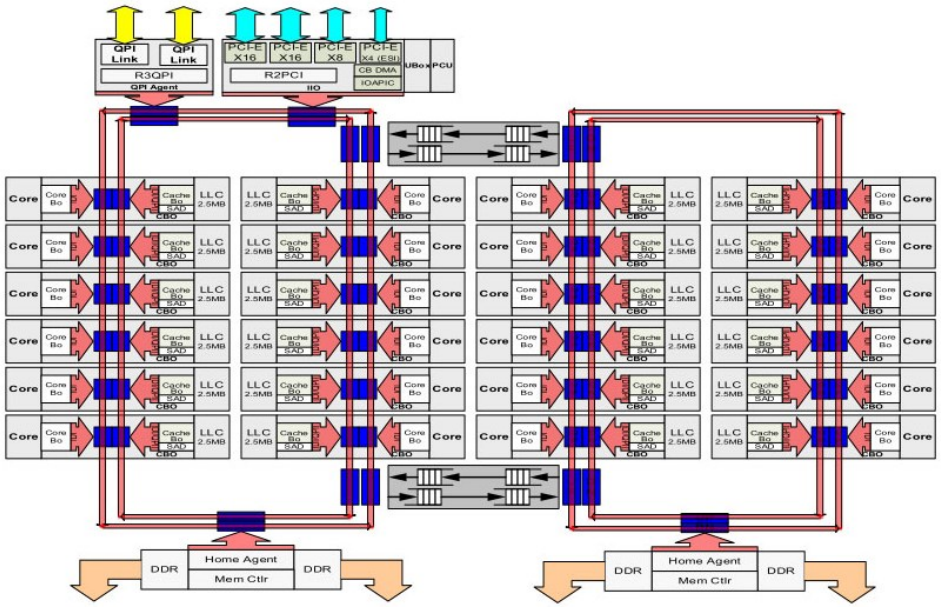


Intel Xeon Scalable Processor

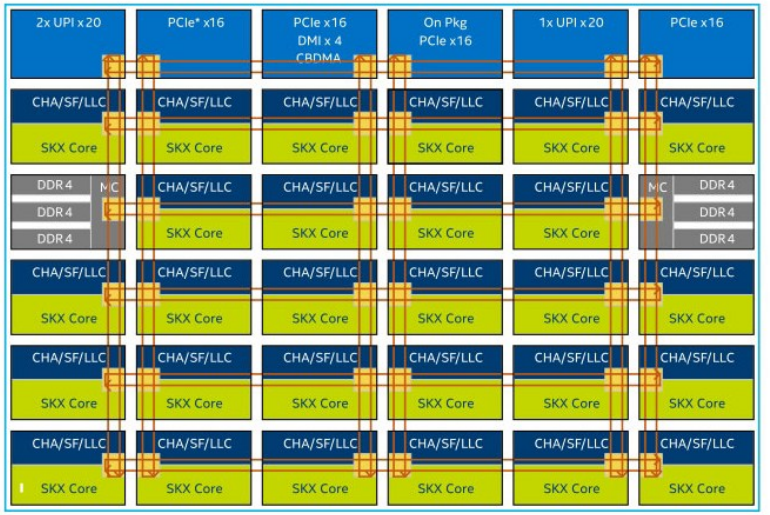
This slide under embargo until 1:00 PM PDT June 15, 2017

New Mesh Interconnect Architecture

Broadwell EX 24-core die



Skylake-SP 28-core die



CHA – Caching and Home Agent ; SF – Snoop Filter; LLC – Last Level Cache;
SKX Core – Skylake Server Core; UPI – Intel® UltraPath Interconnect

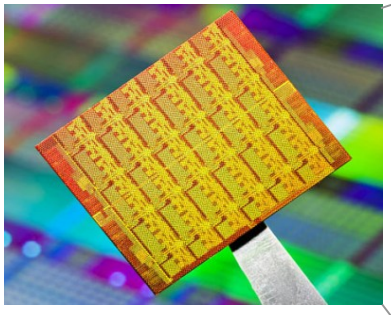
MESH IMPROVES SCALABILITY WITH HIGHER BANDWIDTH AND REDUCED LATENCIES

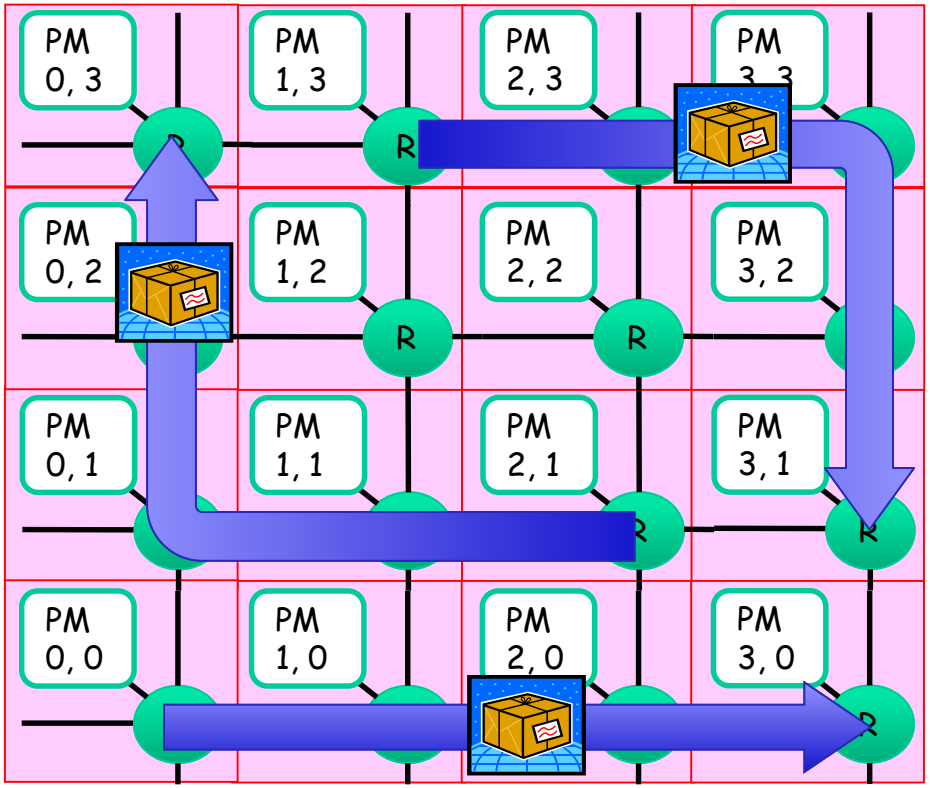
Bus vs. Networks on Chip (NoC) of mesh topology

intersection

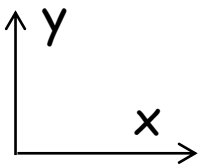
Typical NoC architecture of mesh topology

- NoC requirements: low latency, high throughput, low cost
- Packet based data transmission via NoC routers and XY-dimension order routing





Packet
(tag + data)

