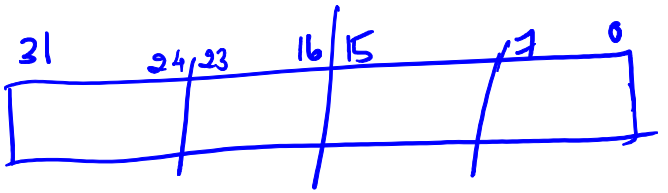
M4 – Parallelism

Outline

- Parallelism
- Flynn's classification
- Vector Processing
  - Subword Parallelism
- Symmetric Multiprocessors, Distributed Memory Machines
  - Shared Memory Multiprocessing, Message Passing
- Synchronization
  - ISA
- Cache coherence



Flynn's Classification

- **Single instruction stream, single data stream (SISD)**
  - Uniprocessor.

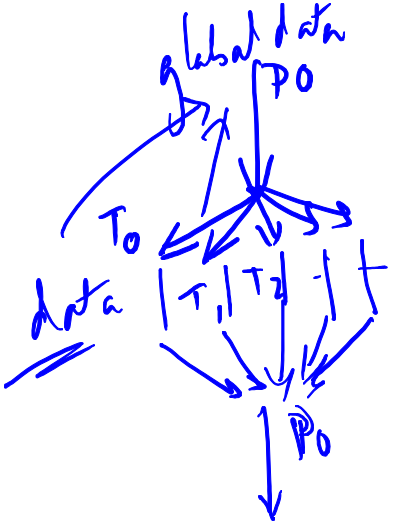
Flynn's Classification

- **Single instruction stream, single data stream (SISD)**
  - Uniprocessor.
- **Single instruction stream, multiple data streams (SIMD)**
  - Data-level parallelism, Subword Parallelism
  - Applying same operations to multiple items of data in parallel
  - Eg. Multimedia extensions, Vector architectures
  - Gaming, 3D, real-time VR, rendering, ...

Flynn's Classification

- **Multiple instruction streams, single data stream (MISD)**
- **Multiple instruction streams, multiple data streams (MIMD)**
  - Thread-level parallelism

Task - level

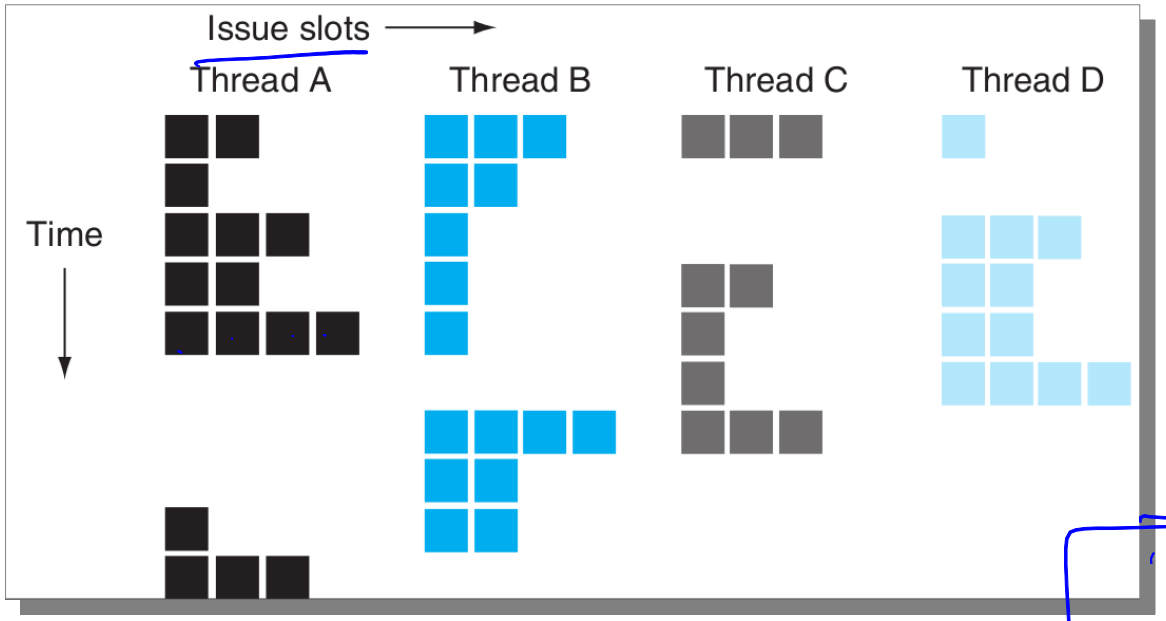


- light weight process
- has its own data (stack)
- shares the code

MIMD

Multithreading

spawn



data protection

• data inconsistency

x = 10

private caches

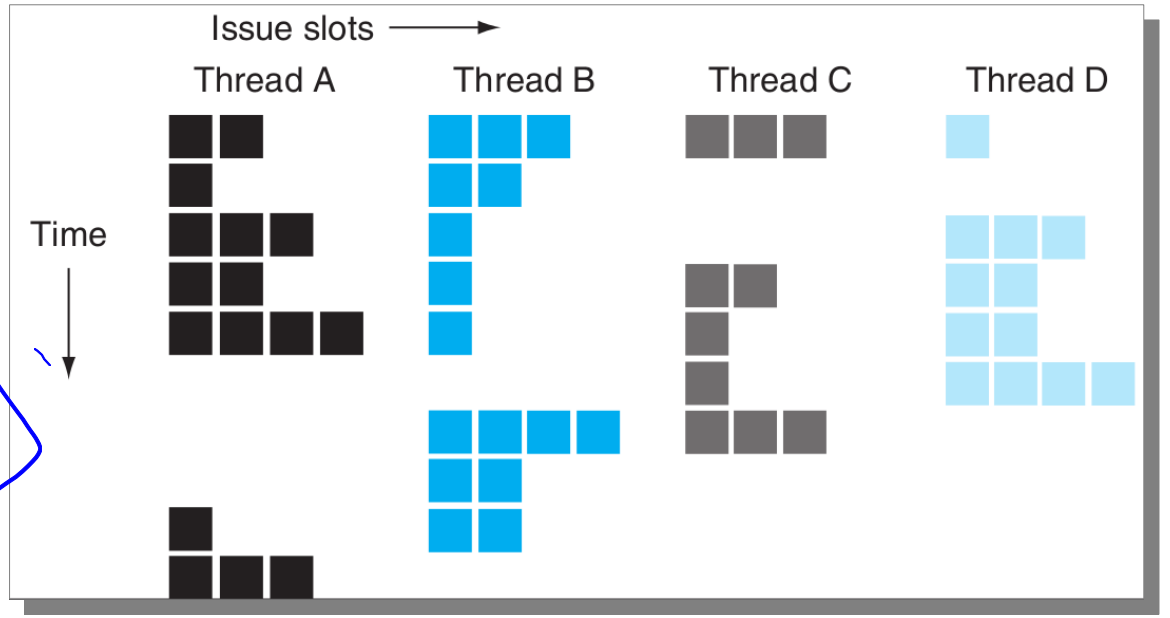
10 T_0 read x ← 10

20 T_1 read x ← 10

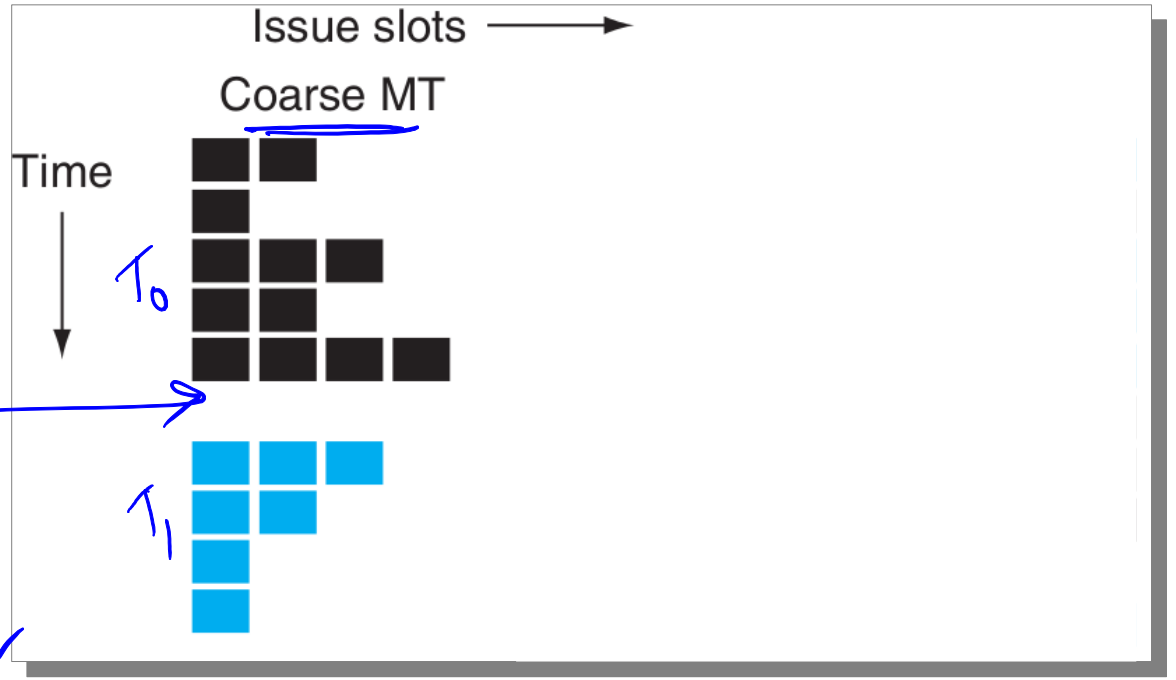
30 T_0 modify x = 20

40 T_1 read x ← 20

Multithreading



PC, RF, special regs, PTBR



- save state $\leftarrow T_0$
- restore state $\leftarrow T_1$

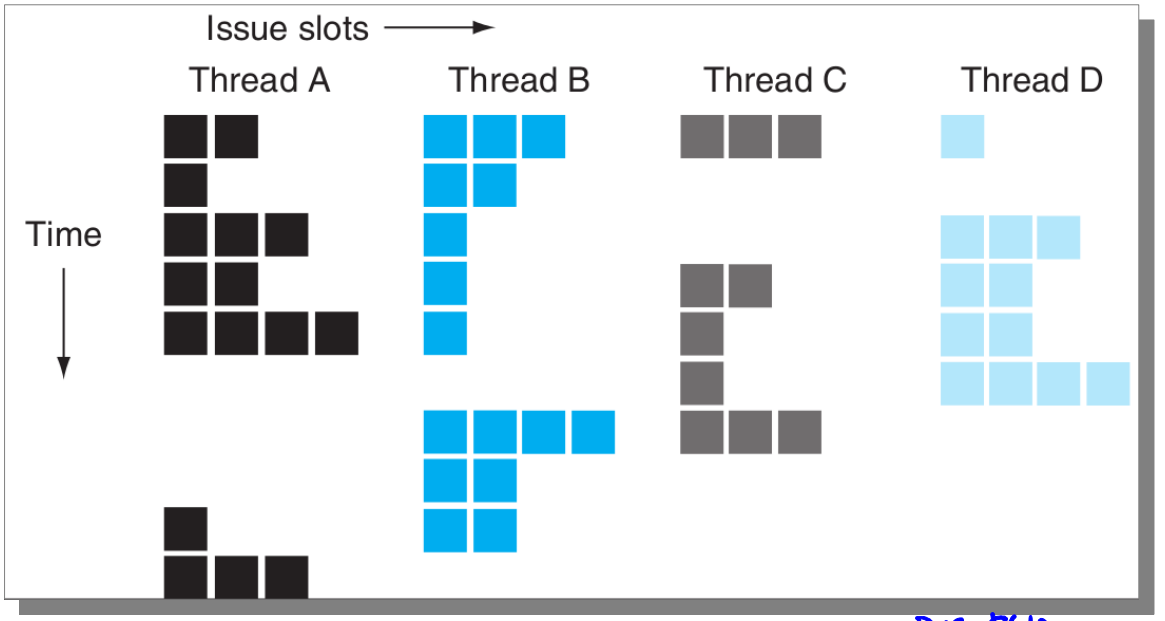
$T_0 \rightarrow$ L00

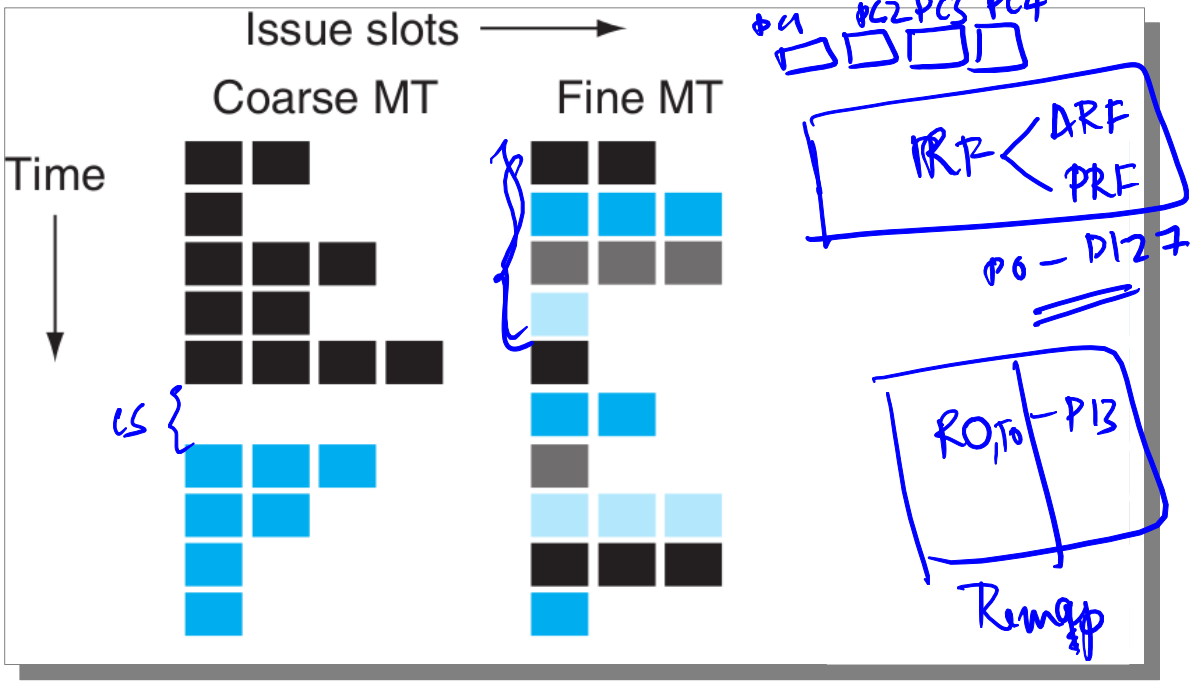
Content switch

$T_1 \rightarrow$ L00

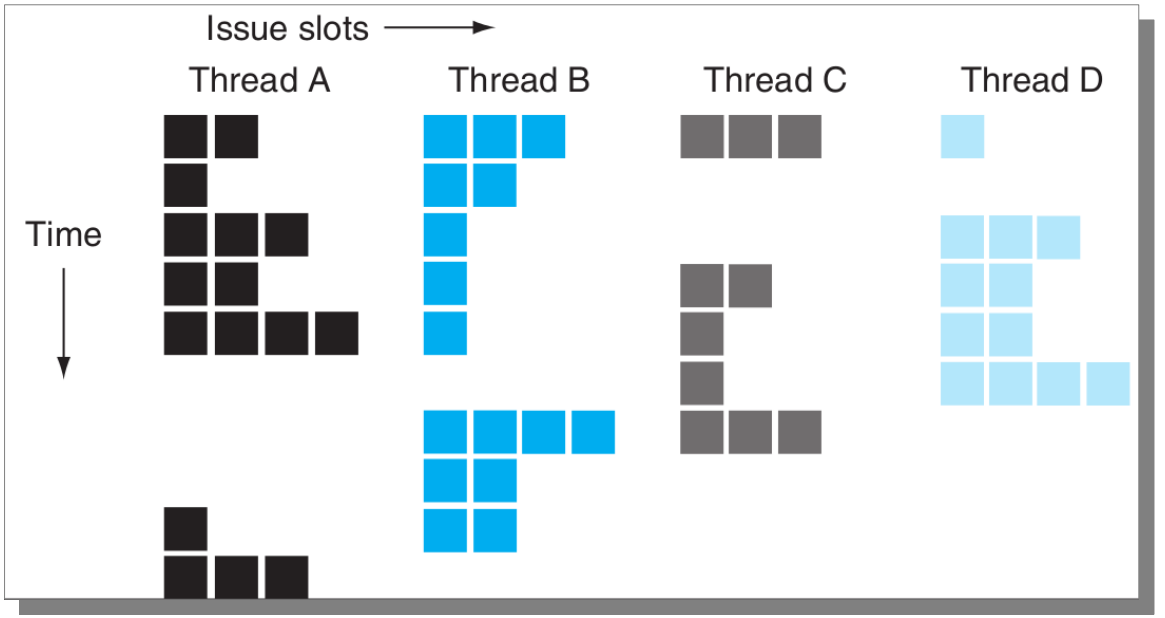
- update dirty blocks L2
- equal to dummy contents
-