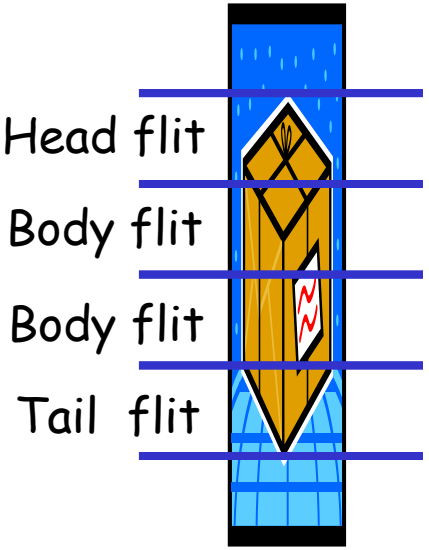


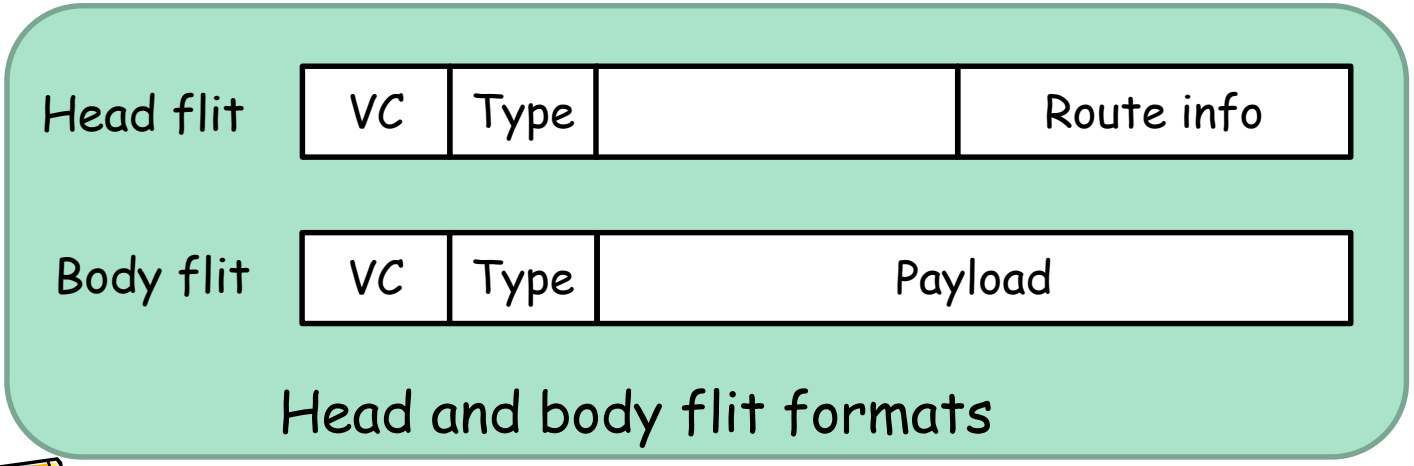
PM: Processing Module or Core,
R: Router

Packet organization (Flit encoding)

- A flit (flow control unit or flow control digit) is a link-level atomic piece that forms a network packet.
  - A packet has one head flit and some body flits.
- Each flit has typical three fields:
  - payload(data) or route information(tag)
  - flit type : head, body, tail, etc.
  - virtual channel identifier

Packet (tag + data)



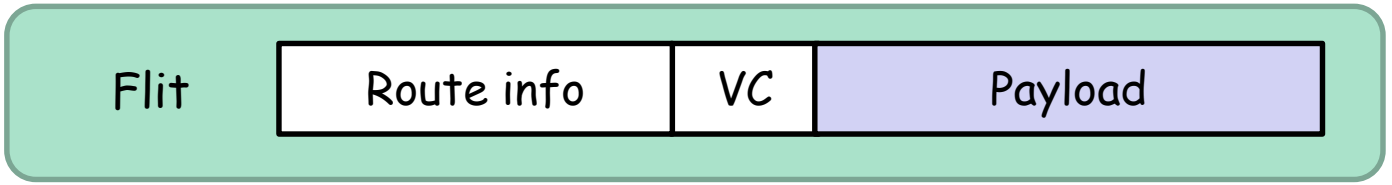


Head and body flit formats

Packet organization (Flit encoding)

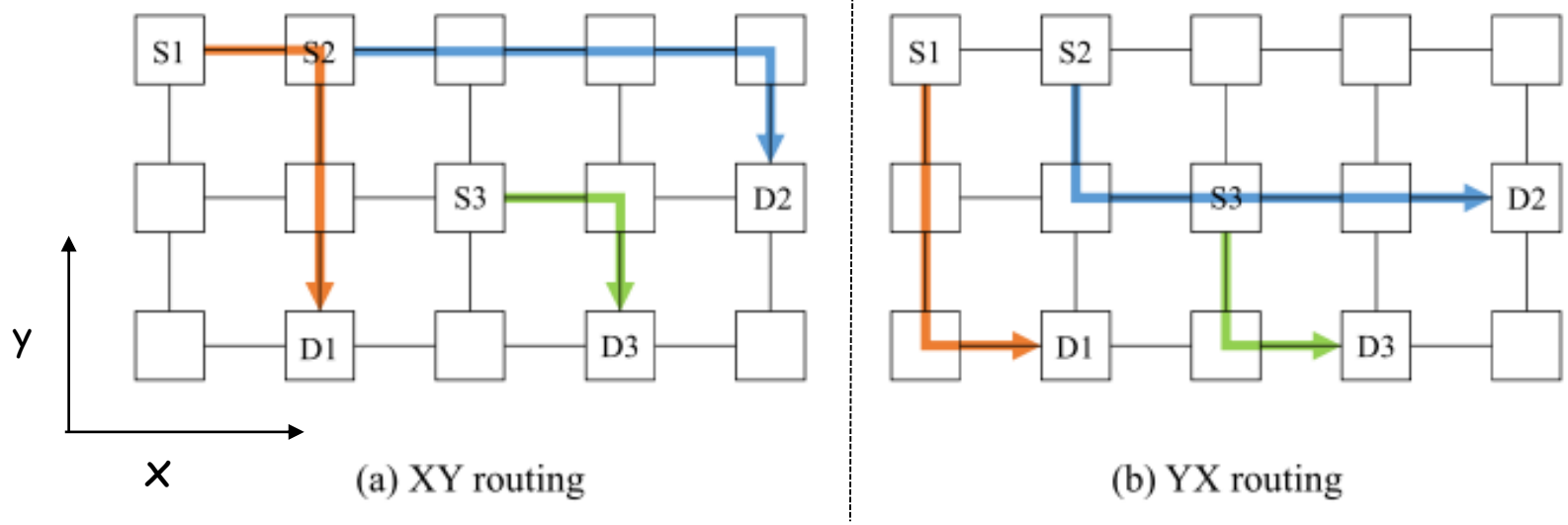
- A flit (flow control unit or flow control digit) is a link-level atomic piece that forms a network packet.
  - A packet has one head flit and some body flits.
- For simplicity, assume that a packet has only one flit.
- Each flit has typical three fields:
  - Payload (data)
  - Route information
  - Virtual channel identifier (VC)