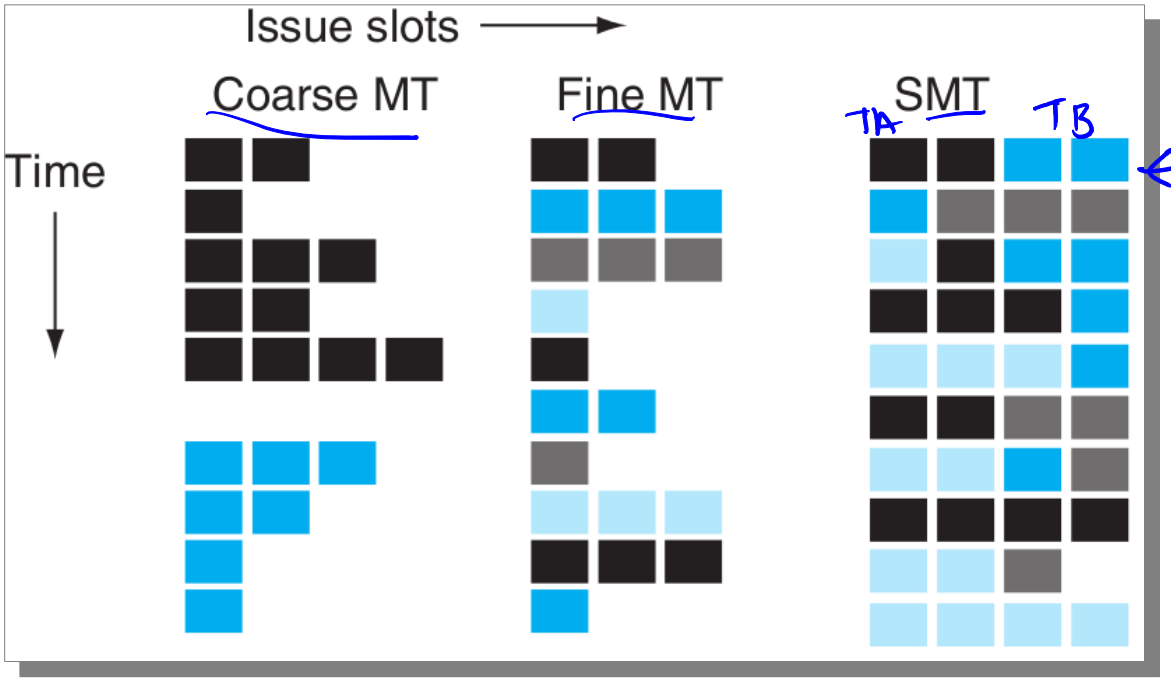
Multithreading





Multithreading



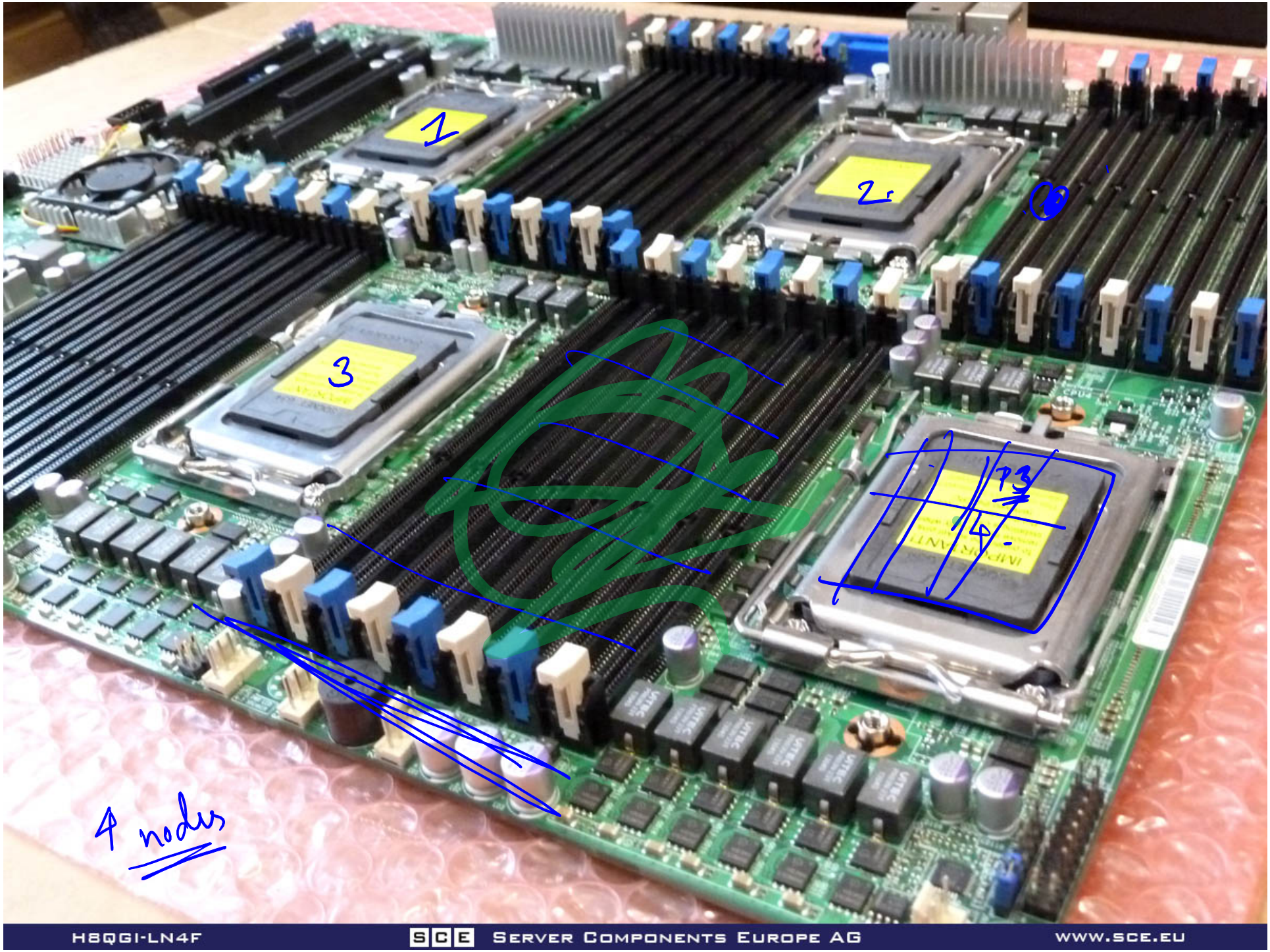


8 core, 2 threads with Hyperthreading

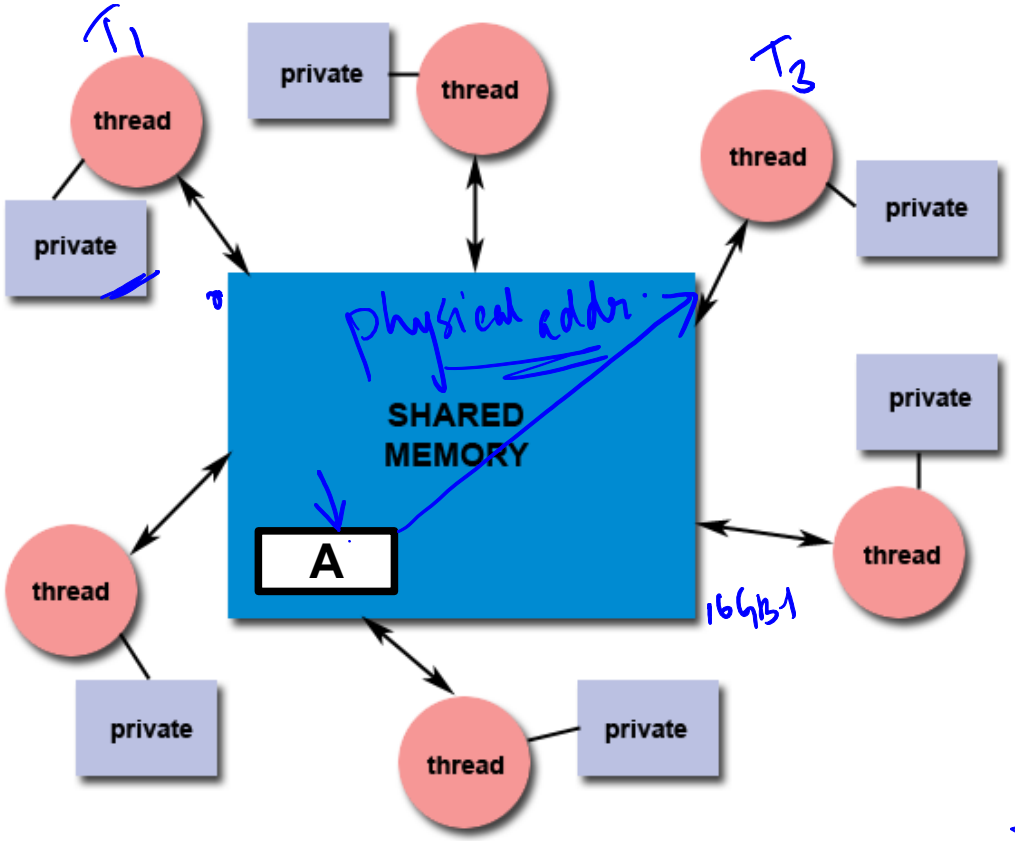
← cycle

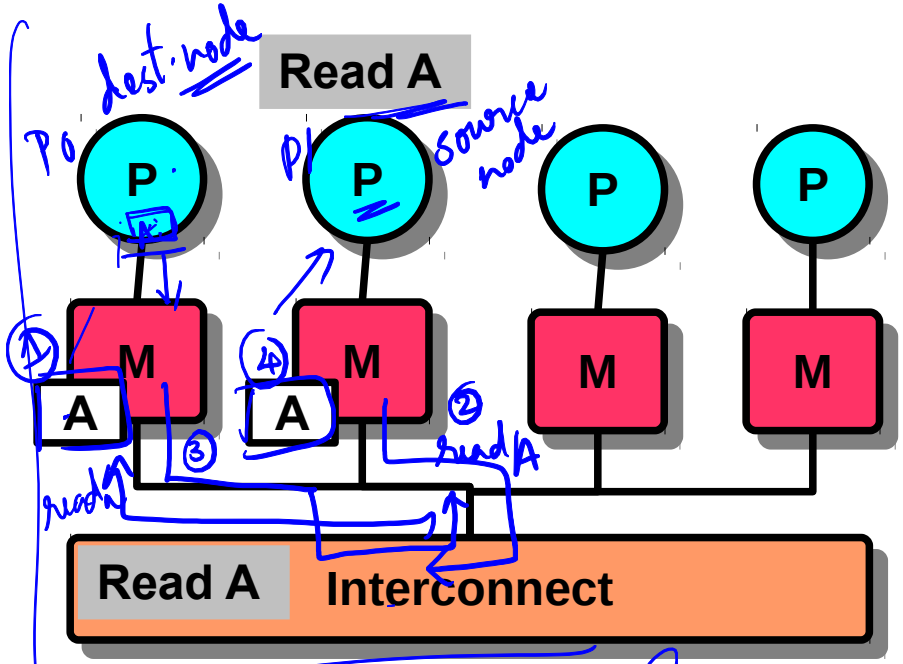
Shared vs. Distributed Memory

- Shared – Processors share physical memory space → 1 MB, 1 socket.
- Distributed – Each node has its own physical address space

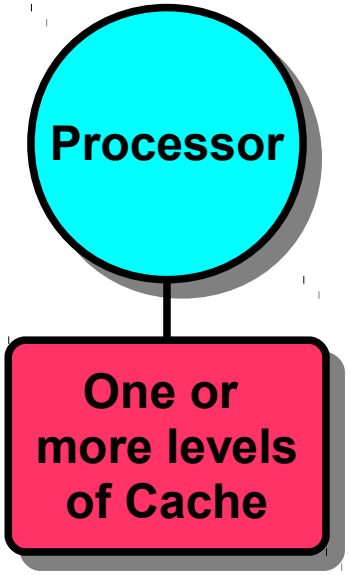

1
2
3
4 nodes

Shared Memory vs. Message Passing

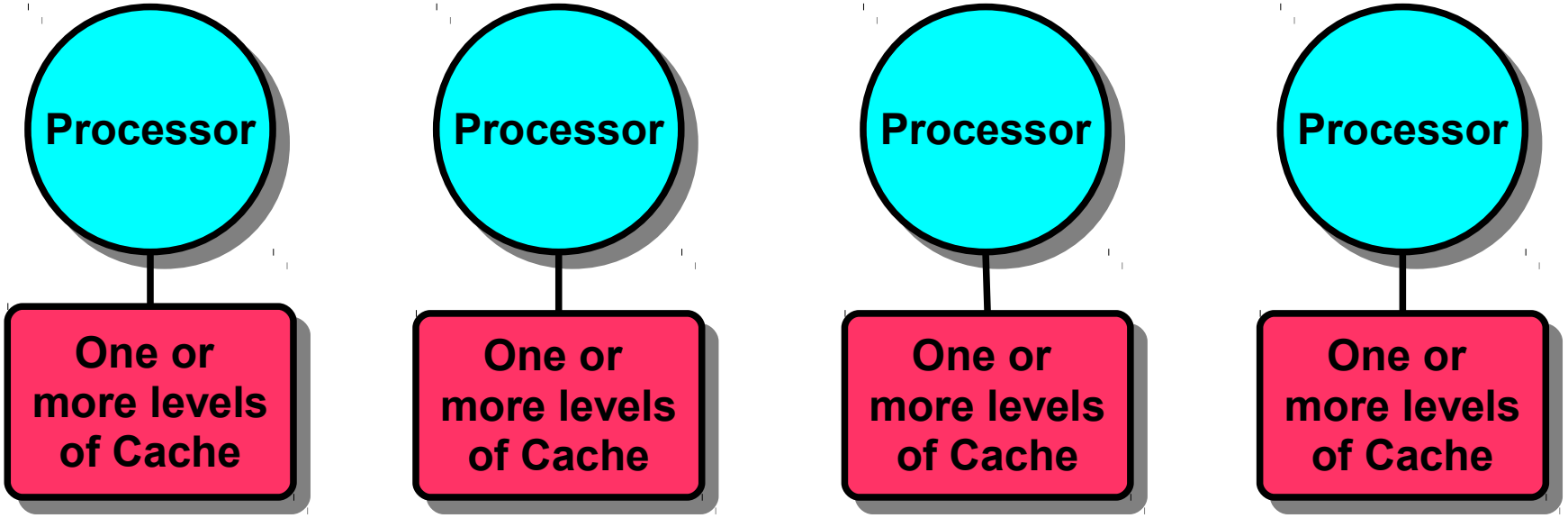




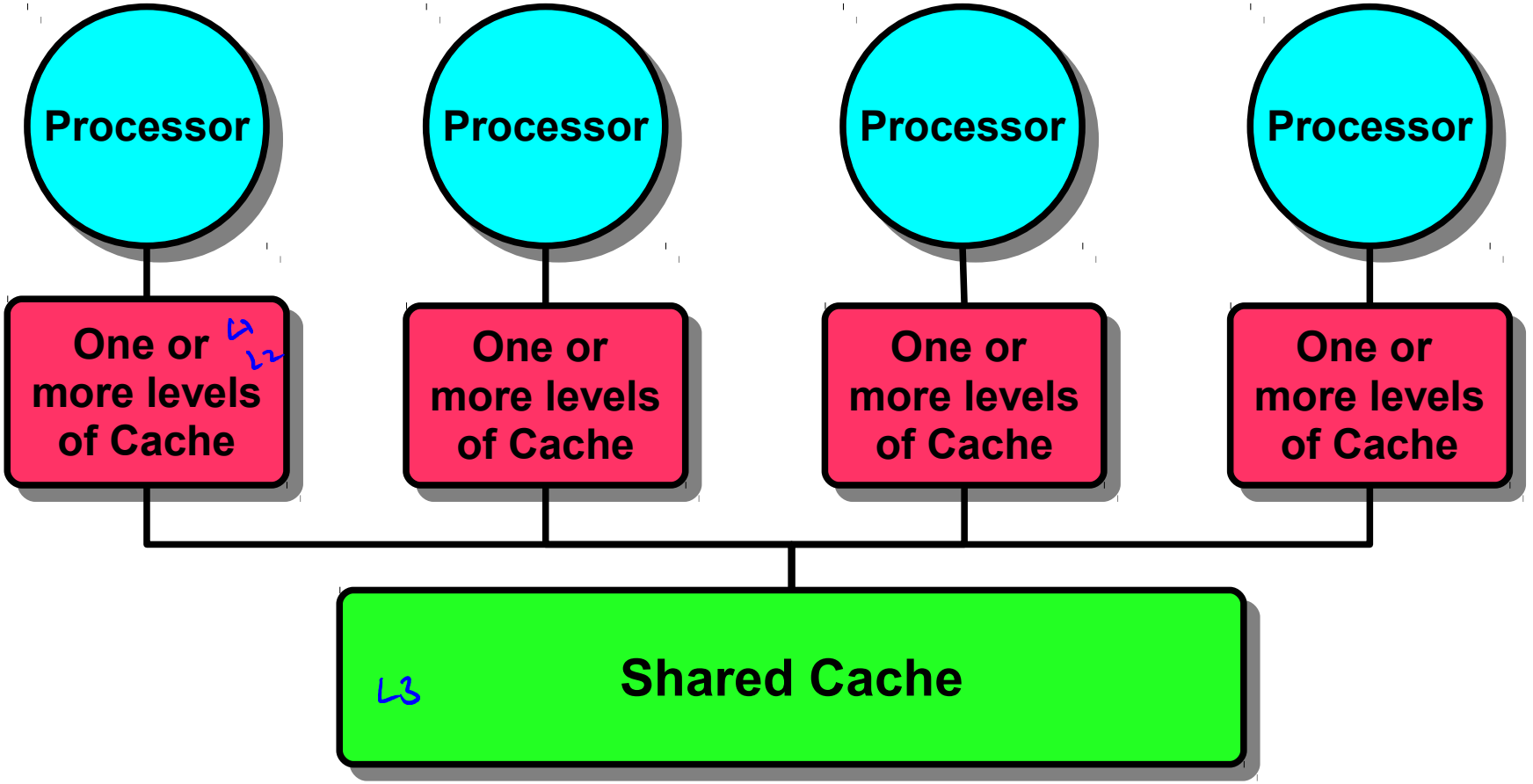
Symmetric Multiprocessor (SMP)



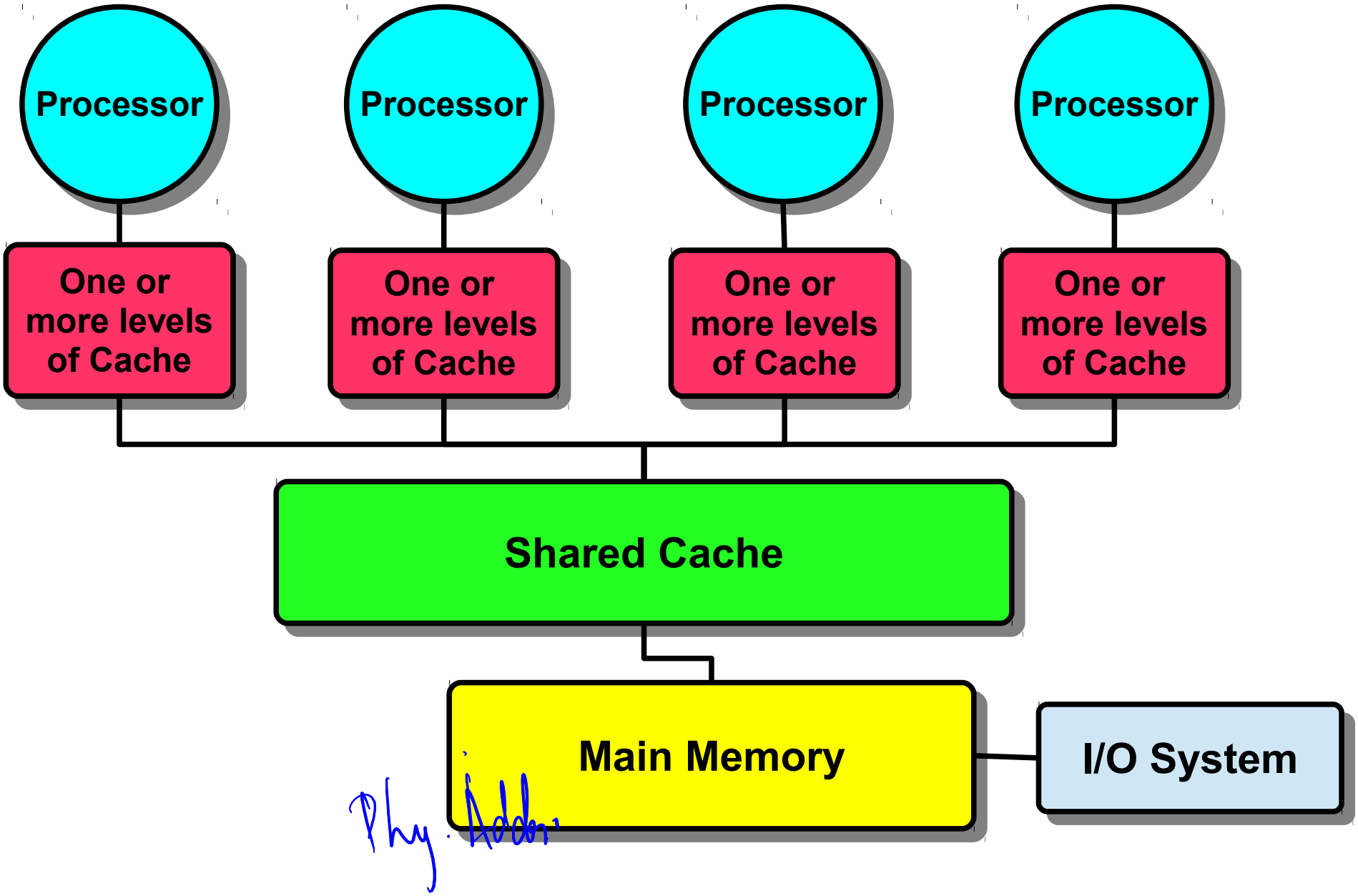
Symmetric Multiprocessor (SMP)