Packet (tag + data)



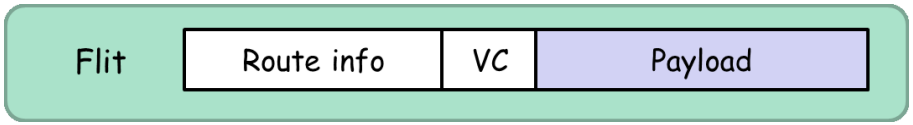
Routing

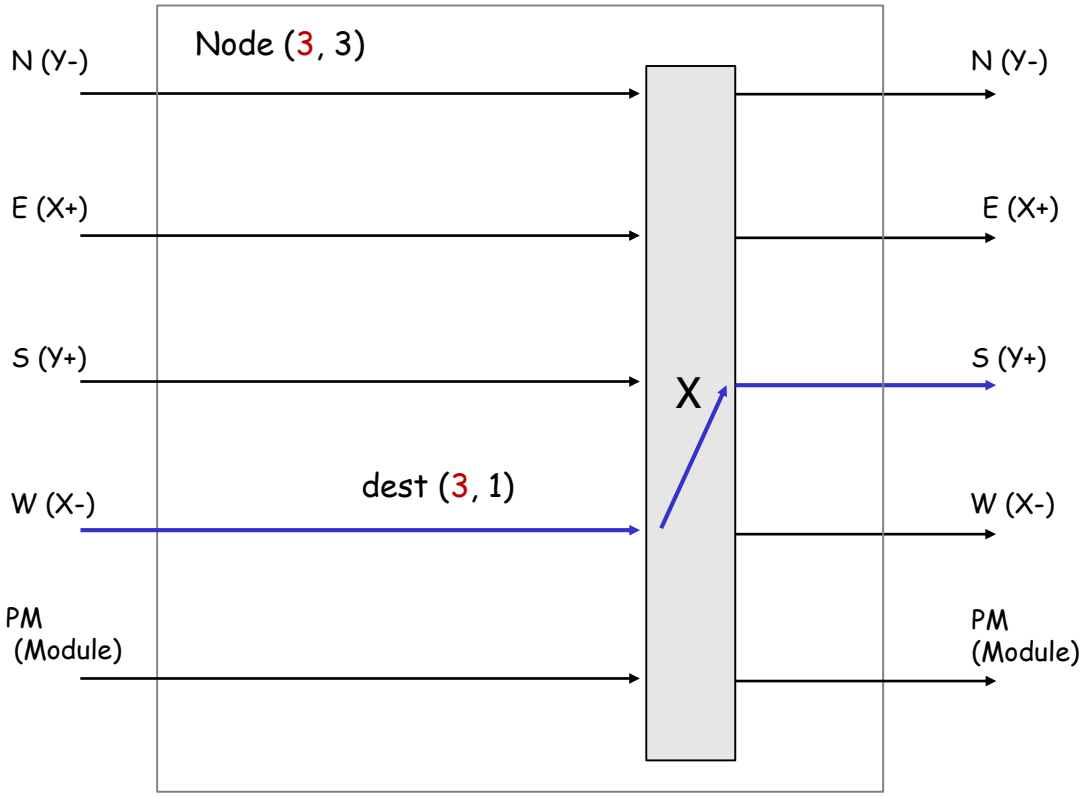
- XY dimension order routing (XY DOR), and YX DOR



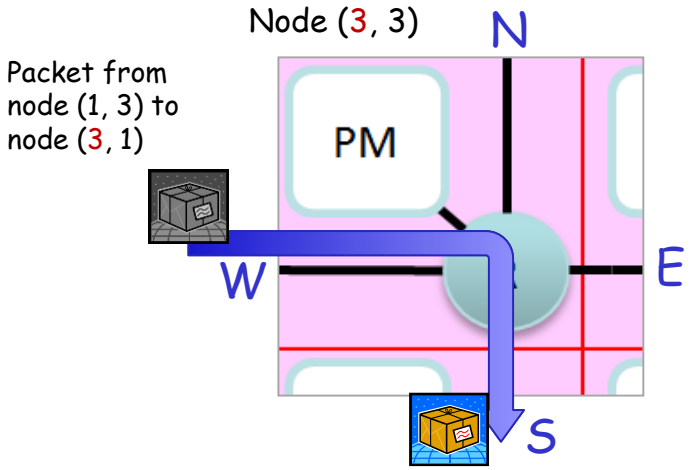
Simple NoC router architecture

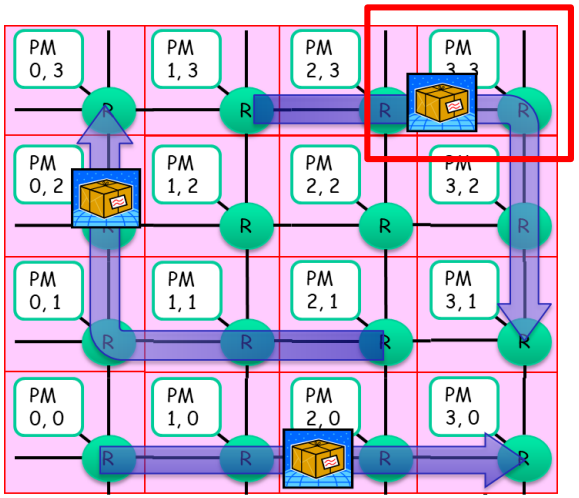
- Routing computation for XY-dimension order