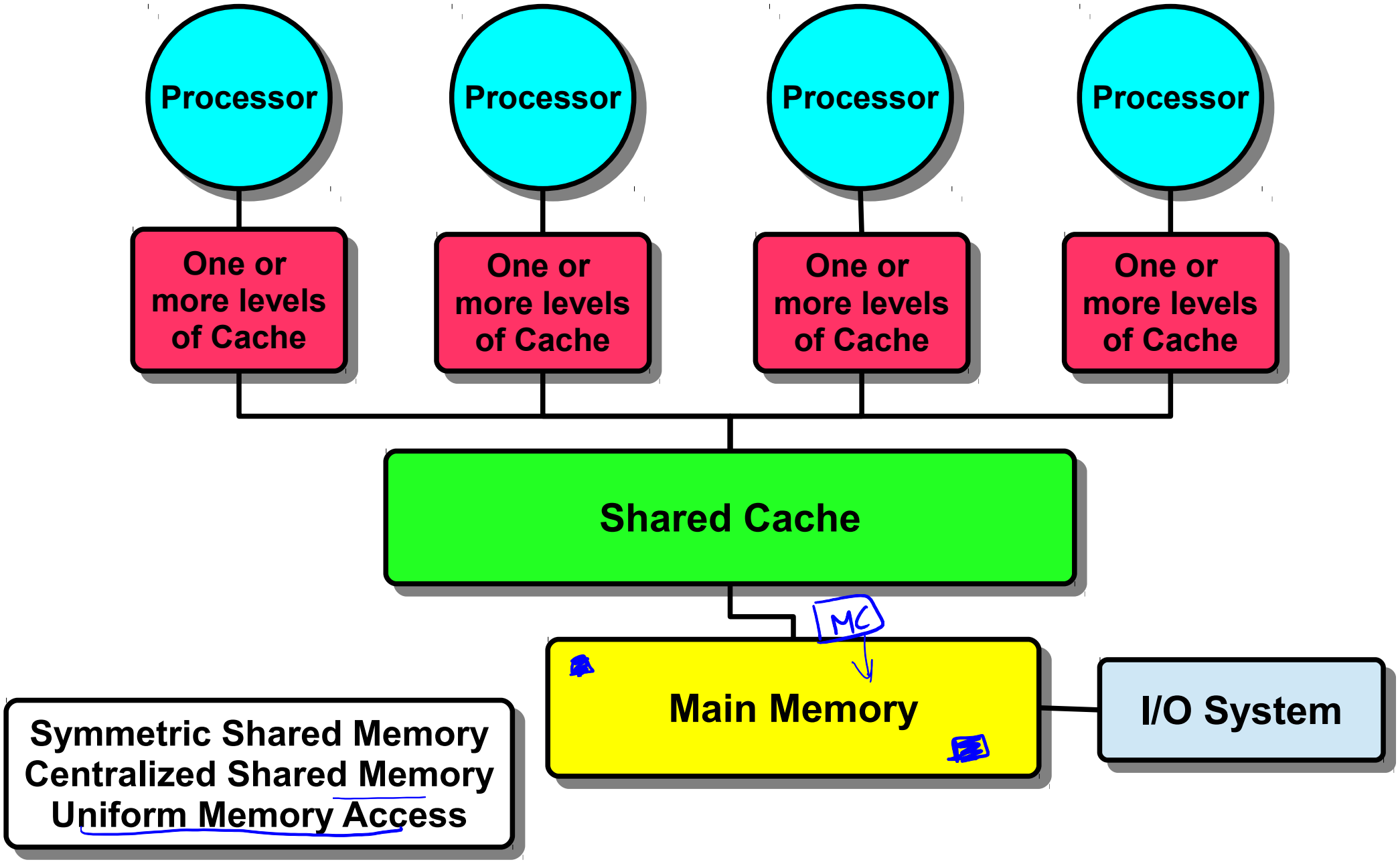
Symmetric Multiprocessor (SMP)



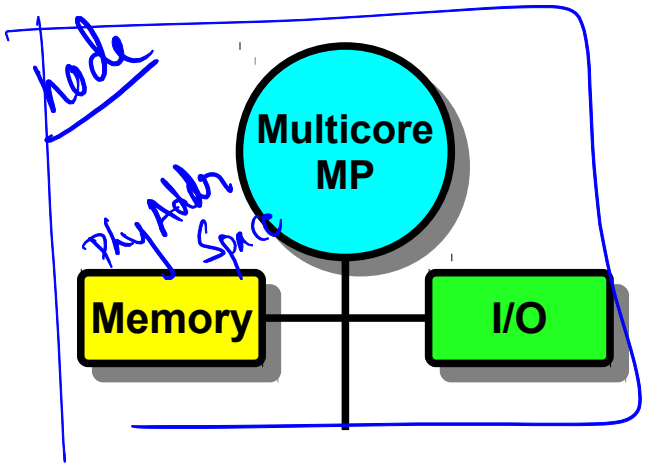
Symmetric Multiprocessor (SMP)



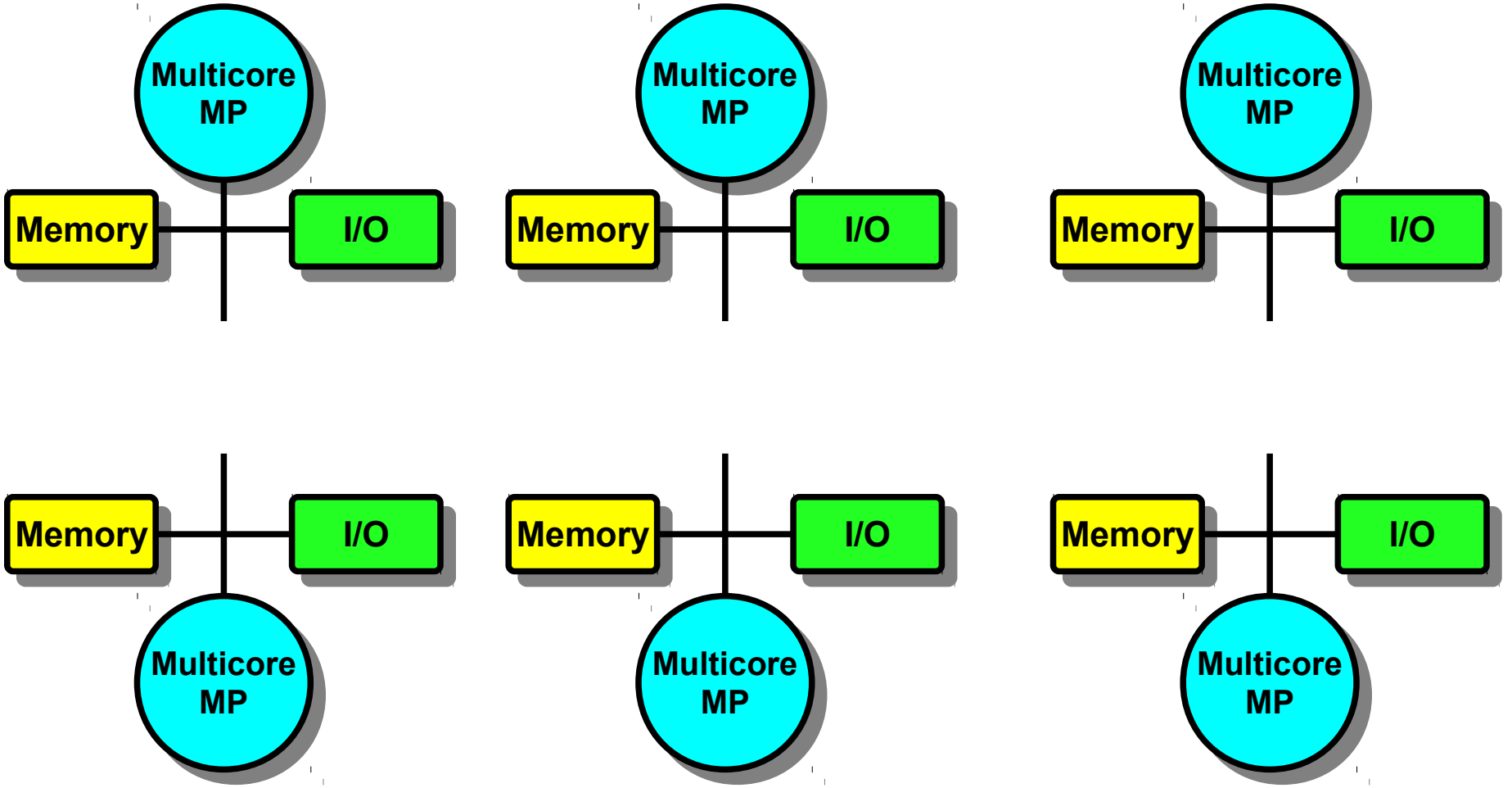
Symmetric Multiprocessor (SMP)



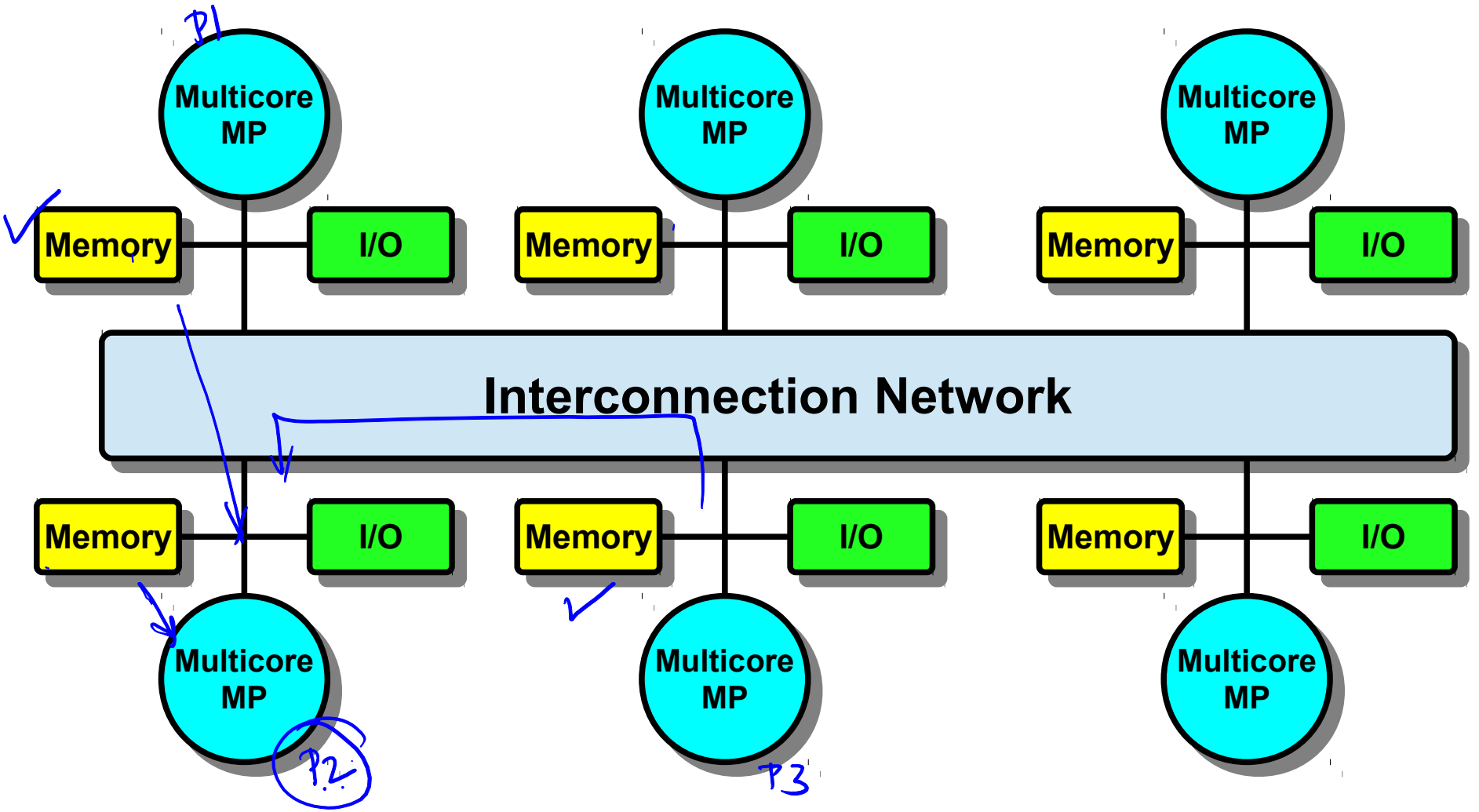
Distributed Shared Memory



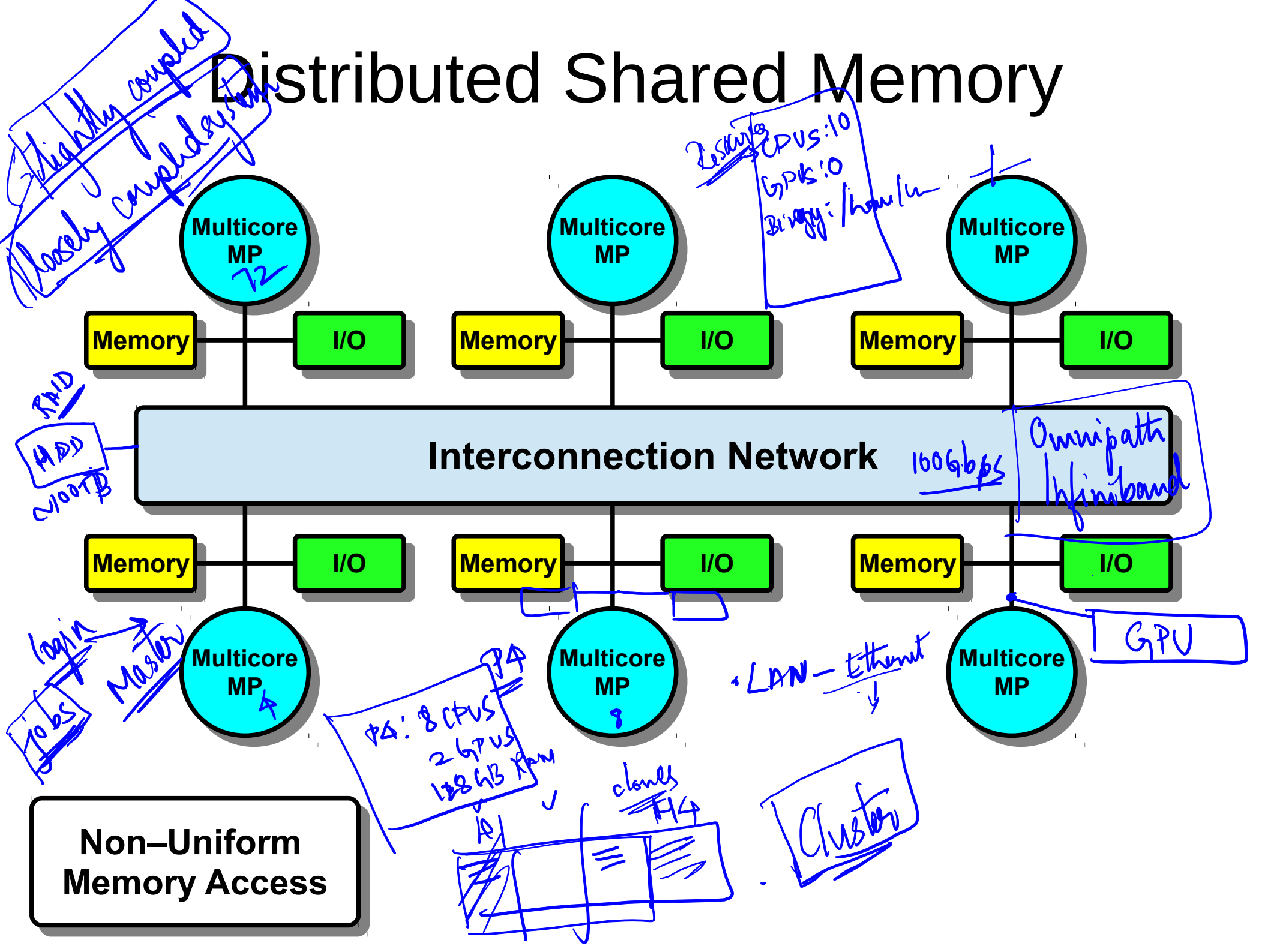
Distributed Shared Memory



Distributed Shared Memory



Distributed Shared Memory



Distributed Shared Memory

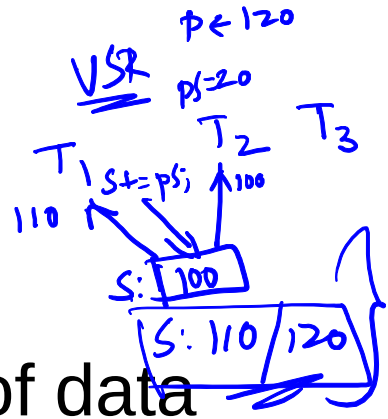
- Scalability
- High memory bandwidth demands
- Low memory access latency to local memory
- Communication infrastructure is complex

Shared Memory vs. Message Passing

- **Shared Memory Machine**: processors share the same physical address space
  -
  -
- **Message Passing Machine:** Memory is private
  -
  -

Shared Memory vs. Message Passing

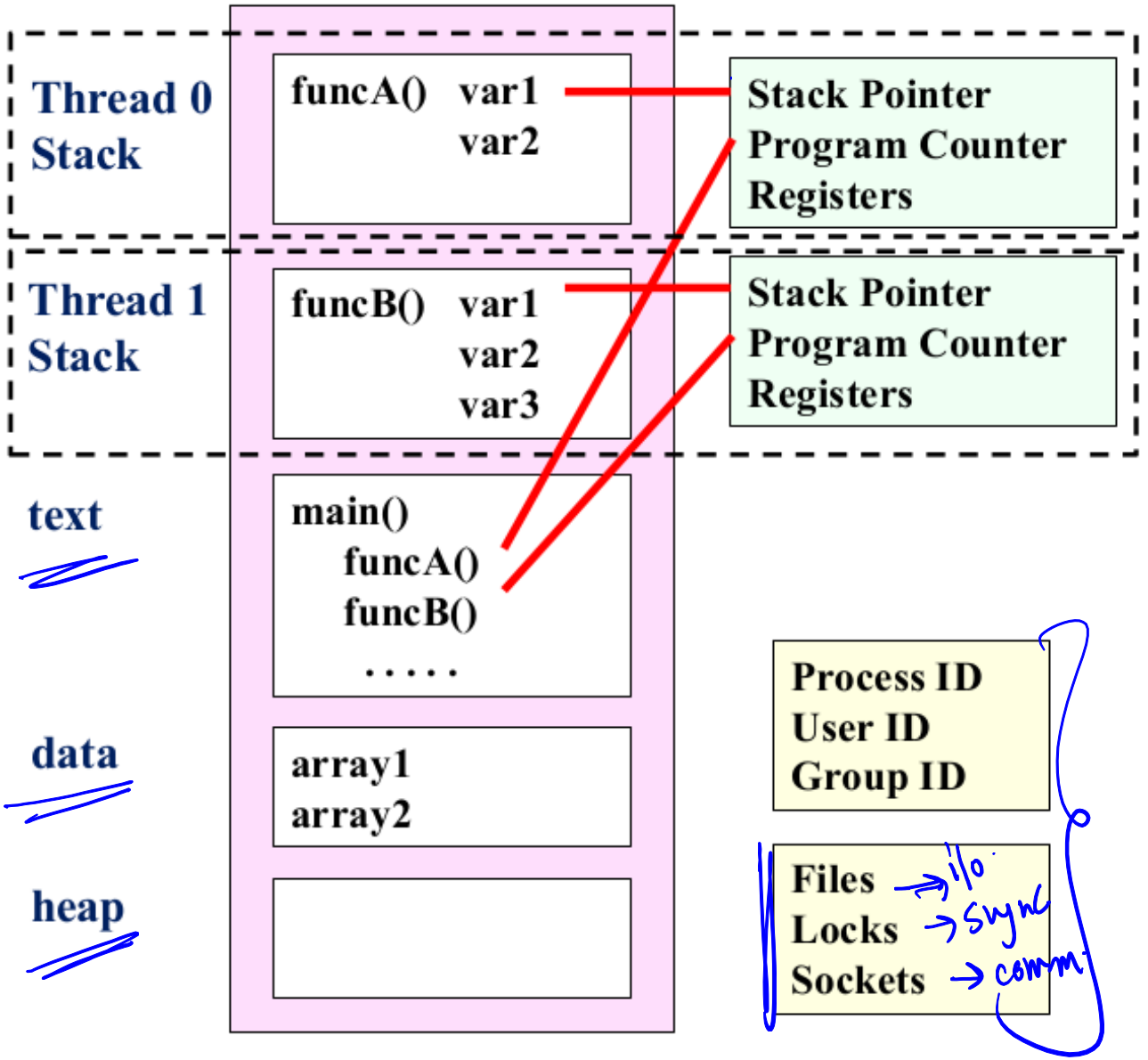
- **Shared Memory Machine**: processors share the same physical address space
  - Implicit communication via loads and stores
  - Needs locks, fences and critical sections to synchronize access
  - No need to know Destination on generation of data (can store in memory and user of data can pick up later)
  - Easy for multiple threads accessing a shared table
  - Eg. OpenMP
- **Message Passing Machine:** Memory is private