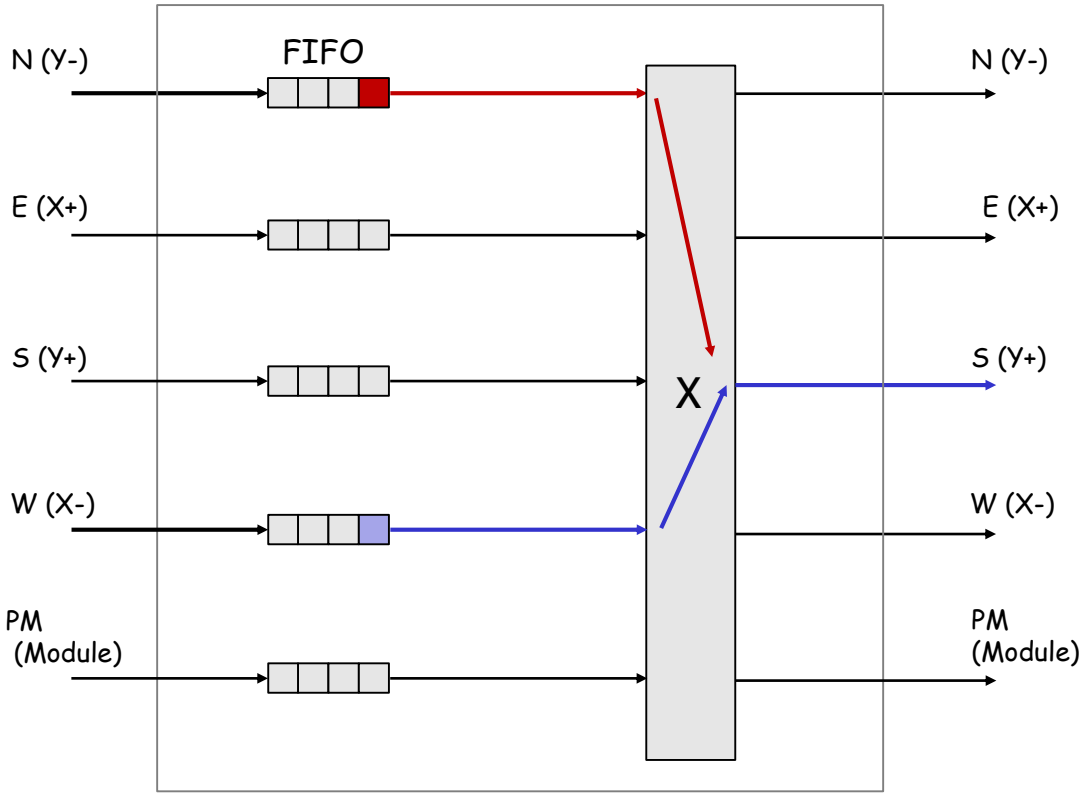
NoC router

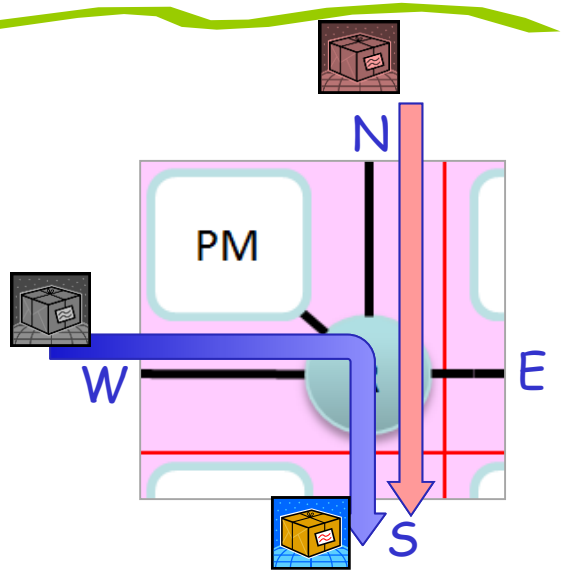




Simple NoC router architecture

- Buffering and arbitration
  - time stamp based, round robin, etc.

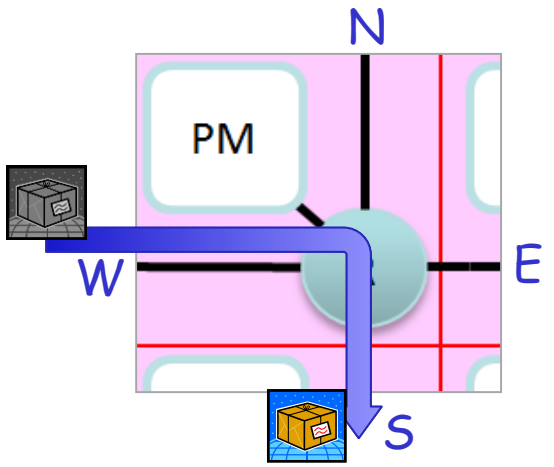




NoC router

Simple NoC router architecture

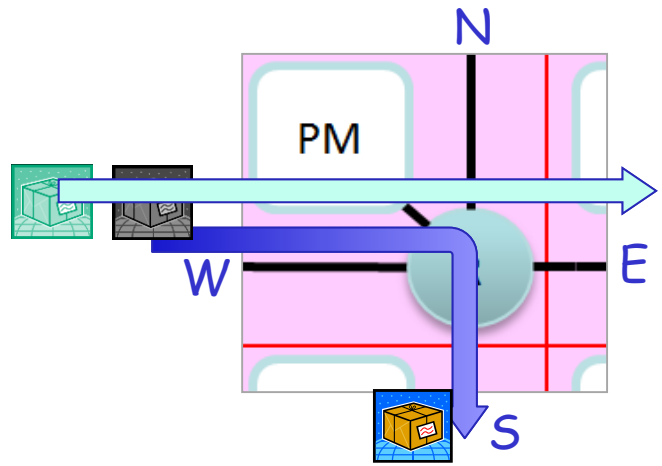
- Flow control (back pressure)
  - When the destination router's input buffer is full, the packet cannot be sent.



NoC router

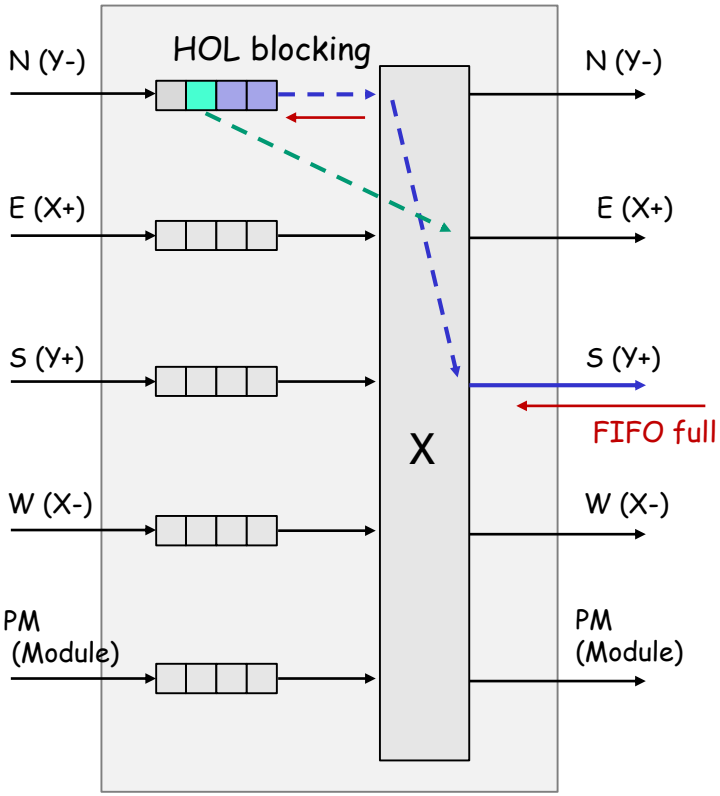
Simple NoC router architecture

- Problem: Head-of-line (HOL) blocking
  - The first (head) packet in the same buffer blocks the movement of subsequent packets.