Shared Memory vs. Message Passing

- **Shared Memory Machine**: processors share the same physical address space
- **Message Passing Machine:** Memory is private
  - Explicit send/receive to communicate
  - Message contains data and synchronization
  - Need to know Destination on generation of data (send)
  - Easy for Producer-Consumer
  - Eg. MPI

Threads vs. Processes

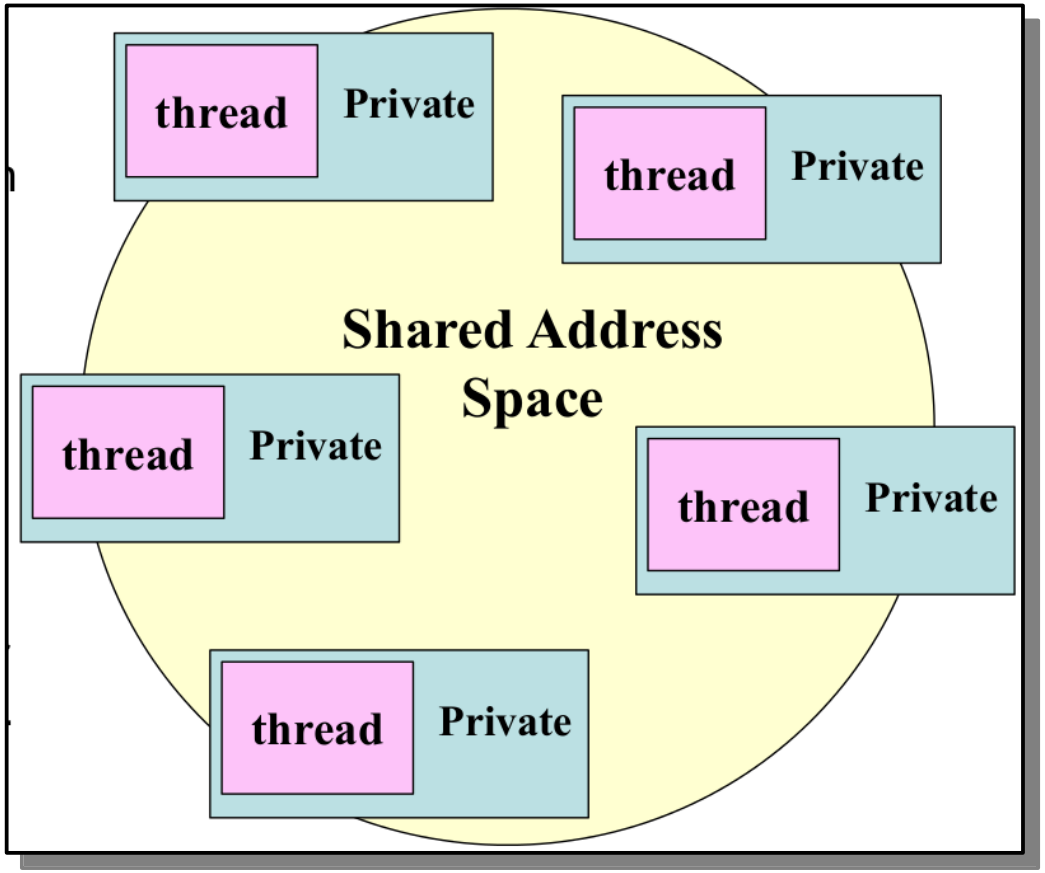


Threads:

- Threads are "light weight processes"
- Threads share Process state among multiple threads ... this greatly reduces the cost of switching context.

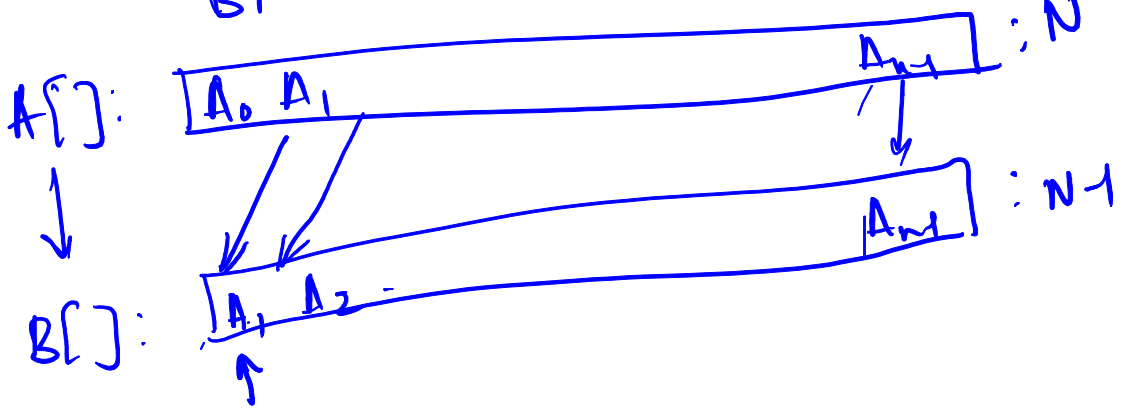
Shared Memory Program

- One process, multiple threads.
- Threads interact through reads/writes to a shared address space.
- OS scheduler decides when to run which threads. Interleaved for fairness.
- Synchronization to assure every legal order results in correct results.



$B_0 = A_0 + A_1$;

$B_1 = A_1 + A_2$;




```
{ for (i: 0 -> N-2)
    B[i] = A[i+1];
}
```

tasks: N-1

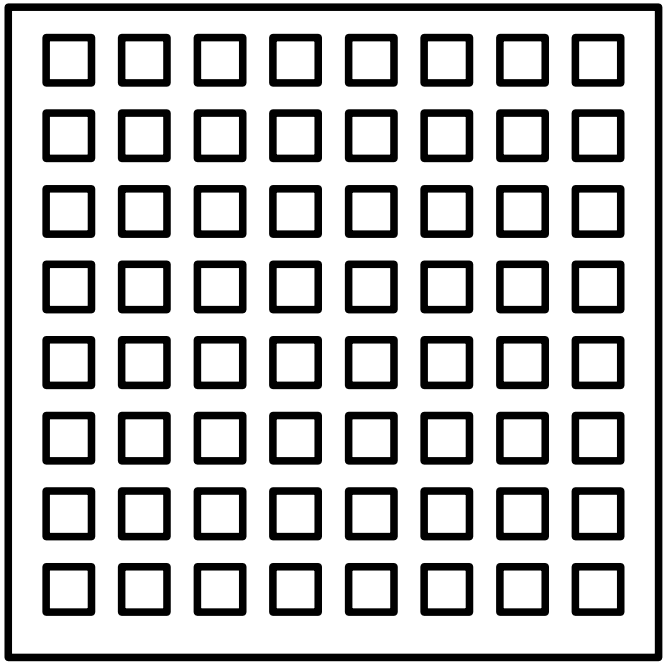
indep- : N-1 tid

Threads: N-1 : 0, 1, ..., N-2

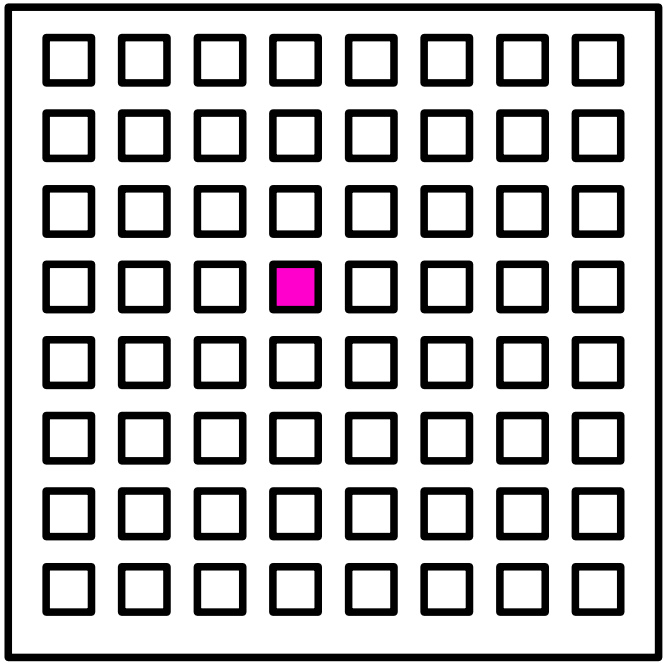
task per thread: update 1 element in B.

; B[tid] = A[tid+1];

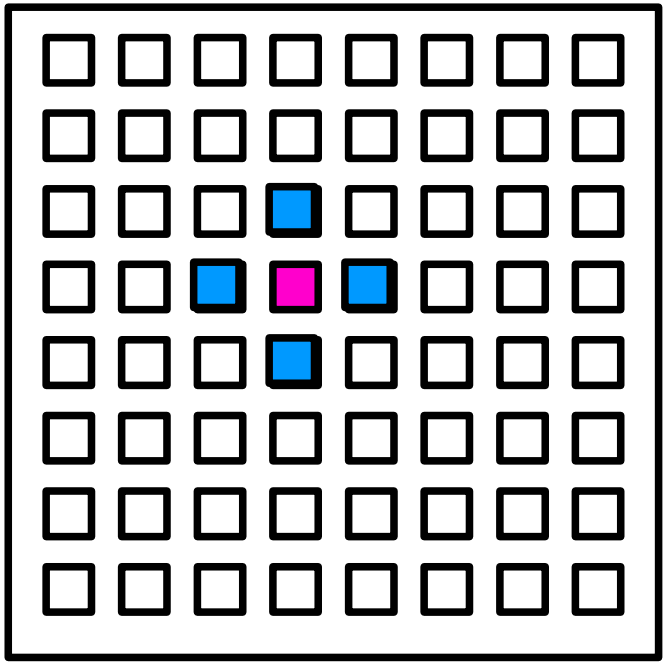
Ocean Kernel



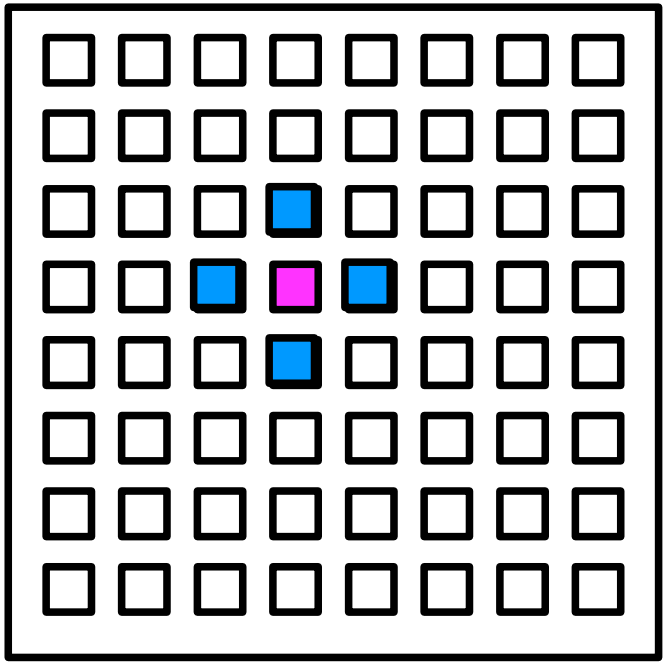
Ocean Kernel



Ocean Kernel

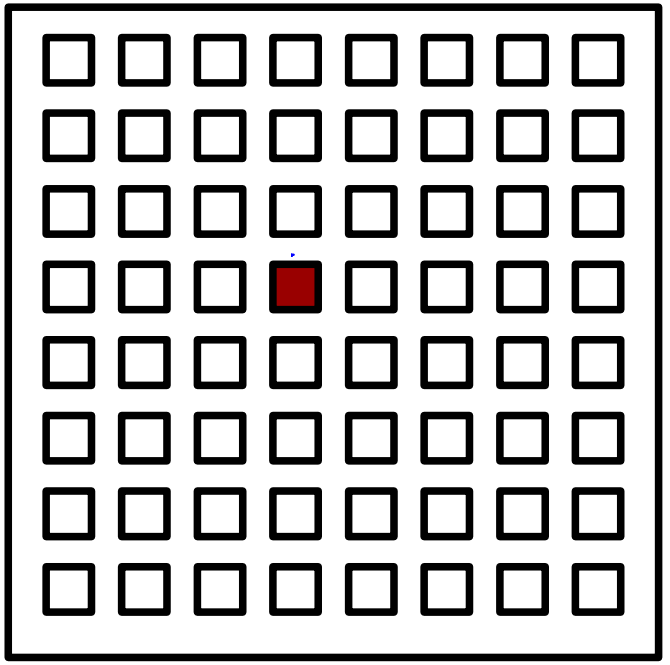


Ocean Kernel



A[i,j] ← 0.2 * (A[i,j] + neighbors);

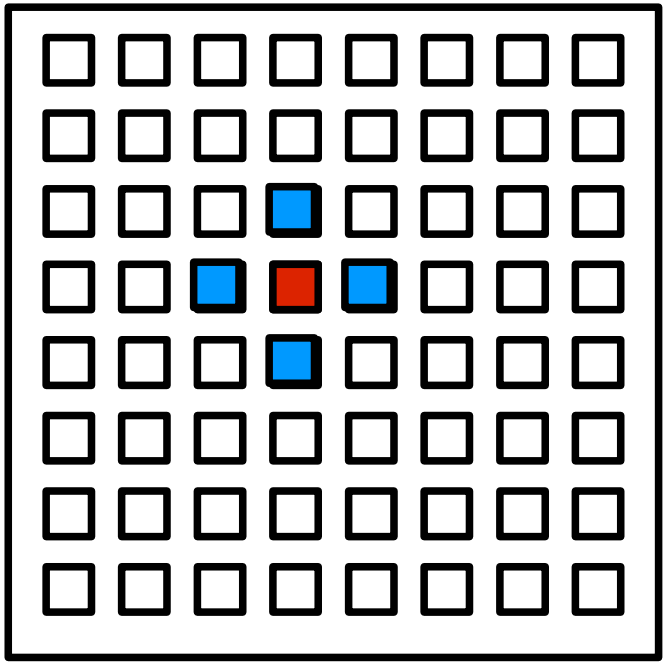
Ocean Kernel



A[i,j] ← 0.2 * (A[i,j] + neighbors);