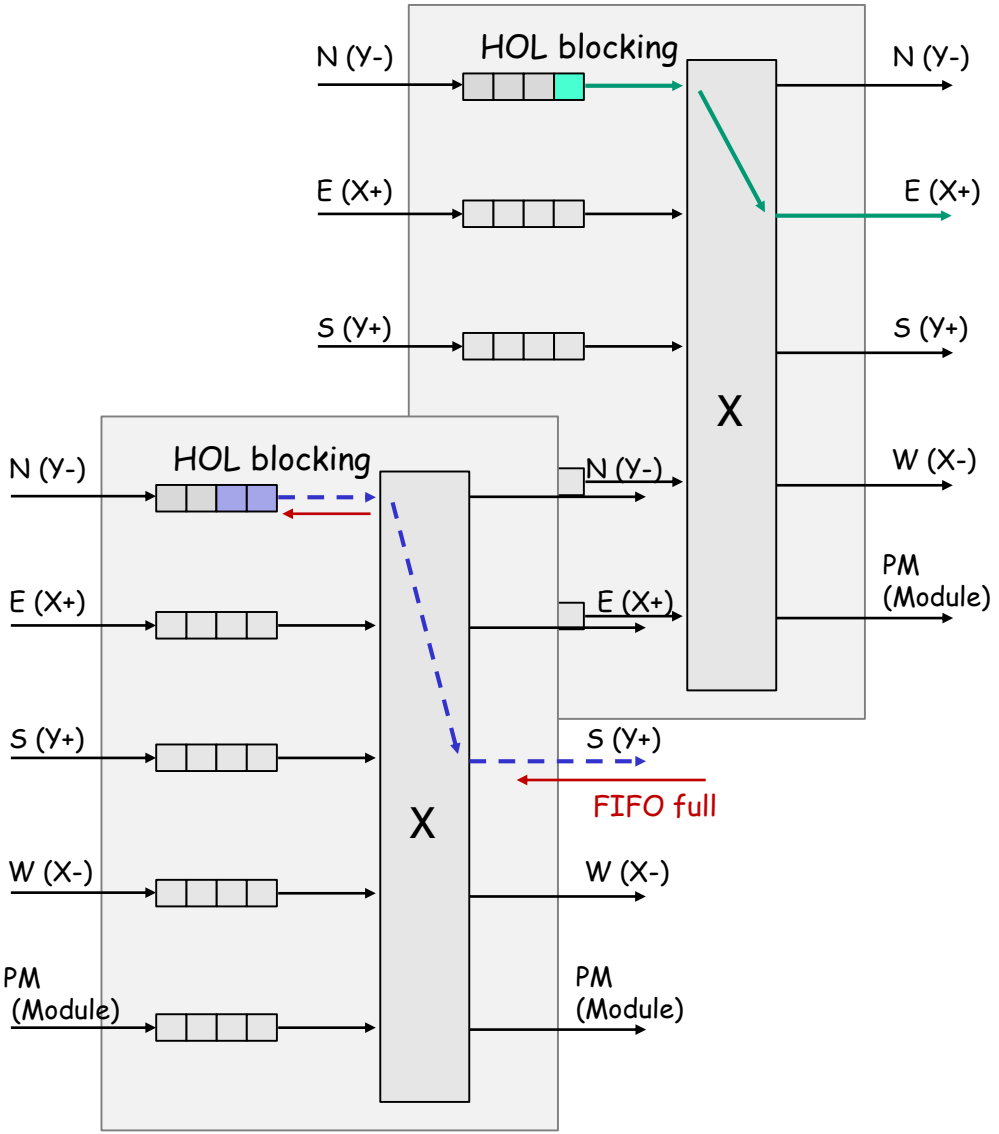


NoC router

Two (physical) networks to mitigate HOL ?