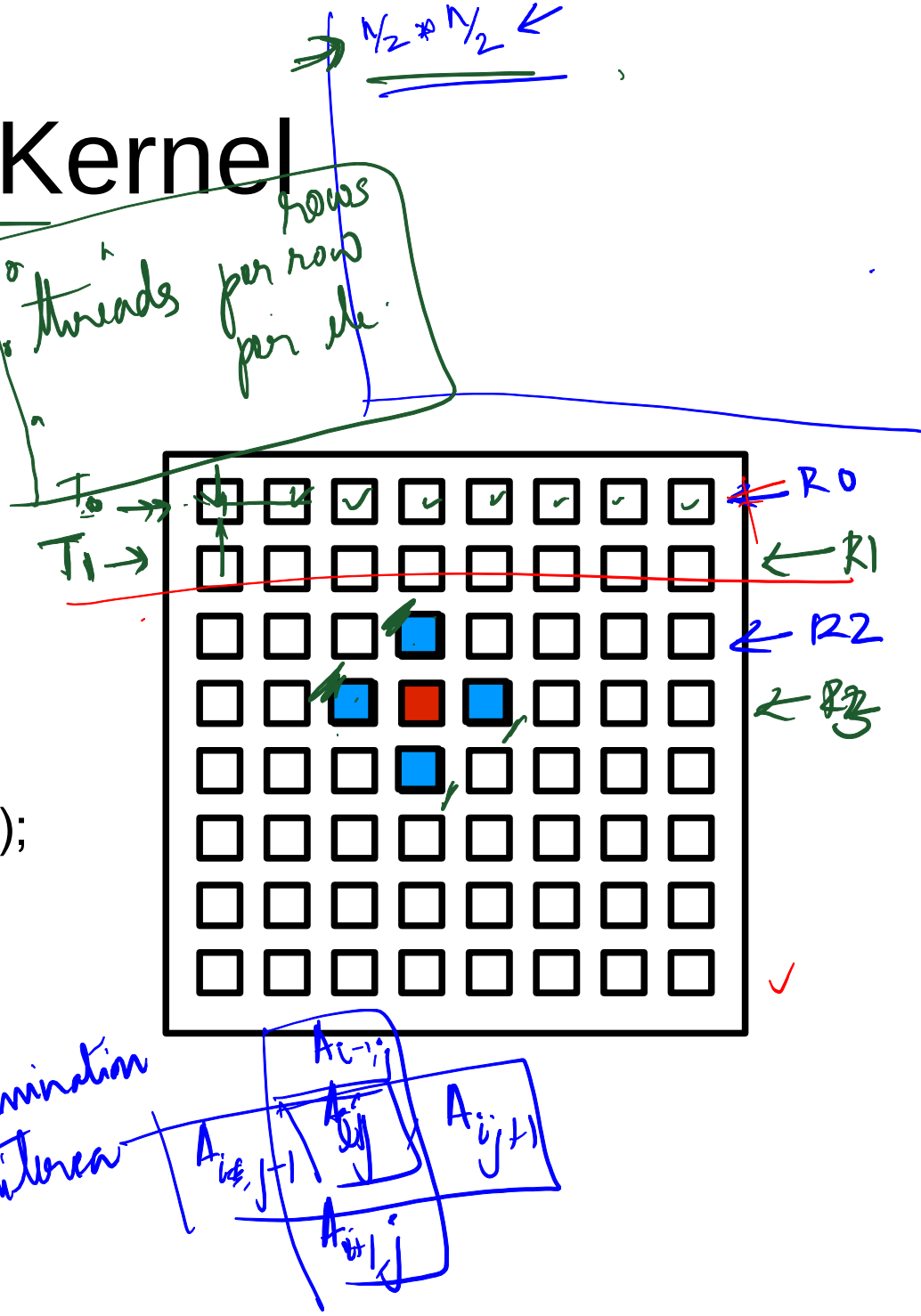
Ocean Kernel

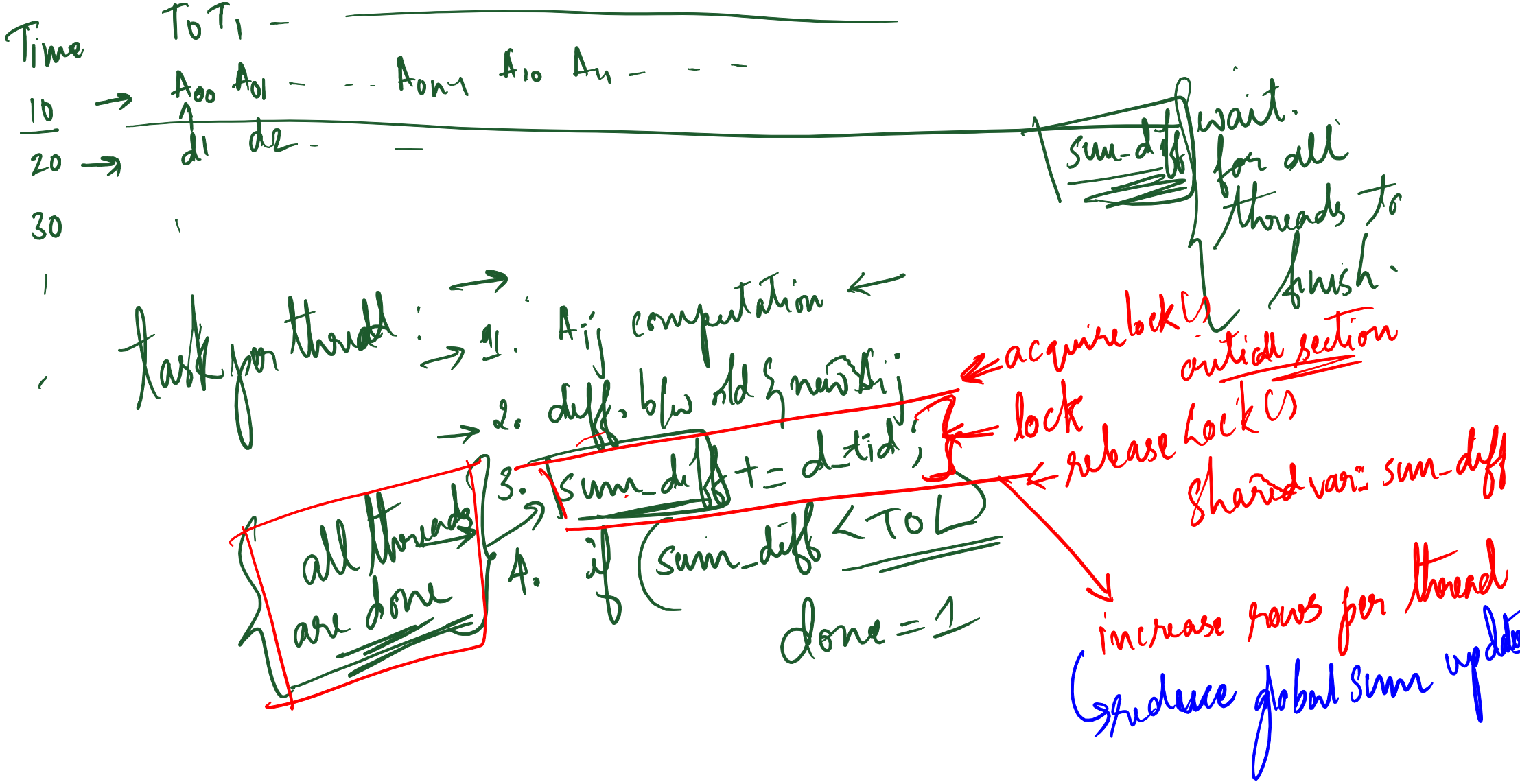
```
Procedure Solve(A)
begin
  for i ← 1 to n do
    for j ← 1 to n do
      temp = A[i,j];
      A[i,j] ← 0.2 * (A[i,j] + neighbors);
      diff += abs(A[i,j] – temp);
    end for
  end for
end procedure
```



Ocean Kernel

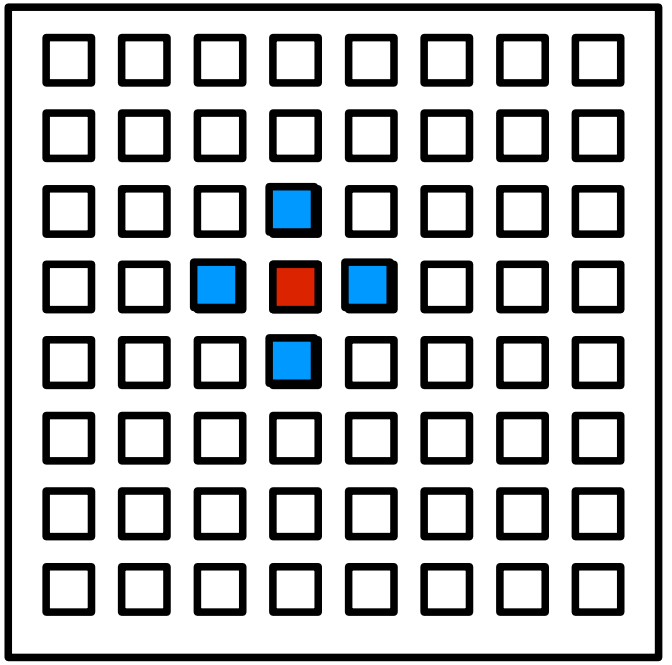
```
Procedure Solve(A)
begin

  while (!done) do
    diff = 0;
    for i ← 1 to n do
      for j ← 1 to n do
        temp = A[i,j];
        A[i,j] ← 0.2 * (A[i,j] + neighbors);
        diff += abs(A[i,j] – temp);
      end for
    end for
    if (diff < TOL) then done = 1;
  end while
end procedure
```




Time
T0 T1 - ——————
10 → A00 A01 - - - A0n-1 A10 A11 - - -
20 → d1 d2 - -
30
sum_diff
wait. for all threads to finish.
task for thread:
1. Aij computation
2. diff. b/w old & new Aij
3. sum_diff += d_tid;
4. if (sum_diff < TOL)
done = 1
all threads are done
acquire lock
lock
release lock
critical section
shared var: sum_diff
increase rows per thread
reduce global sum update

Ocean Kernel

```
Procedure Solve(A)
begin
  diff = done = 0;
  while (!done) do
     diff = 0;
     for i ← 1 to n do
        for j ← 1 to n do
           temp = A[i,j];
           A[i,j] ← 0.2 * (A[i,j] + neighbors);
           diff += abs(A[i,j] – temp);
        end for
     end for
     if (diff < TOL) then done = 1;
  end while
end procedure
```

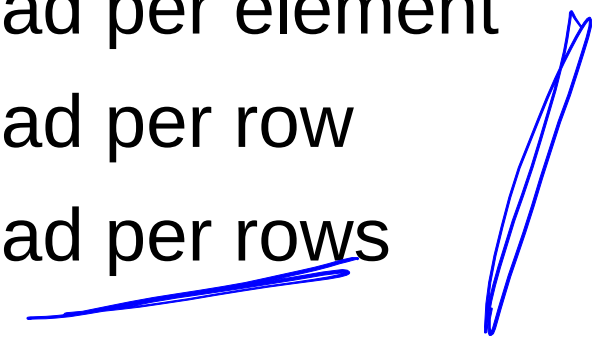


Shared Address – Ocean Kernel

- List the tasks.
- Create how many threads?
-

Shared Address – Ocean Kernel

- List the tasks.
- Create how many threads?
  - Thread per element
  - Thread per row
  - Thread per rows



Shared Address – Ocean Kernel

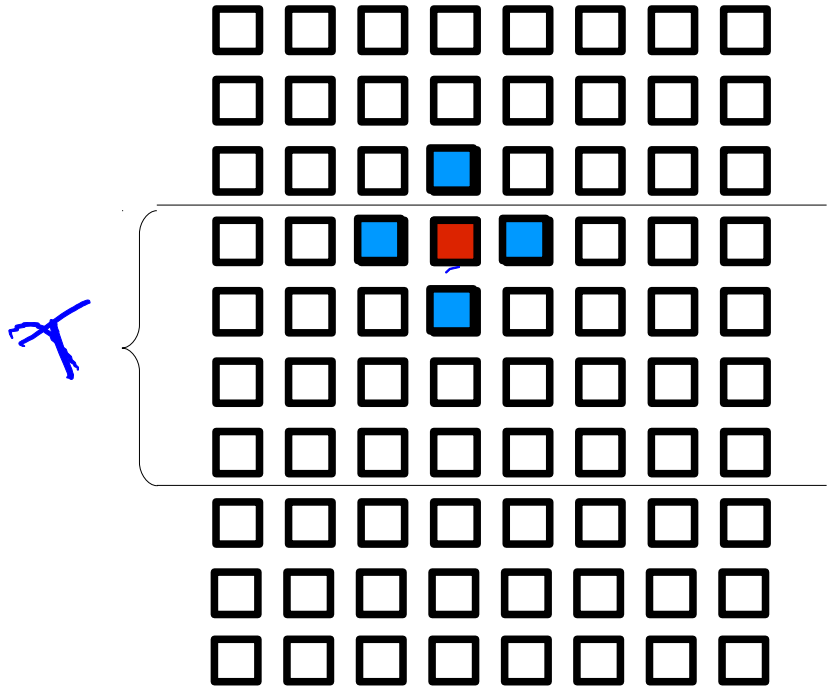
- List the tasks.
- Create how many threads?
- Tasks per thread?

Ocean Kernel – Threads

- One thread for a few rows

Ocean Kernel – Threads

- One thread for a few rows



new $A_{ij} \leftarrow$ avg (A_{ij_old} + 4 neighbours)

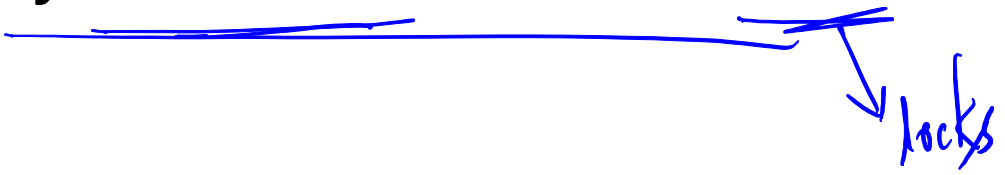
$\leftarrow 0.2$ (

Ocean Kernel – Threads

- One thread for a few rows
- Thread must update the partial sum (diff) after averaging every element in its rows

Ocean Kernel – Threads

- One thread for a few rows
- Thread must update the partial sum (diff)
  - Synchronization, Mutual exclusion



Ocean Kernel – Threads

- One thread for a few rows
- Thread must update the partial sum (diff)
  - Synchronization, Mutual exclusion
  - Lock

Ocean Kernel – Threads

- One thread for a few rows
- Thread must update the partial sum (diff)
  - Synchronization, Mutual exclusion
  - Lock
    - Acquire, Release

Ocean Kernel – Threads

- One thread for a few rows
- Thread must update the partial sum (diff)
- After all threads update diff, check if exceeds threshold

Ocean Kernel – Threads

- One thread for a few rows
- Thread must update the partial sum (diff)
- After all threads update diff, check if exceeds threshold
  - Barrier

Ocean Kernel – Threads

- One thread for a few rows
- Thread must update the partial sum (diff)
- After all threads update diff, check if exceeds threshold
  - Barrier
    - All threads call barrier
    - Every thread waits till peer reaches barrier call

Ocean Kernel – Threads

- One thread for a few rows
- Thread must update the partial sum (diff)
- After all threads update diff, check if exceeds threshold
  - Yes – terminate thread
  - No – repeat iteration

Shared Address – Ocean Kernel

Each thread executes instructions from a procedure – Solve(A)

procedure Solve(A)

Shared Address – Ocean Kernel

procedure Solve(A)

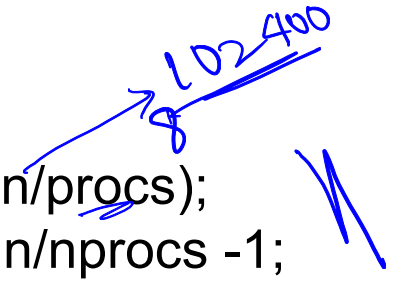
Thread id

Thread identifies its start row and end row

Shared Address – Ocean Kernel

Thread identifies its start row and end row

```
procedure Solve(A)

    int mymin = 1 + (pid * n/procs);
    int mymax = mymin + n/nprocs -1;
```



Shared Address – Ocean Kernel

```
procedure Solve(A)

    int mymin = 1 + (pid * n/procs);
    int mymax = mymin + n/nprocs -1;
```

Read elements and its neighbors
Average
Accumulate partial sum

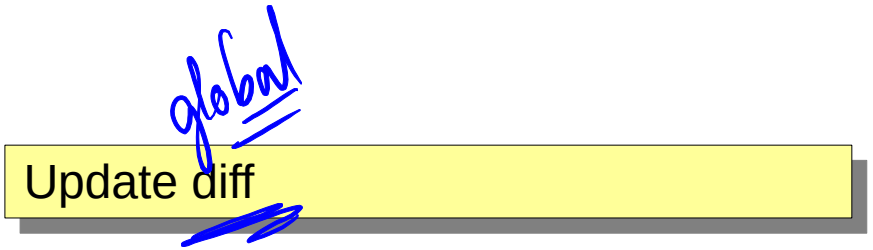
Shared Address – Ocean Kernel

Read elements and its neighbours
Average
Accumulate partial sum

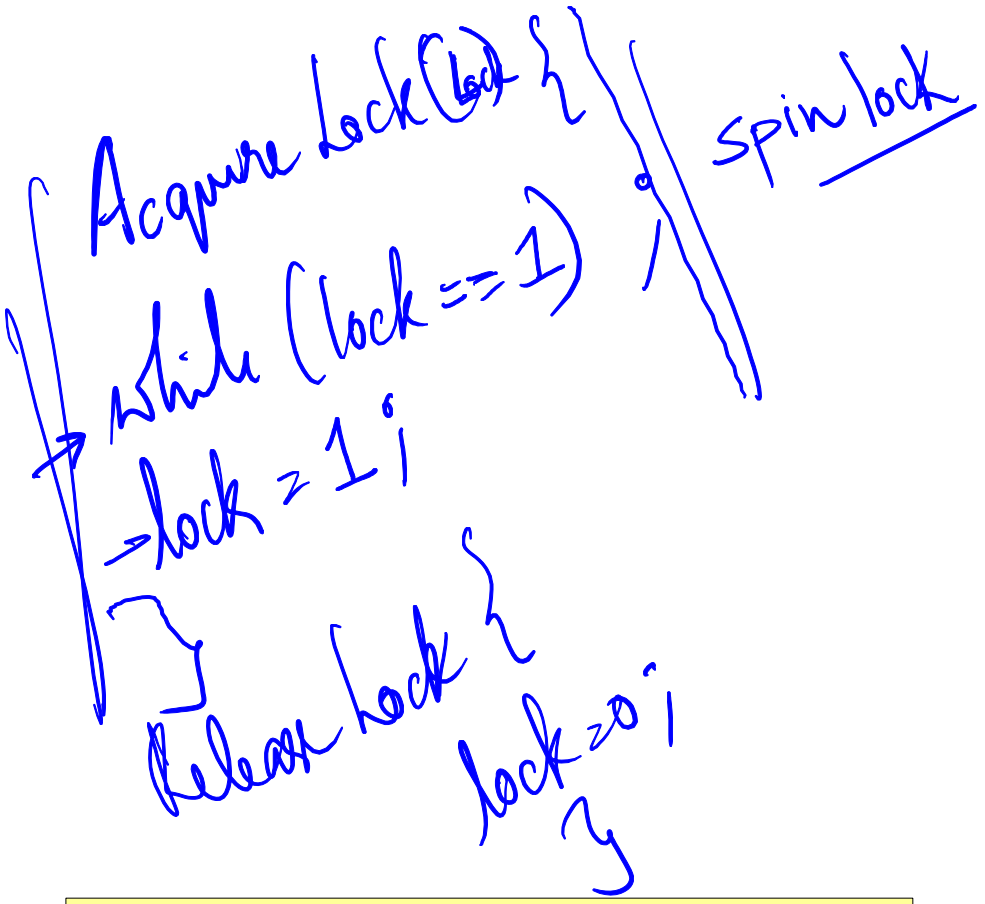
```
procedure Solve(A)

  int mymin = 1 + (pid * n/procs);
  int mymax = mymin + n/nprocs -1;

  for i ← mymin to mymax
    for j ← 1 to n do
      /* avg neighbours */
      /* accumulate mydiff */
    endfor
  endfor
```

Shared Address – Ocean Kernel

```
procedure Solve(A)

  int mymin = 1 + (pid * n/procs);
  int mymax = mymin + n/nprocs -1;


  for i ← mymin to mymax
    for j ← 1 to n do
      /* avg neighbours */
      /* accumulate mydiff */
    endfor
  endfor
```



Shared Address – Ocean Kernel



Update diff

```
procedure Solve(A)

  int mymin = 1 + (pid * n/procs);
  int mymax = mymin + n/nprocs -1;


  for i ← mymin to mymax
    for j ← 1 to n do
      /* avg neighbours */
      /* accumulate mydiff */
    endfor
  endfor
  LOCK(diff_lock);
  diff += mydiff;
  UNLOCK(diff_lock);
```

Shared Address – Ocean Kernel

```
procedure Solve(A)

   int mymin = 1 + (pid * n/procs);
   int mymax = mymin + n/nprocs -1;


   for i ← mymin to mymax
      for j ← 1 to n do
         /* avg neighbours */
         /* accumulate mydiff */
      endfor
   endfor
   LOCK(diff_lock);
   diff += mydiff;
   UNLOCK(diff_lock);
```

Wait for all threads to update diff

Shared Address – Ocean Kernel

```
procedure Solve(A)

   int mymin = 1 + (pid * n/procs);
   int mymax = mymin + n/nprocs -1;


   for i ← mymin to mymax
      for j ← 1 to n do
         /* avg neighbours */
         /* accumulate mydiff */
      endfor
   endfor
   LOCK(diff_lock);
   diff += mydiff;
   UNLOCK(diff_lock);
   BARRIER (bar1, nprocs);
```

Wait for all threads to update diff

Shared Address – Ocean Kernel

```
procedure Solve(A)

   int mymin = 1 + (pid * n/procs);
   int mymax = mymin + n/nprocs -1;


   for i ← mymin to mymax
      for j ← 1 to n do
         /* avg neighbours */
         /* accumulate mydiff */
      endfor
   endfor
   LOCK(diff_lock);
   diff += mydiff;
   UNLOCK(diff_lock);
   BARRIER (bar1, nprocs);
```

Check for threshold
Terminate or Repeat

vs. global diff

global diff is now ready

Shared Address – Ocean Kernel

```
procedure Solve(A)

   int mymin = 1 + (pid * n/procs);
   int mymax = mymin + n/nprocs -1;


      for i ← mymin to mymax
         for j ← 1 to n do
            /* avg neighbours */
            /* accumulate mydiff */
         endfor
      endfor
      LOCK(diff_lock);
      diff += mydiff;
      UNLOCK(diff_lock);
      BARRIER (bar1, nprocs);
      if (diff < TOL) then done = 1;
```