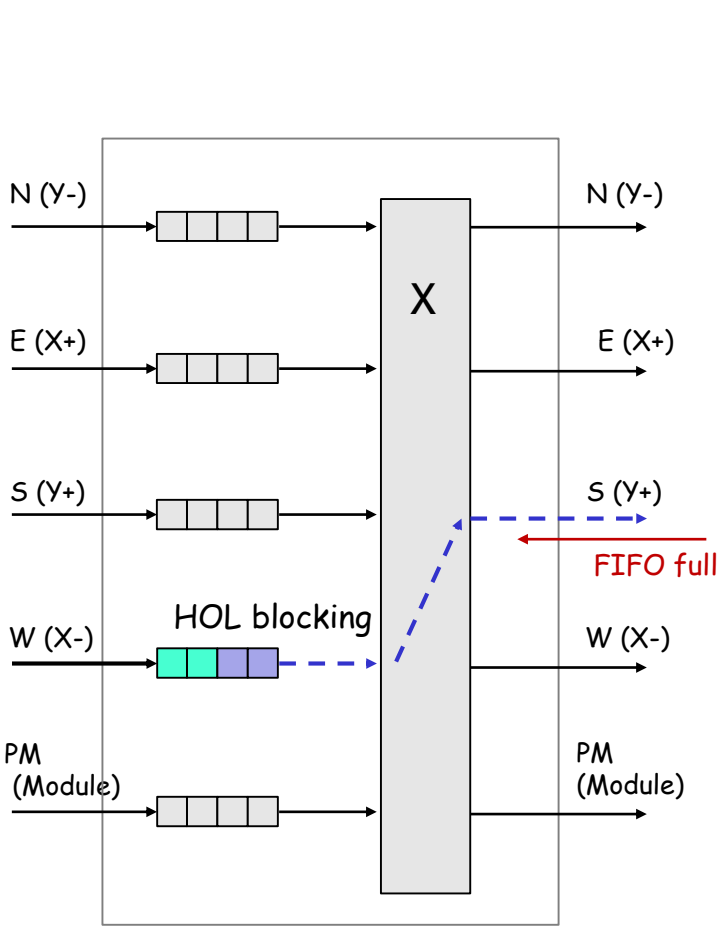


Simple NoC router

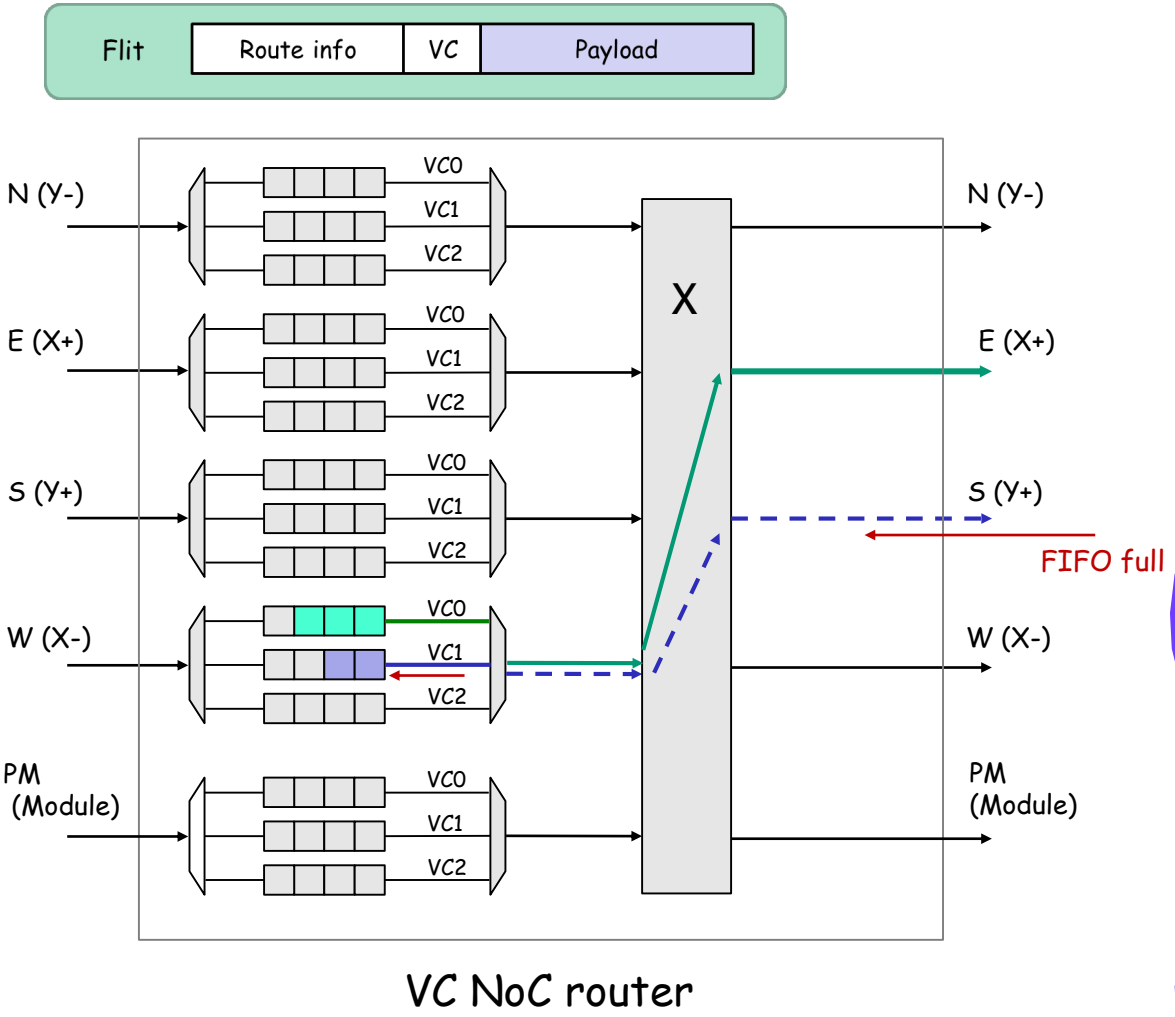


Datapath of Virtual Channel (VC) NoC router

- To mitigate head-of-line (HOL) blocking, virtual channels are used



Simple NoC router



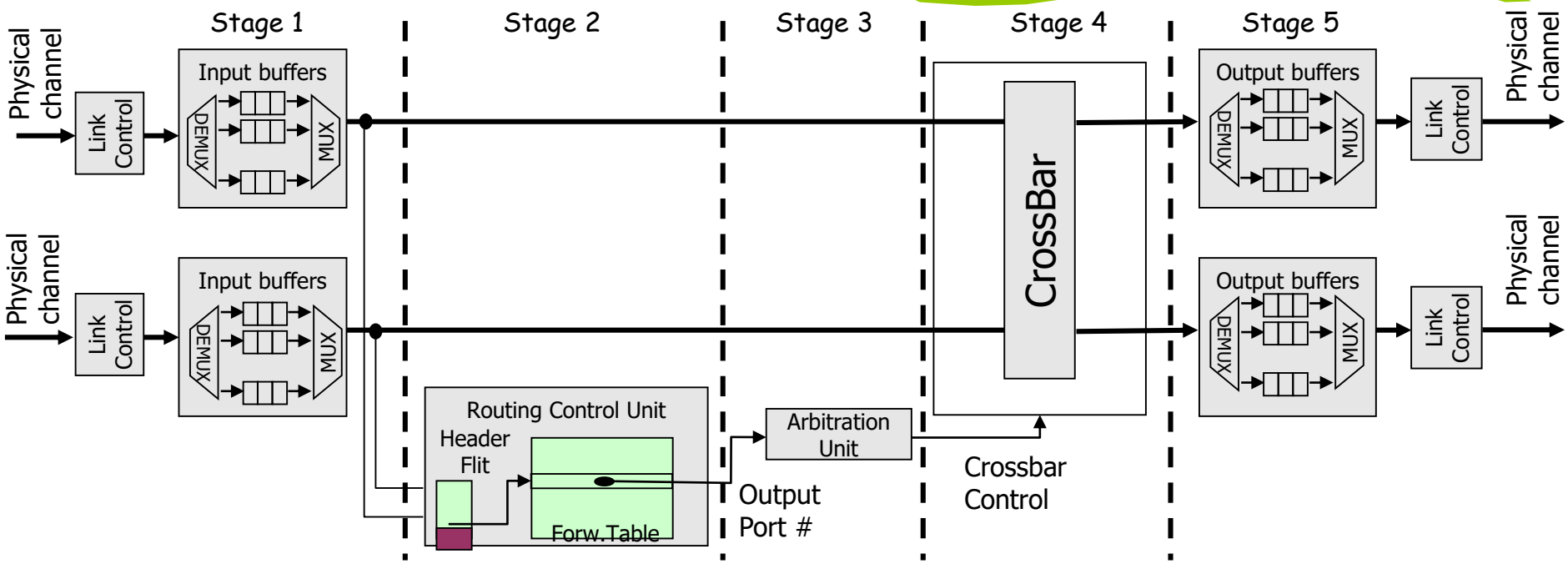
VC NoC router

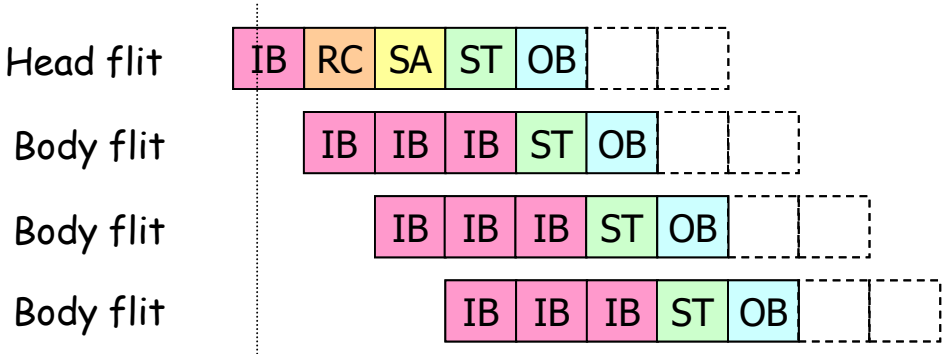
Bus vs. Networks on Chip (NoC) of mesh topology

To mitigate
head-of-line (HOL) blocking

Virtual Channel

Pipelining the NoC router microarchitecture





"A Delay Model and Speculative Architecture for Pipelined Routers," L. S. Peh and W. J. Dally, *Proc. of the 7th Int'l Symposium on High Performance Computer Architecture*, January, 2001.