Check for threshold
Terminate or Repeat

Shared Address – Ocean Kernel

```
procedure Solve(A)

   int mymin = 1 + (pid * n/procs);
   int mymax = mymin + n/nprocs -1;
   while (!done) do
      mydiff = diff = 0;

      for i ← mymin to mymax
         for j ← 1 to n do
            /* avg neighbours */
            /* accumulate mydiff */
         endfor
      endfor
      LOCK(diff_lock);
      diff += mydiff;
      UNLOCK(diff_lock);
      BARRIER (bar1, nprocs);
      if (diff < TOL) then done = 1;

   endwhile
```

Check for threshold
Terminate or Repeat

Shared Address – Ocean Kernel

What happens if one thread runs ahead and executes

```
procedure Solve(A)

  int mymin = 1 + (pid * n/procs);
  int mymax = mymin + n/nprocs -1;
  while (!done) do
    mydiff = diff = 0;
    for i ← mymin to mymax
      for j ← 1 to n do
        /* avg neighbours */
        /* accumulate mydiff */
      endfor
    endfor
    LOCK(diff_lock);
    diff += mydiff;
    UNLOCK(diff_lock);
    BARRIER (bar1, nprocs);
    if (diff < TOL) then done = 1;
    BARRIER(
  endwhile
```

Shared Address – Ocean Kernel

What happens if one thread runs ahead and executes while other threads are still executing

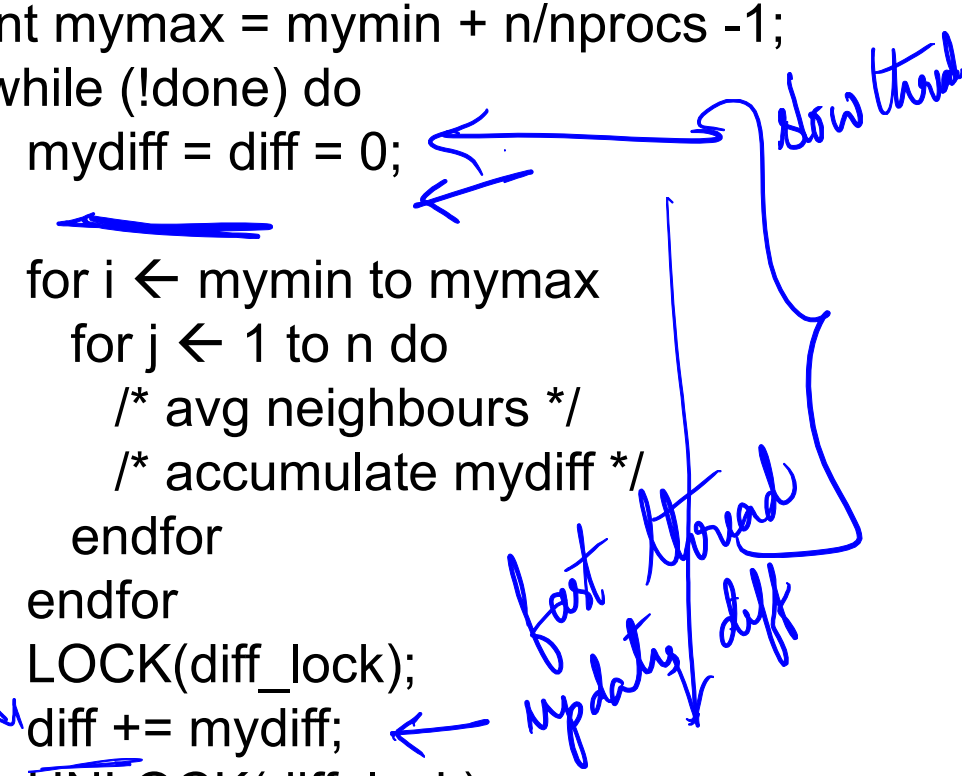
```
procedure Solve(A)

  int mymin = 1 + (pid * n/procs);
  int mymax = mymin + n/nprocs -1;
  while (!done) do
    mydiff = diff = 0;

    for i ← mymin to mymax
      for j ← 1 to n do
        /* avg neighbours */
        /* accumulate mydiff */
      endfor
    endfor
    LOCK(diff_lock);
    diff += mydiff;
    UNLOCK(diff_lock);
    BARRIER (bar1, nprocs);
    if (diff < TOL) then done = 1;
    BARRIER()
  endwhile
```

Shared Address – Ocean Kernel

What happens if one thread runs ahead and executes while other threads are still executing

All thread should have completed threshold checking before any thread moves to the next iteration

```
procedure Solve(A)

  int mymin = 1 + (pid * n/procs);
  int mymax = mymin + n/nprocs -1;
  while (!done) do
    mydiff = diff = 0;

    for i ← mymin to mymax
      for j ← 1 to n do
        /* avg neighbours */
        /* accumulate mydiff */
      endfor
    endfor
    LOCK(diff_lock);
    diff += mydiff;
    UNLOCK(diff_lock);
    BARRIER (bar1, nprocs);
    if (diff < TOL) then done = 1;

  endwhile
```



Shared Address – Ocean Kernel

What happens if one thread runs ahead and executes while other threads are still executing

All thread should have completed threshold checking before any thread moves to the next iteration

```
procedure Solve(A)

  int mymin = 1 + (pid * n/procs);
  int mymax = mymin + n/nprocs -1;
  while (!done) do
    mydiff = diff = 0;

    for i ← mymin to mymax
      for j ← 1 to n do
        /* avg neighbours */
        /* accumulate mydiff */
      endfor
    endfor
    LOCK(diff_lock);
    diff += mydiff;
    UNLOCK(diff_lock);
    BARRIER (bar1, nprocs);
    if (diff < TOL) then done = 1;
    BARRIER (bar1, nprocs);
  endwhile
```

Shared Address – Ocean Kernel

```
procedure Solve(A)

  int mymin = 1 + (pid * n/procs);
  int mymax = mymin + n/nprocs -1;
  while (!done) do
    mydiff = diff = 0;

    for i ← mymin to mymax
      for j ← 1 to n do
        /* avg neighbours */
        /* accumulate mydiff */
      endfor
    endfor
    LOCK(diff_lock);
    diff += mydiff;
    UNLOCK(diff_lock);
    BARRIER (bar1, nprocs);
    if (diff < TOL) then done = 1;
    BARRIER (bar1, nprocs);
  endwhile
```

Shared Address – Ocean Kernel

What happens if one thread runs ahead and executes

```
procedure Solve(A)

   int mymin = 1 + (pid * n/procs);
   int mymax = mymin + n/nprocs -1;
   while (!done) do
      mydiff = diff = 0;

      for i ← mymin to mymax
         for j ← 1 to n do
            /* avg neighbours */
            /* accumulate mydiff */
         endfor
      endfor
      LOCK(diff_lock);
      diff += mydiff;
      UNLOCK(diff_lock);
      BARRIER (bar1, nprocs);
      if (diff < TOL) then done = 1;
      BARRIER (bar1, nprocs);
   endwhile
```

Shared Address – Ocean Kernel

What happens if one thread runs ahead and executes while other threads are executing

```
procedure Solve(A)

  int mymin = 1 + (pid * n/procs);
  int mymax = mymin + n/nprocs -1;
  while (!done) do
    mydiff = diff = 0;

    for i ← mymin to mymax
      for j ← 1 to n do
        /* avg neighbours */
        /* accumulate mydiff */
      endfor
    endfor
    LOCK(diff_lock);
    diff += mydiff;
    UNLOCK(diff_lock);
    BARRIER (bar1, nprocs);
    if (diff < TOL) then done = 1;
    BARRIER (bar1, nprocs);
  endwhile
```

Shared Address – Ocean Kernel

What happens if one thread runs ahead and executes while other threads are executing

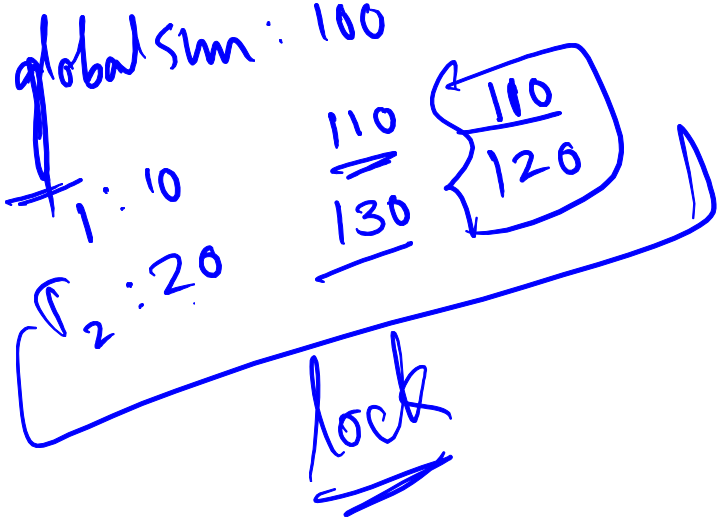
All threads have begin for loops after initializing diff, mydiff

```
procedure Solve(A)

  int mymin = 1 + (pid * n/procs);
  int mymax = mymin + n/nprocs -1;
  while (!done) do
    mydiff = diff = 0;

    for i ← mymin to mymax
      for j ← 1 to n do
        /* avg neighbours */
        /* accumulate mydiff */
      endfor
    endfor
    LOCK(diff_lock);
    diff += mydiff;
    UNLOCK(diff_lock);
    BARRIER (bar1, nprocs);
    if (diff < TOL) then done = 1;
    BARRIER (bar1, nprocs);
  endwhile
```

Shared Address – Ocean Kernel

What happens if one thread runs ahead and executes while other threads are executing

All threads have begin for loops after initializing diff, mydiff




```
procedure Solve(A)

  int mymin = 1 + (pid * n/procs);
  int mymax = mymin + n/nprocs -1;
  while (!done) do
    mydiff = diff = 0;
    BARRIER(bar1,nprocs);
    for i ← mymin to mymax
      for j ← 1 to n do
        /* avg neighbours */
        /* accumulate mydiff */
      endfor
    endfor
    LOCK(diff_lock);
    diff += mydiff;
    UNLOCK(diff_lock);
    BARRIER (bar1, nprocs);
    if (diff < TOL) then done = 1;
    BARRIER (bar1, nprocs);
  endwhile
```

Shared Address – Ocean Kernel

```
procedure Solve(A)
    int i, j, pid, done=0;
    float temp, mydiff=0;
    int mymin = 1 + (pid * n/procs);
    int mymax = mymin + n/nprocs -1;
    while (!done) do
       mydiff = diff = 0;
       BARRIER(bar1,nprocs);
       for i ← mymin to mymax
          for j ← 1 to n do
             …
          endfor
       endfor
       LOCK(diff_lock);
       diff += mydiff;
       UNLOCK(diff_lock);
       BARRIER (bar1, nprocs);
       if (diff < TOL) then done = 1;
       BARRIER (bar1, nprocs);
    endwhile
```

Shared Address – Ocean Kernel

```
main()
begin
   read(n); read(nprocs);
   A ← G_MALLOC();
   initialize (A);
   CREATE (nprocs,Solve,A);
   WAIT_FOR_END (nprocs);
end main
```

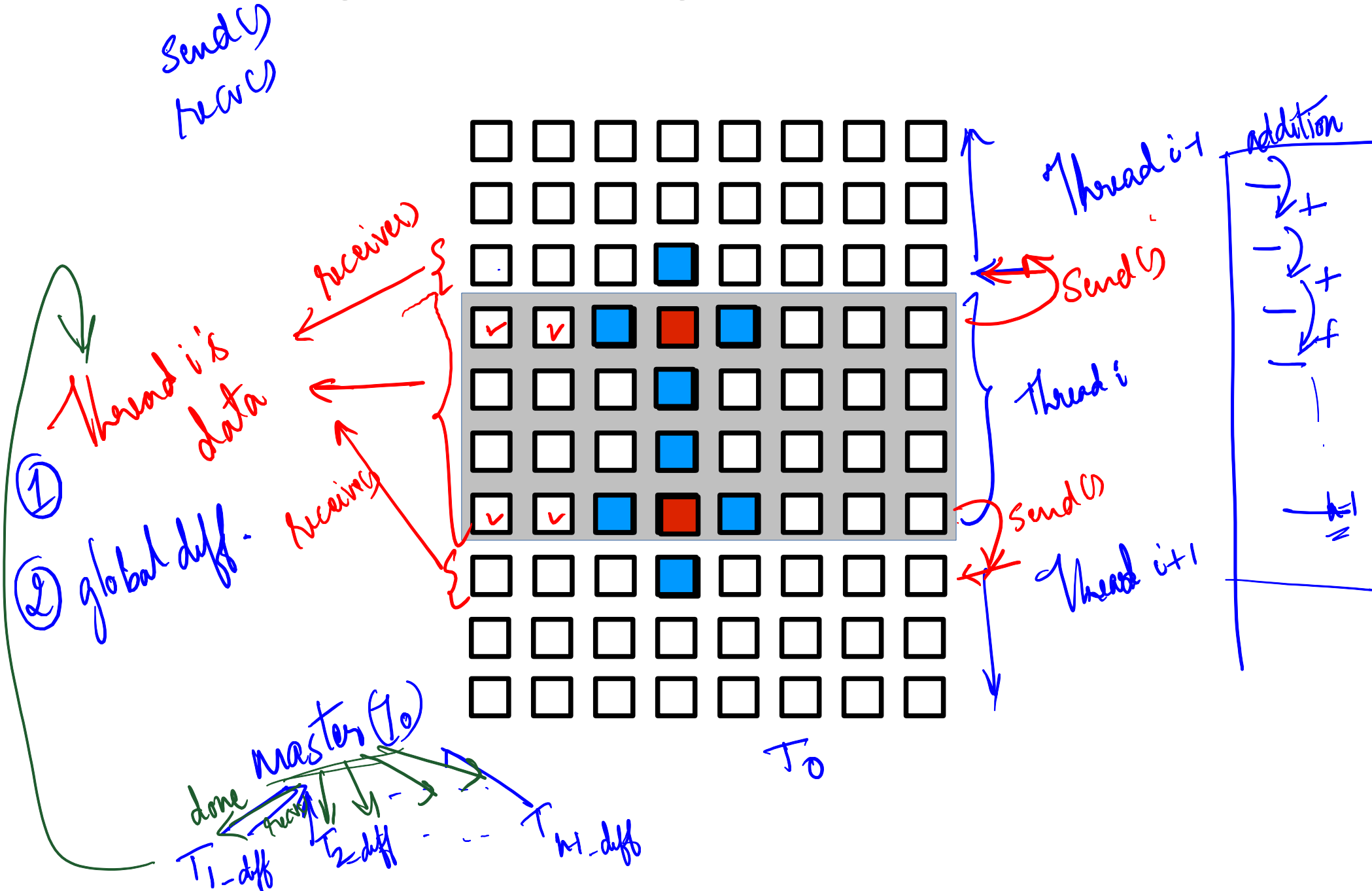
```
procedure Solve(A)
   int i, j, pid, done=0;
   float temp, mydiff=0;
   int mymin = 1 + (pid * n/procs);
   int mymax = mymin + n/nprocs -1;
   while (!done) do
      mydiff = diff = 0;
      BARRIER(bar1,nprocs);
      for i ← mymin to mymax
         for j ← 1 to n do
            …
         endfor
      endfor
      LOCK(diff_lock);
      diff += mydiff;
      UNLOCK(diff_lock);
      BARRIER (bar1, nprocs);
      if (diff < TOL) then done = 1;
      BARRIER (bar1, nprocs);
   endwhile
```

Shared Address – Ocean Kernel

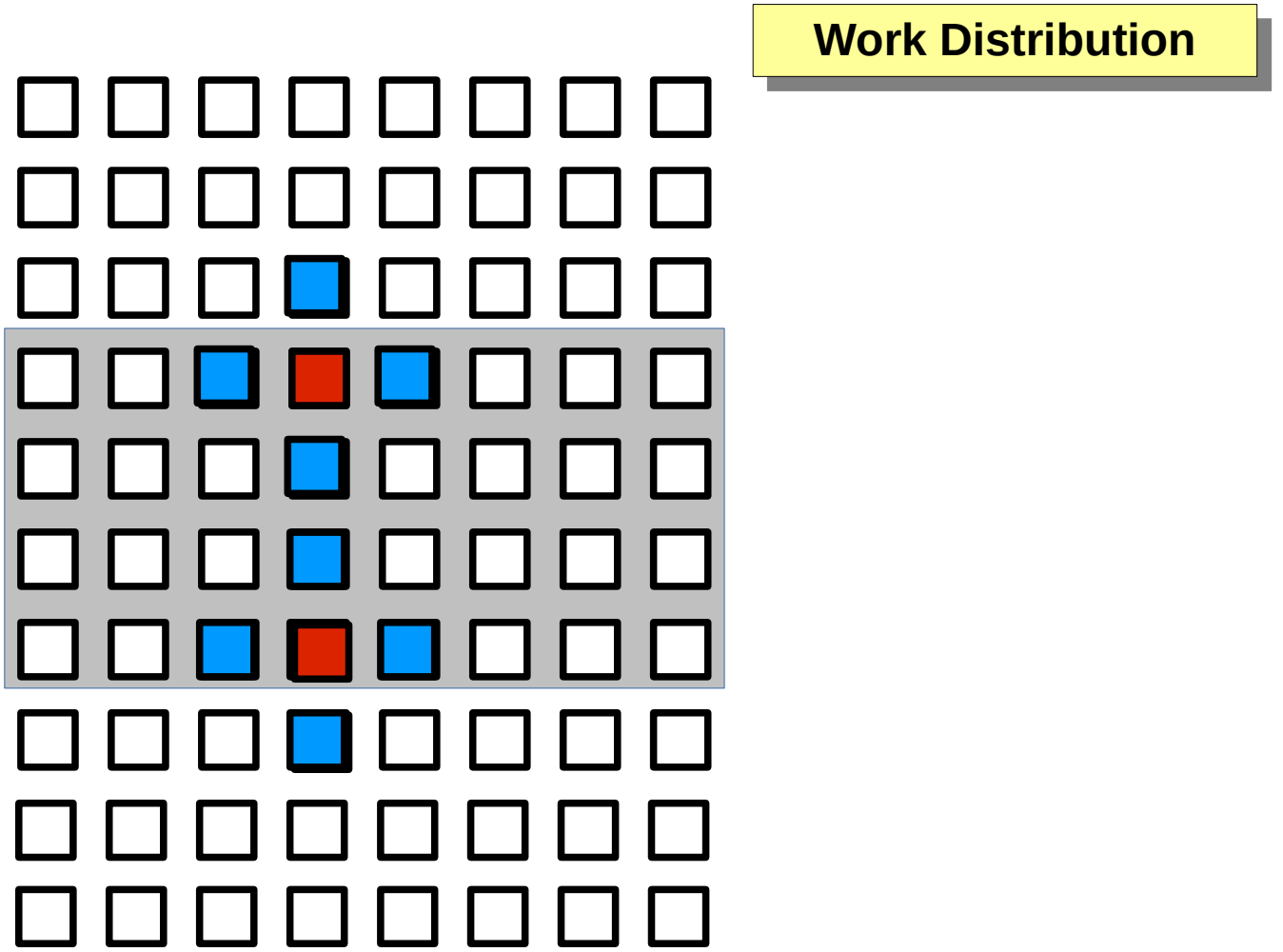
```
int  n, nprocs;
float  **A, diff;
LOCKDEC(diff_lock);
BARDEC(bar1);

main()
begin
   read(n); read(nprocs);
   A ← G_MALLOC();
   initialize (A);
   CREATE (nprocs,Solve,A);
   WAIT_FOR_END (nprocs);
end main
```

```
procedure Solve(A)
   int i, j, pid, done=0;
   float temp, mydiff=0;
   int mymin = 1 + (pid * n/procs);
   int mymax = mymin + n/nprocs -1;
   while (!done) do
      mydiff = diff = 0;
      BARRIER(bar1,nprocs);
      for i ← mymin to mymax
         for j ← 1 to n do
            …
         endfor
      endfor
      LOCK(diff_lock);
      diff += mydiff;
      UNLOCK(diff_lock);
      BARRIER (bar1, nprocs);
      if (diff < TOL) then done = 1;
      BARRIER (bar1, nprocs);
   endwhile
```