Bus vs. Networks on Chip (NoC) of mesh topology

Packet
(tag + data)

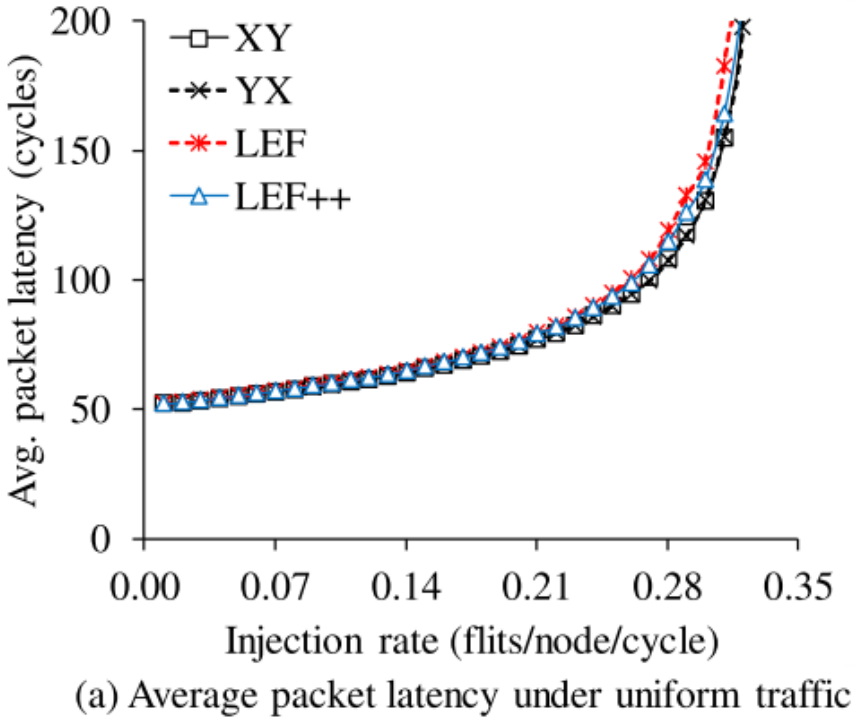
Distributed system

FIFO

intersection

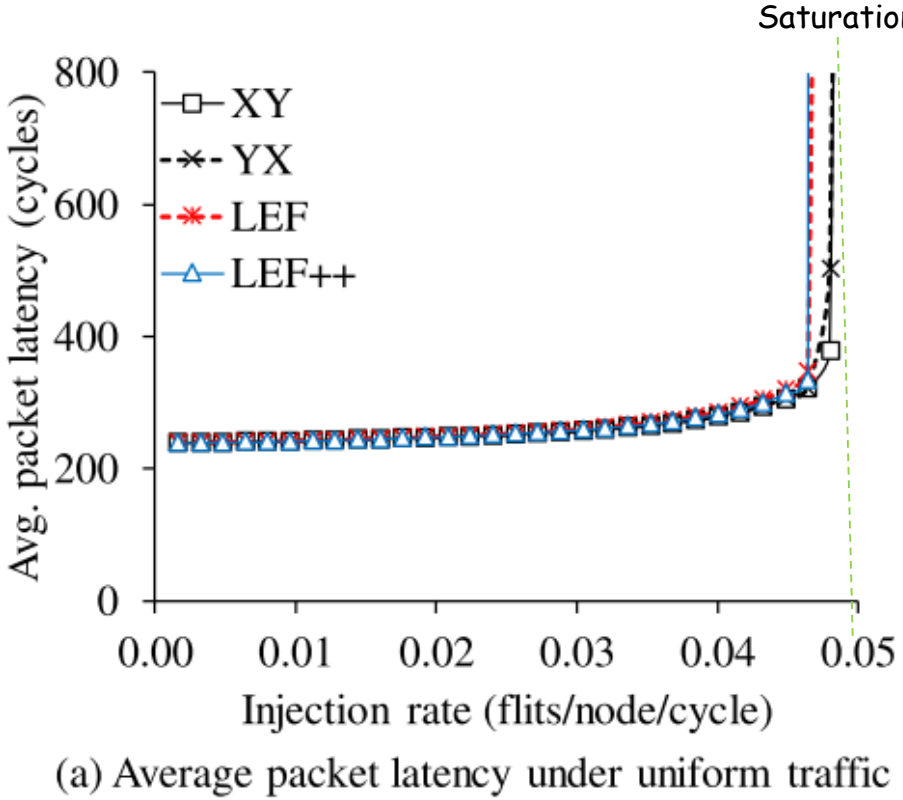
Average packet latency of mesh NoCs

- 5 stage router pipeline
- Uniform traffic (destination nodes are selected randomly)



(a) Average packet latency under uniform traffic

8x8 NoC



(a) Average packet latency under uniform traffic

64x64 NoC (4096 nodes)

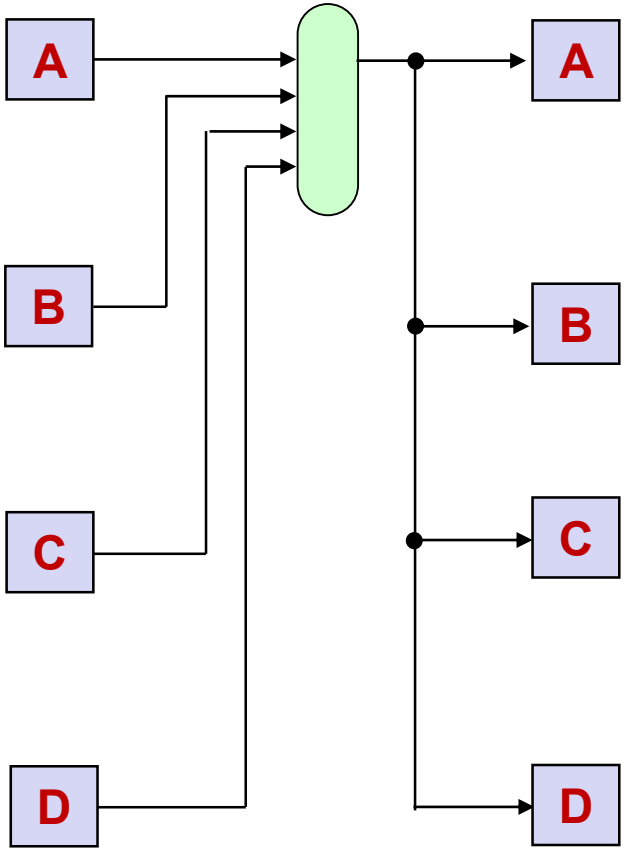
Thiem Van Chu, Myeonggu Kang, Shi FA and Kenji Kise: Enhanced Long Edge First Routing Algorithm and Evaluation in Large-Scale Networks-on-Chip, IEEE 11th International Symposium on Embedded Multicore/Many-core Systems-on-Chip, (September 2017).

On-Chip Interconnection network requirements

- Connecting many modules on a chip achieving high throughput and low latency
  - Topology
    - the number of ports, links, switches (HW resources)
    - bus, ring bus, tree, fat-tree, crossbar, mesh, torus
  - Circuit switching / packet switching
    - Centralized control / distributed control with FIFO and flow control (scalability)
  - Routing
    - deadlock free, livelock free
    - in-order data delivery / out-of-order delivery
    - adaptive routing, XY-dimension order routing
  - Network-on-chip (NoC) router architecture

Bus Network with multiplexer (mux)

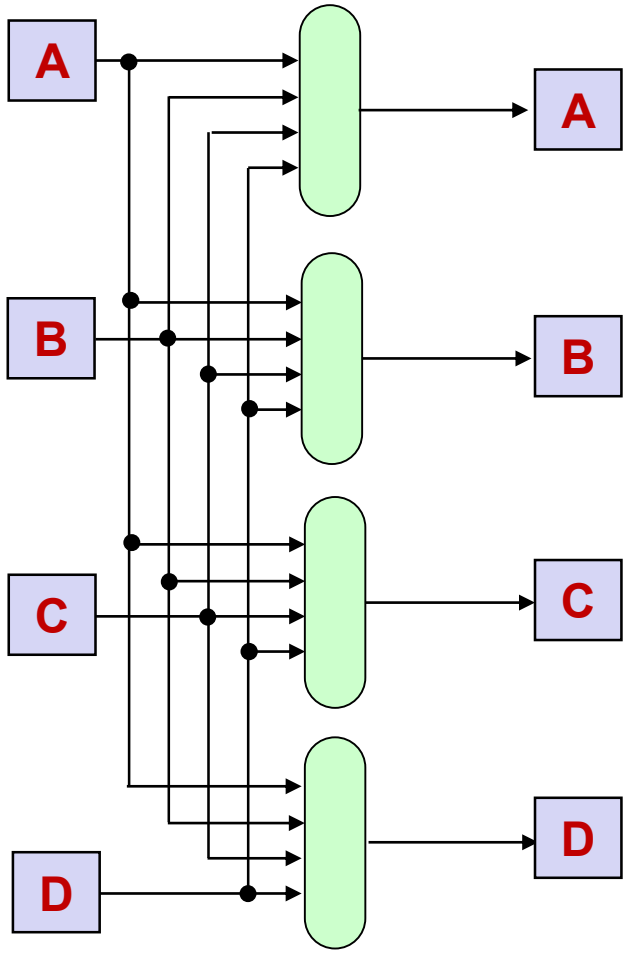
- One N-input multiplexer for N cores
- Arbitration, node ID, centralized control

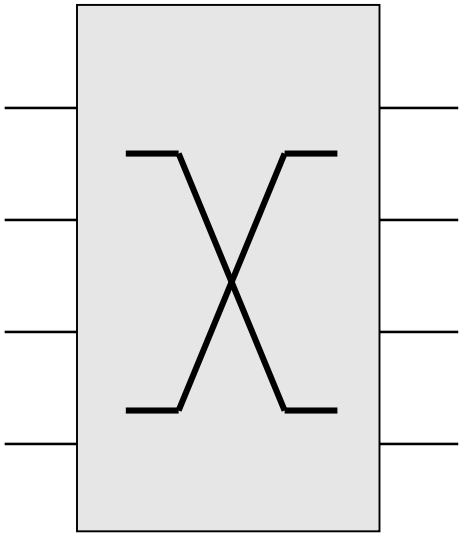


The bus network organization of 4 cores using a 4-input mux.

Crossbar (Xbar) Network with mux

- N N-input multiplexers

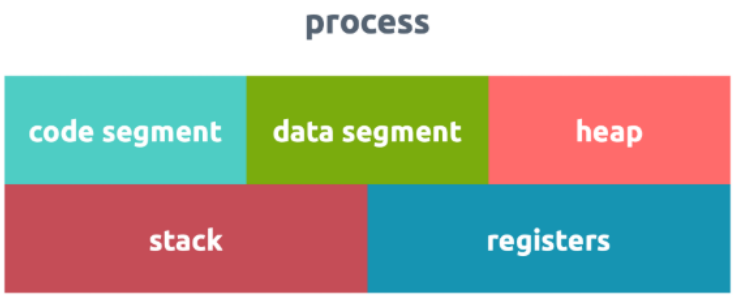




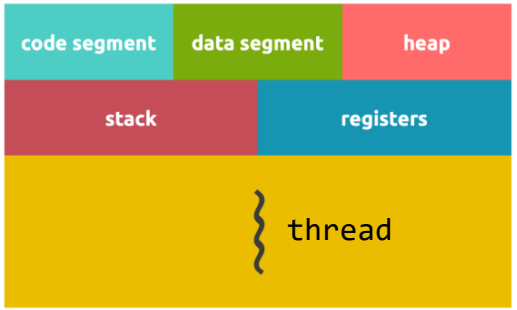
A symbol of Xbar

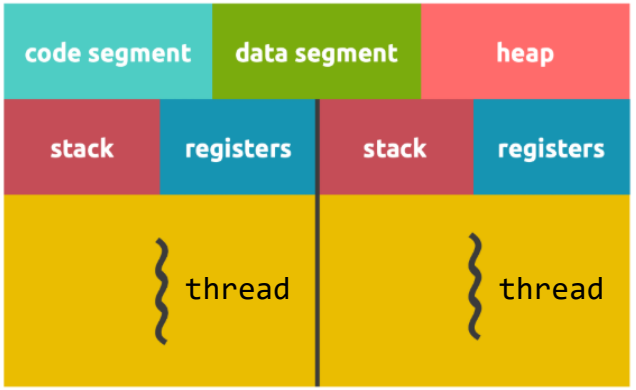
Program, process, and thread

- A process is an instance of a program that is being executed whereas a thread is part of a process.
- A process can have more than one thread. All the threads within one process are interrelated to each other. Threads have some common information, such as code segment, data segment, heap, etc., that is shared to their threads. But contains its own stack and registers (PC and x0 – x32 registers).



Multiprogramming





Parallel programming

https://zenn.dev/farstep/articles/process-thread-difference

Quiz

- Are these three barriers necessary in the parallel program?
- What happens if we remove Barrier 1?
- What happens if we remove Barrier 2?
- What happens if we remove Barrier 3?

```
float A[N+2], B[N+2]; /* these are in shared memory */
float diff=0.0;       /* variable  in shared memory */

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_barrier_t barrier;
void solve_pp (int pid) {
    int i, done = 0;                    /* private variables */
    int mymin = (pid==0) ? 1 : 5;       /* private variable  */
    int mymax = (pid==0) ? 4 : 8;       /* private variable  */
    while (!done) {
        float mydiff = 0;
        for (i=mymin; i<=mymax; i++) {
            B[i] = 0.333 * (A[i-1] + A[i] + A[i+1]);
            mydiff = mydiff + fabsf(B[i] - A[i]);
        }
        pthread_mutex_lock(&m);
        diff = diff + mydiff;
        pthread_mutex_unlock(&m);

        pthread_barrier_wait(&barrier);  // Barrier 1
        if (diff <TOL) done = 1;
        pthread_barrier_wait(&barrier);  // Barrier 2
        if (pid==1) diff = 0.0;
        for (i=mymin; i<=mymax; i++) A[i] = B[i];
        pthread_barrier_wait(&barrier);  // Barrier 3
    }
}
```