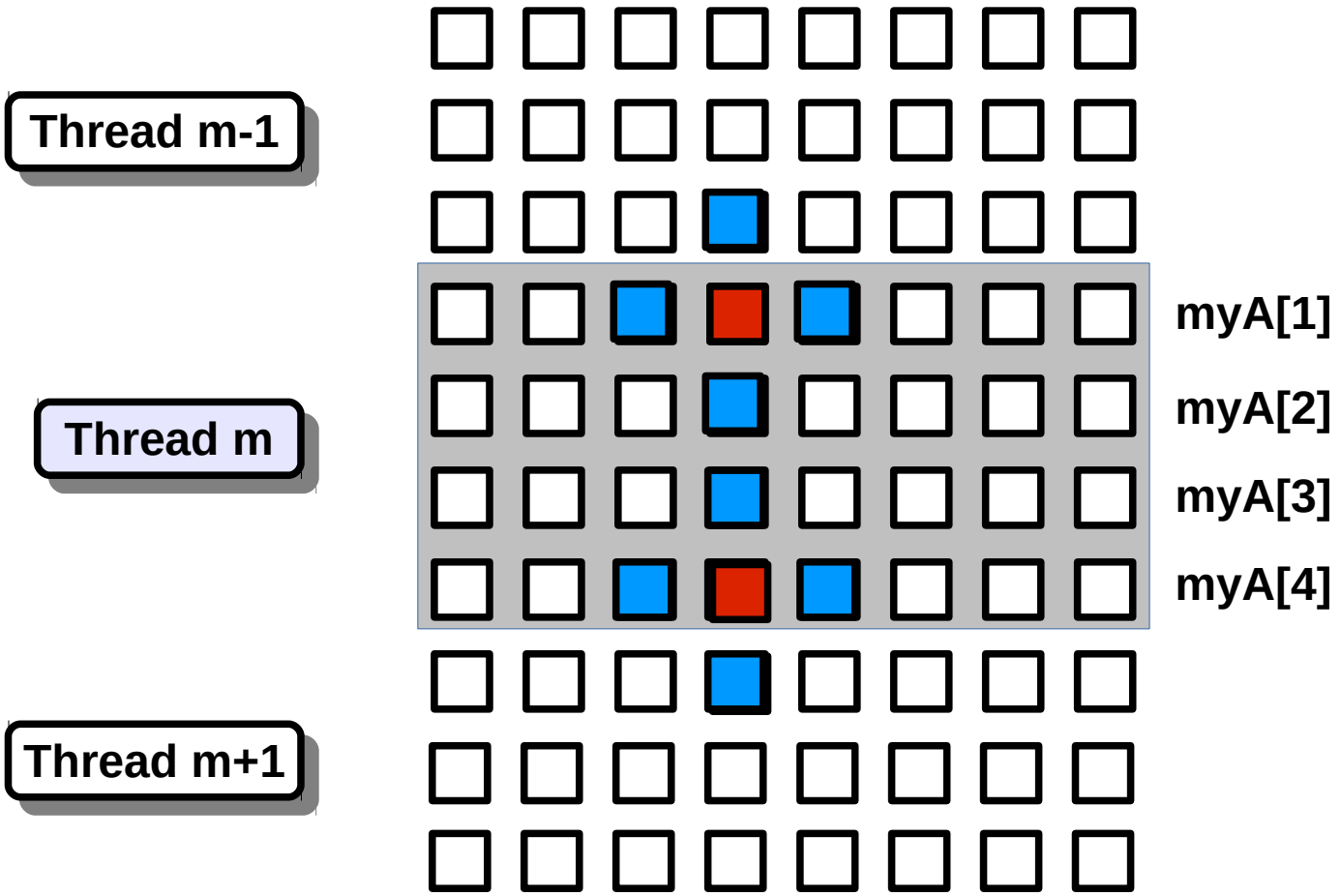
Message Passing – Ocean Kernel



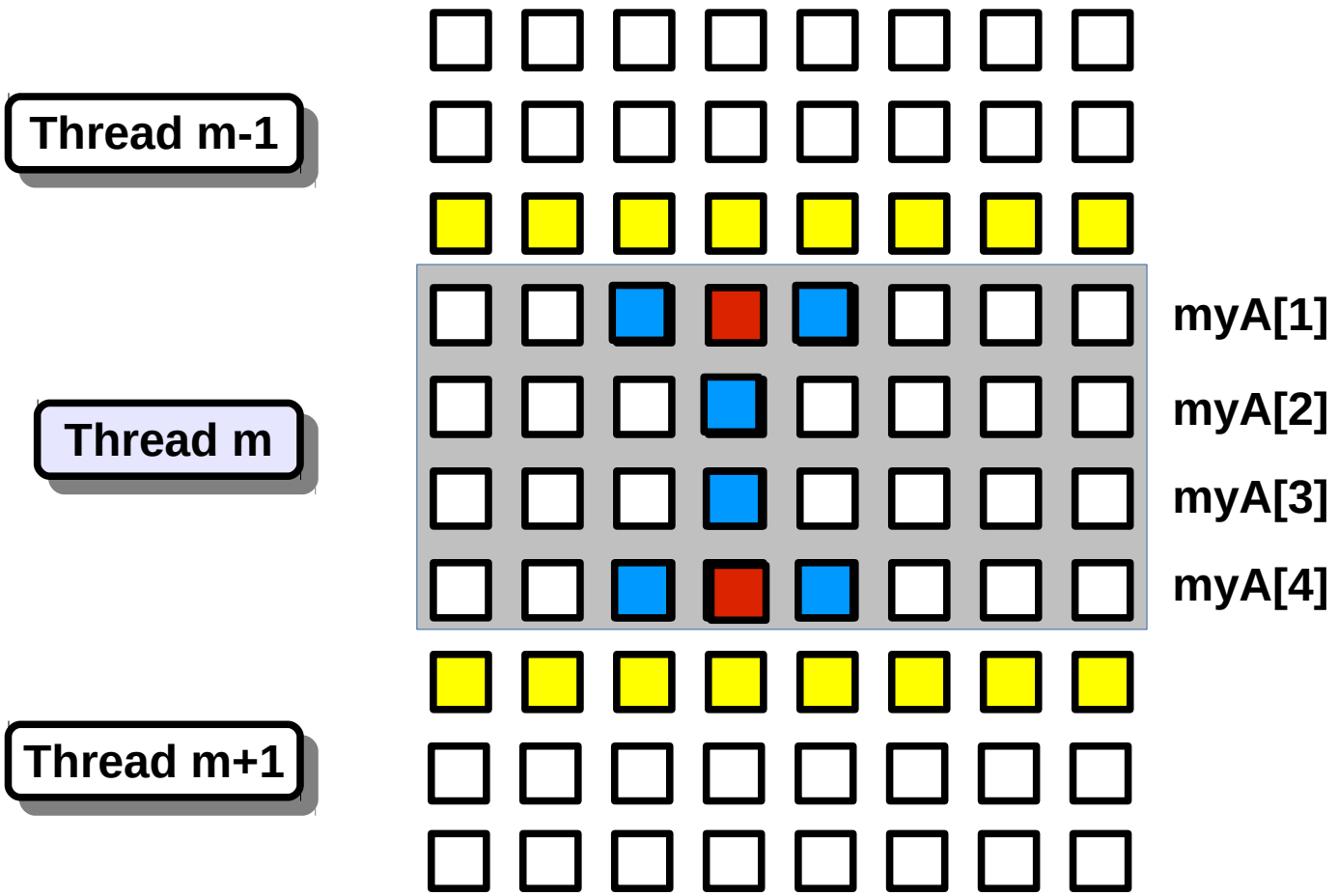
Message Passing – Ocean Kernel



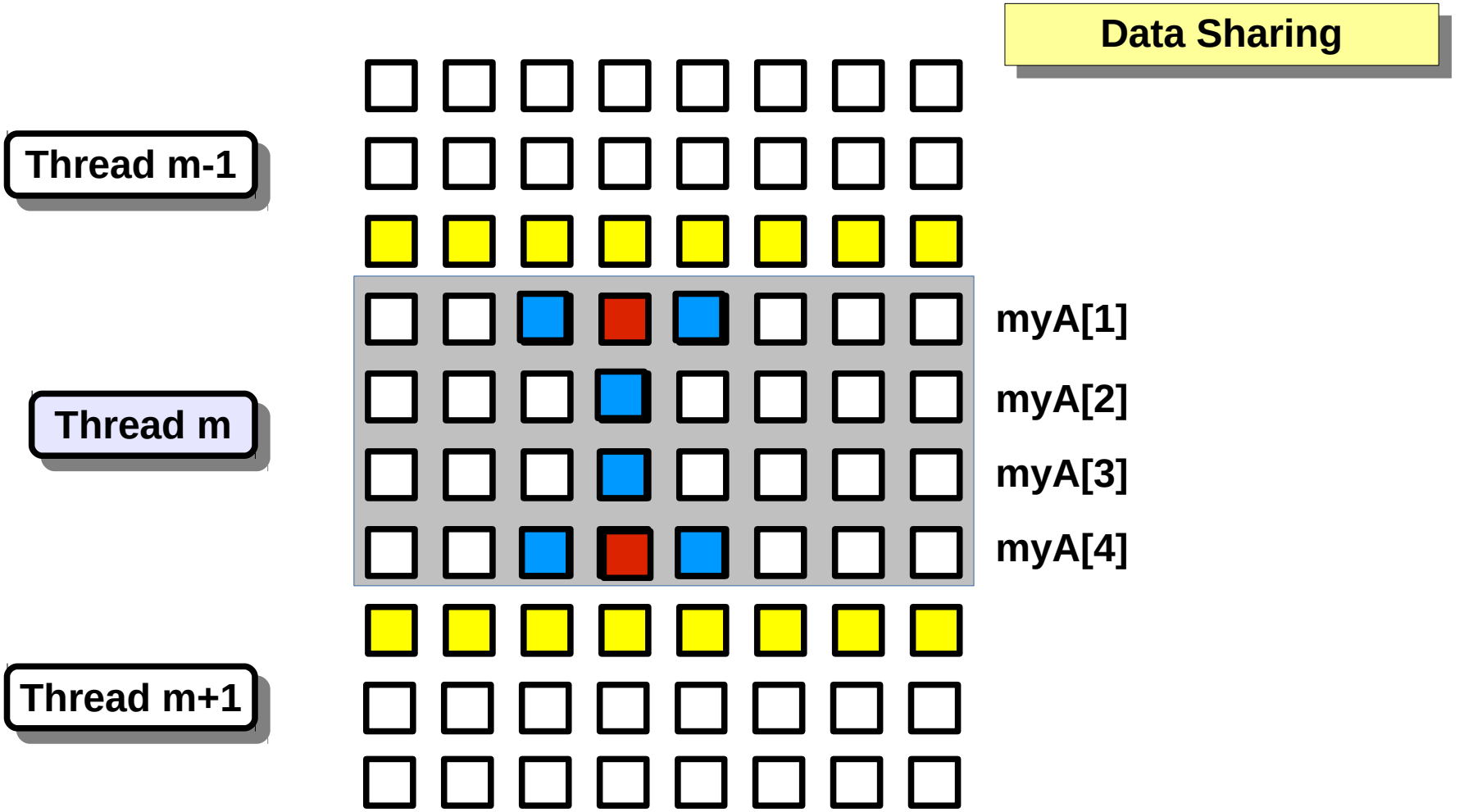
Message Passing – Ocean Kernel



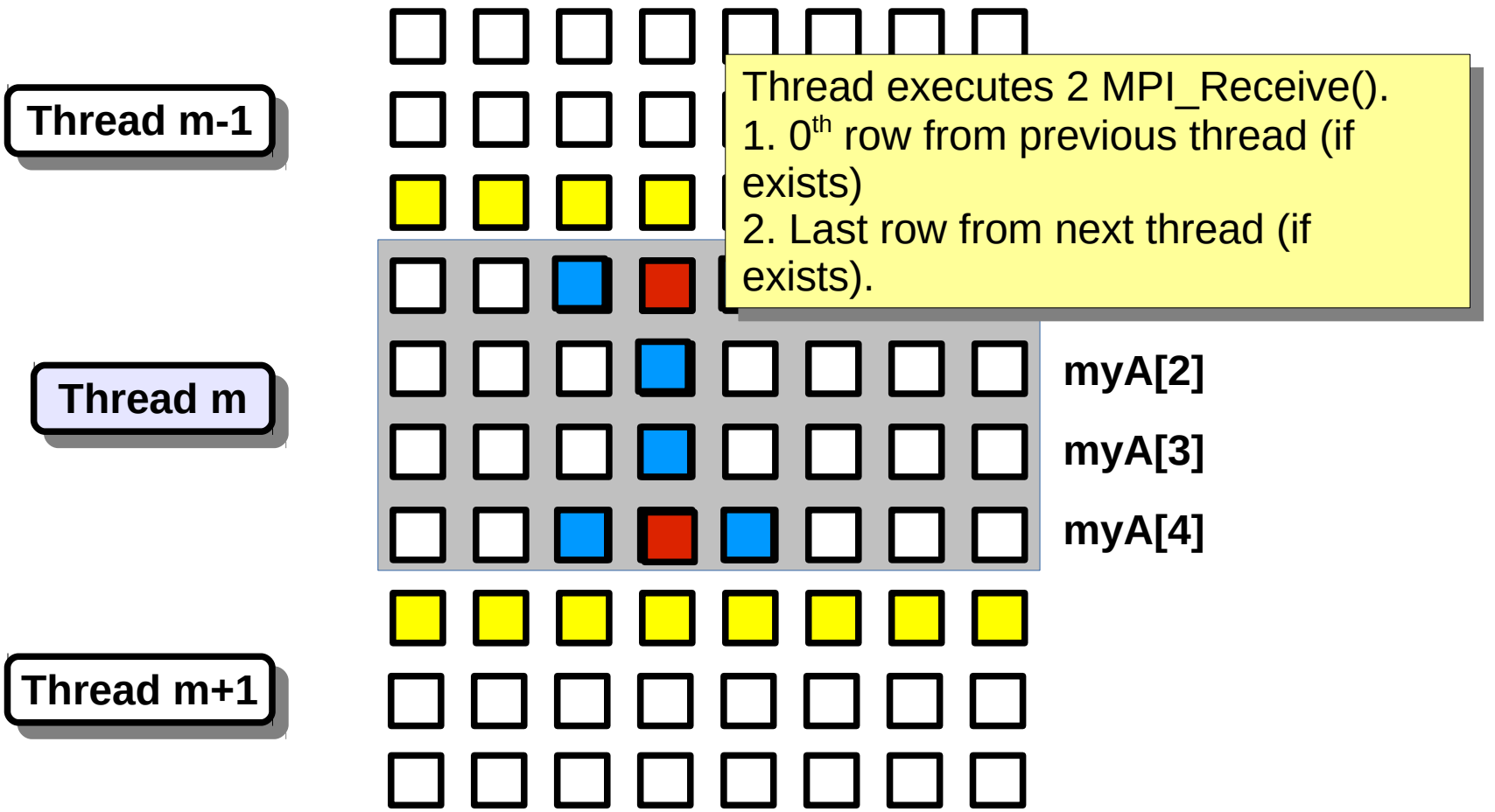
Message Passing – Ocean Kernel



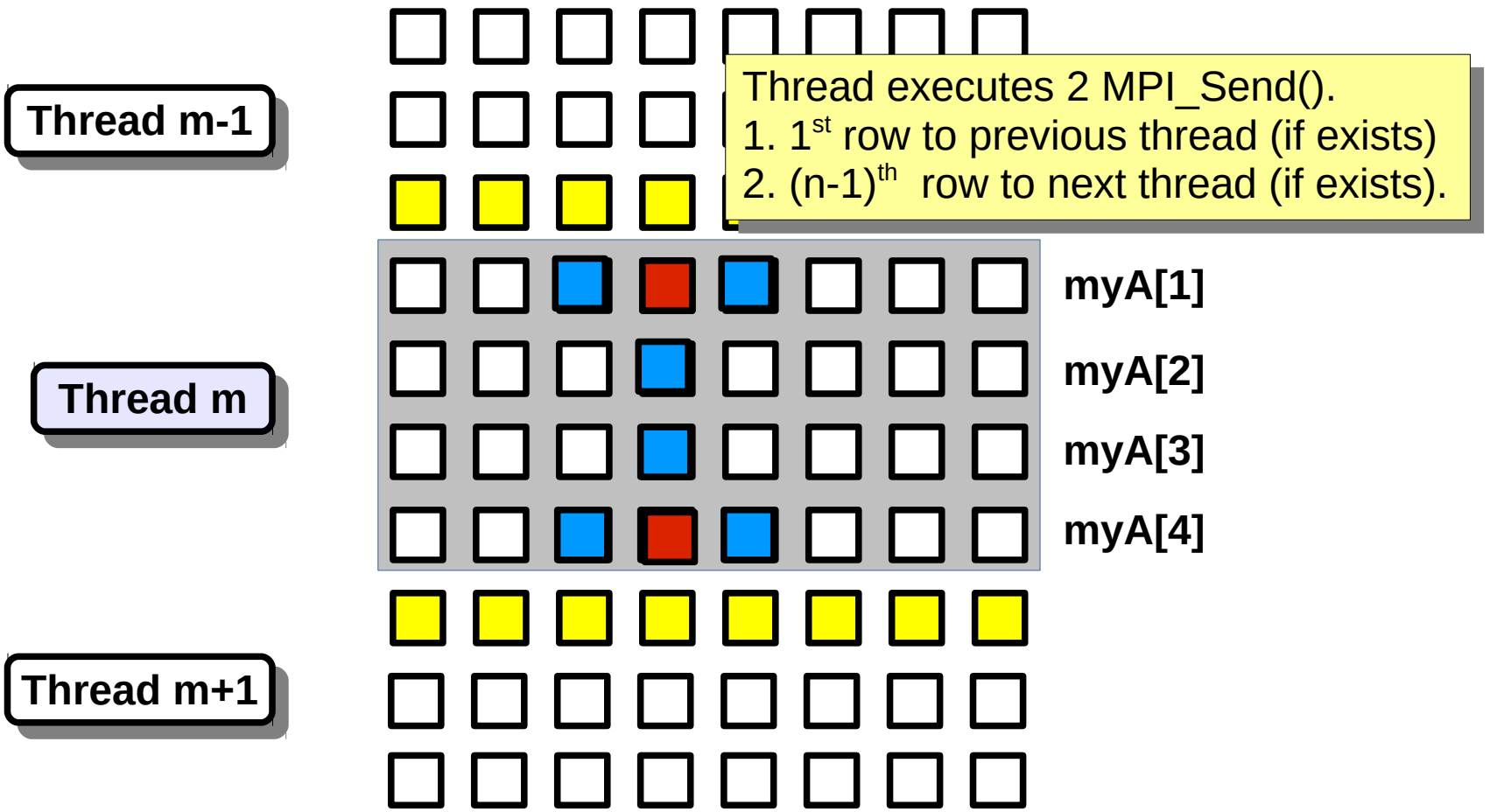
Message Passing – Ocean Kernel



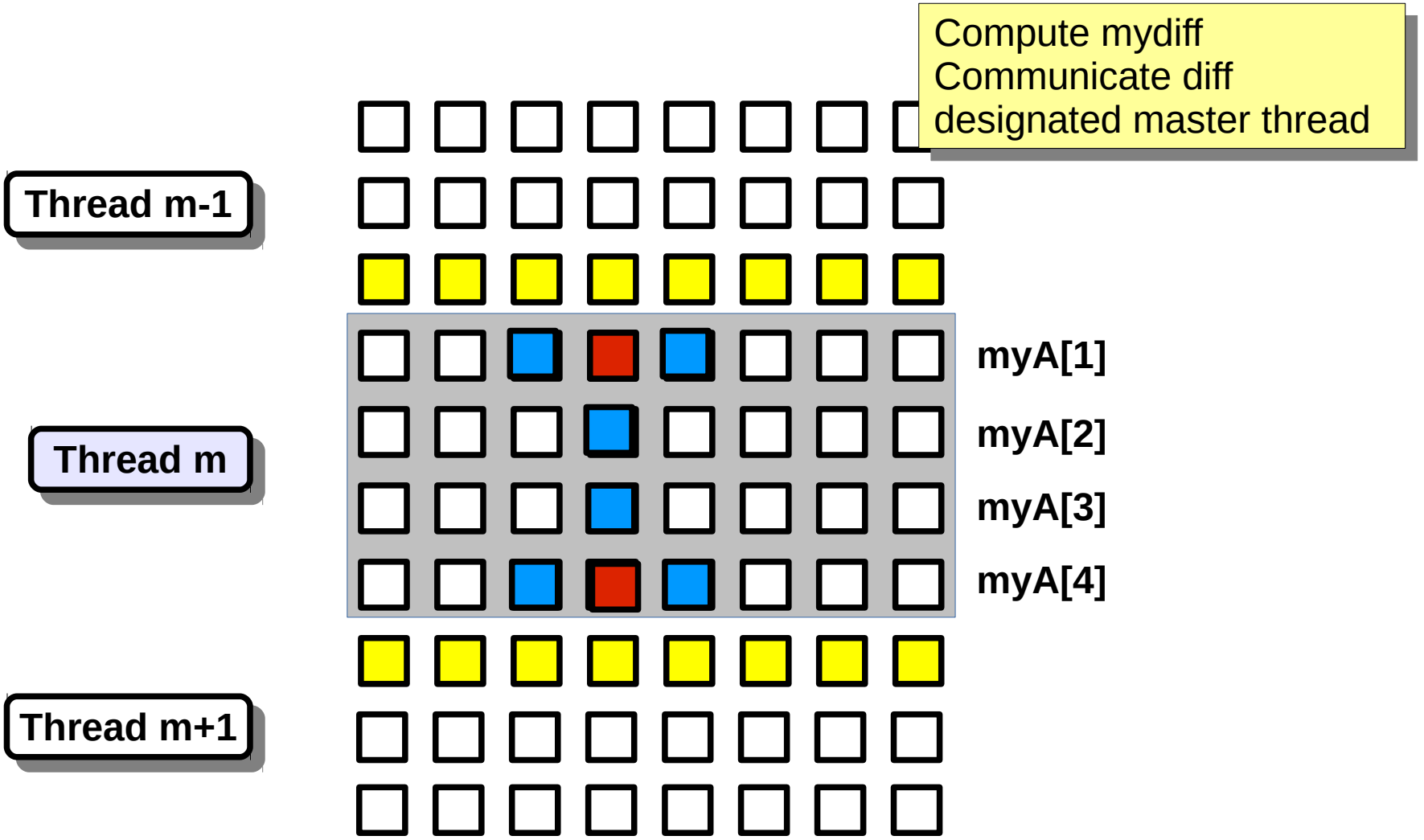
Message Passing – Ocean Kernel



Message Passing – Ocean Kernel

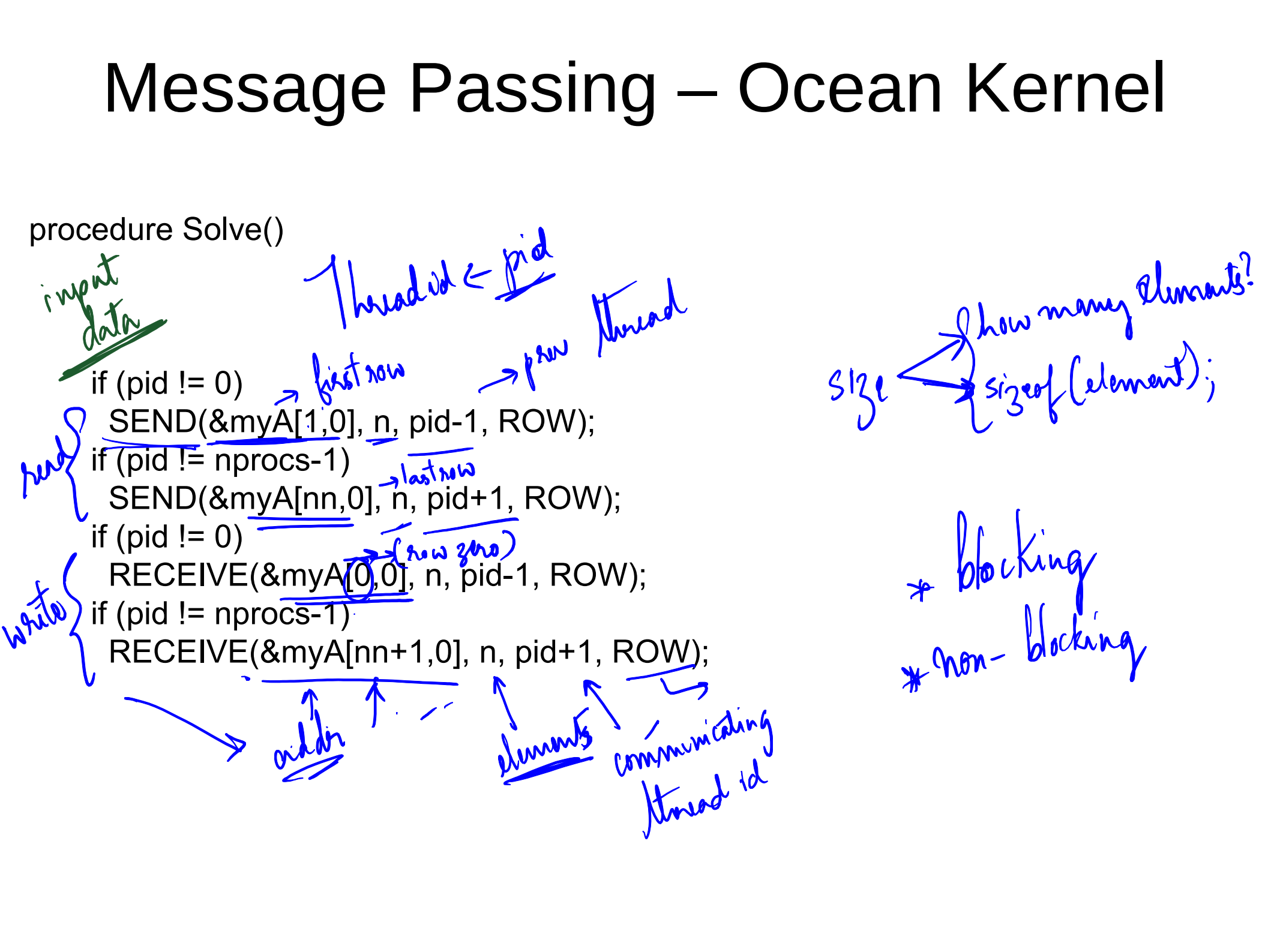


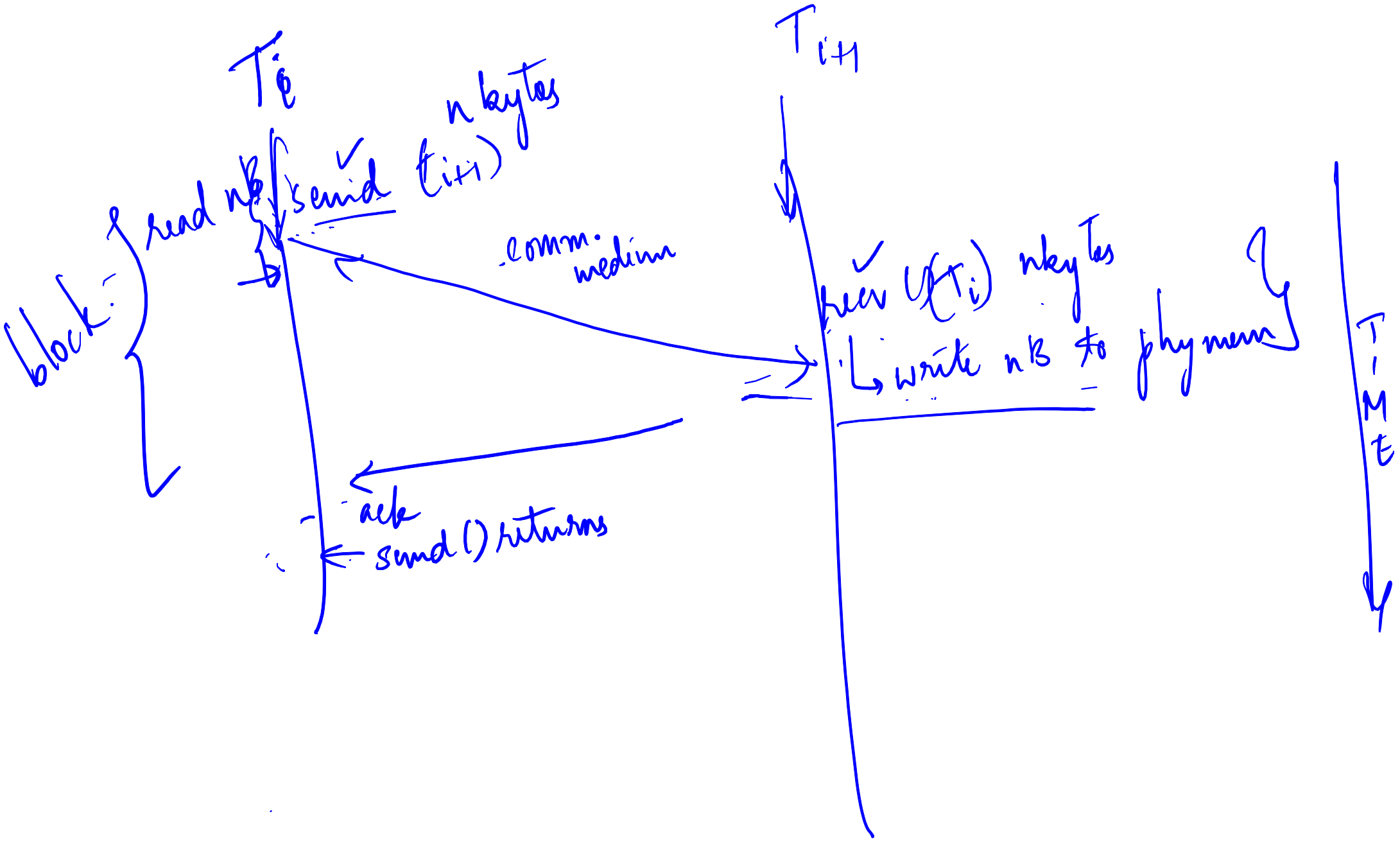
Message Passing – Ocean Kernel



Message Passing – Ocean Kernel

```
procedure Solve()
  if (pid != 0)
    SEND(&myA[1,0], n, pid-1, ROW);
  if (pid != nprocs-1)
    SEND(&myA[nn,0], n, pid+1, ROW);
  if (pid != 0)
    RECEIVE(&myA[0,0], n, pid-1, ROW);
  if (pid != nprocs-1)
    RECEIVE(&myA[nn+1,0], n, pid+1, ROW);
```




T_i
T_{i+1}
n bytes
send (T_{i+1})
read nB
block
comm. medium
recv (T_i) nbytes
↳ write nB to phy mem
ack
send() returns
TIME

Message Passing – Ocean Kernel

Compute mydiff

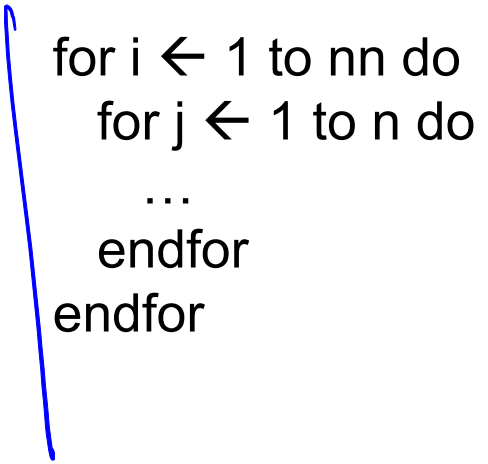
procedure Solve()

```
if (pid != 0)
  SEND(&myA[1,0], n, pid-1, ROW);
if (pid != nprocs-1)
  SEND(&myA[nn,0], n, pid+1, ROW);
if (pid != 0)
  RECEIVE(&myA[0,0], n, pid-1, ROW);
if (pid != nprocs-1)
  RECEIVE(&myA[nn+1,0], n, pid+1, ROW);
```

Message Passing – Ocean Kernel

procedure Solve()

```
if (pid != 0)
  SEND(&myA[1,0], n, pid-1, ROW);
if (pid != nprocs-1)
  SEND(&myA[nn,0], n, pid+1, ROW);
if (pid != 0)
  RECEIVE(&myA[0,0], n, pid-1, ROW);
if (pid != nprocs-1)
  RECEIVE(&myA[nn+1,0], n, pid+1, ROW);
```



Message Passing – Ocean Kernel

```
procedure Solve()
```

```
for i ← 1 to nn do
   for j ← 1 to n do
      …
   endfor
endfor
```

```
if (pid != 0)
  SEND(&myA[1,0], n, pid-1, ROW);
if (pid != nprocs-1)
  SEND(&myA[nn,0], n, pid+1, ROW);
if (pid != 0)
  RECEIVE(&myA[0,0], n, pid-1, ROW);
if (pid != nprocs-1)
  RECEIVE(&myA[nn+1,0], n, pid+1, ROW);
```

Communicate mydiff to a designated master

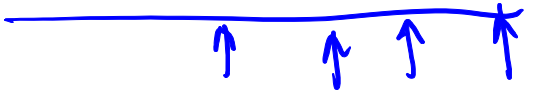
Message Passing – Ocean Kernel

```
procedure Solve()
```

```
if (pid != 0)
  SEND(&myA[1,0], n, pid-1, ROW);
if (pid != nprocs-1)
  SEND(&myA[nn,0], n, pid+1, ROW);
if (pid != 0)
  RECEIVE(&myA[0,0], n, pid-1, ROW);
if (pid != nprocs-1)
  RECEIVE(&myA[nn+1,0], n, pid+1, ROW);
```

```
for i ← 1 to nn do
   for j ← 1 to n do
      …
   endfor
endfor
```

```
SEND(mydiff, 1, 0, DIFF);
```



Message Passing – Ocean Kernel

```
procedure Solve()

    if (pid != 0)
      SEND(&myA[1,0], n, pid-1, ROW);
    if (pid != nprocs-1)
      SEND(&myA[nn,0], n, pid+1, ROW);
    if (pid != 0)
      RECEIVE(&myA[0,0], n, pid-1, ROW);
    if (pid != nprocs-1)
      RECEIVE(&myA[nn+1,0], n, pid+1, ROW);
```

```
for i ← 1 to nn do
   for j ← 1 to n do
      …
   endfor
endfor

  SEND(mydiff, 1, 0, DIFF);
```

Master Thread:
1. accumulate all mydiff
2. Check threshold

Message Passing – Ocean Kernel

```
procedure Solve()

  if (pid != 0)
    SEND(&myA[1,0], n, pid-1, ROW);
  if (pid != nprocs-1)
    SEND(&myA[nn,0], n, pid+1, ROW);
  if (pid != 0)
    RECEIVE(&myA[0,0], n, pid-1, ROW);
  if (pid != nprocs-1)
    RECEIVE(&myA[nn+1,0], n, pid+1, ROW);

  for i ← 1 to nn do
    for j ← 1 to n do
      …
    endfor
  endfor
  if (pid != 0)
    SEND(mydiff, 1, 0, DIFF);
  else
    for i ← 1 to nprocs-1 do
      RECEIVE(tempdiff, 1, *, DIFF);
      mydiff += tempdiff;
    endfor
    if  (mydiff < TOL)  done = 1;

endwhile
```

Message Passing – Ocean Kernel

```
procedure Solve()

  if (pid != 0)
    SEND(&myA[1,0], n, pid-1, ROW);
  if (pid != nprocs-1)
    SEND(&myA[nn,0], n, pid+1, ROW);
  if (pid != 0)
    RECEIVE(&myA[0,0], n, pid-1, ROW);
  if (pid != nprocs-1)
    RECEIVE(&myA[nn+1,0], n, pid+1, ROW);

  for i ← 1 to nn do
    for j ← 1 to n do
      …
    endfor
  endfor
  if (pid != 0)
    SEND(mydiff, 1, 0, DIFF);
  Receive result from master
  else
    for i ← 1 to nprocs-1 do
      RECEIVE(tempdiff, 1, *, DIFF);
      mydiff += tempdiff;
    endfor
    if  (mydiff < TOL)  done = 1;
  Communicate result to all threads

endwhile
```

Message Passing – Ocean Kernel

```
procedure Solve()

  if (pid != 0)
    SEND(&myA[1,0], n, pid-1, ROW);
  if (pid != nprocs-1)
    SEND(&myA[nn,0], n, pid+1, ROW);
  if (pid != 0)
    RECEIVE(&myA[0,0], n, pid-1, ROW);
  if (pid != nprocs-1)
    RECEIVE(&myA[nn+1,0], n, pid+1, ROW);

  for i ← 1 to nn do
     for j ← 1 to n do
        …
     endfor
  endfor
  if (pid != 0)
     SEND(mydiff, 1, 0, DIFF);
     RECEIVE(done, 1, 0, DONE);
  else
     for i ← 1 to nprocs-1 do
        RECEIVE(tempdiff, 1, *, DIFF);
        mydiff += tempdiff;
     endfor
     if  (mydiff < TOL)  done = 1;
     for i ← 1 to nprocs-1  do
        SEND(done, 1, i, DONE);
     endfor
  endif
endwhile
```

Message Passing – Ocean Kernel

```
procedure Solve()
  int i, j, pid, nn = n/nprocs, done=0;
  float temp, tempdiff, mydiff = 0;
  myA ← malloc(…)
  initialize(myA);
  while (!done) do
    mydiff = 0;
    if (pid != 0)
      SEND(&myA[1,0], n, pid-1, ROW);
    if (pid != nprocs-1)
      SEND(&myA[nn,0], n, pid+1, ROW);
    if (pid != 0)
      RECEIVE(&myA[0,0], n, pid-1, ROW);
    if (pid != nprocs-1)
      RECEIVE(&myA[nn+1,0], n, pid+1, ROW);
    for i ← 1 to nn do
      for j ← 1 to n do
        …
      endfor
    endfor
    if (pid != 0)
      SEND(mydiff, 1, 0, DIFF);
      RECEIVE(done, 1, 0, DONE);
    else
      for i ← 1 to nprocs-1 do
        RECEIVE(tempdiff, 1, *, DIFF);
        mydiff += tempdiff;
      endfor
      if  (mydiff < TOL)  done = 1;
      for i ← 1 to nprocs-1  do
        SEND(done, 1, i, DONE);
      endfor
    endif
  endwhile
```