Introduction to Parallel Computing with OpenMP

PRESENTED BY

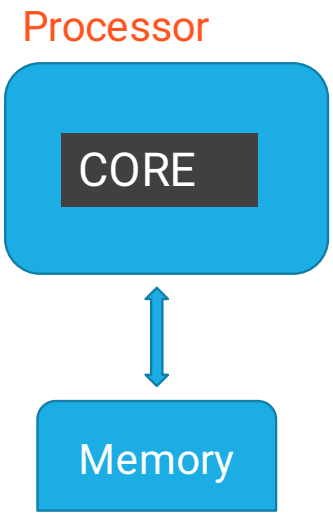
- KAPIL SAWATE
- TEJESH CHAURAGADE
- HPC TECH CDAC(PUNE)

Contents

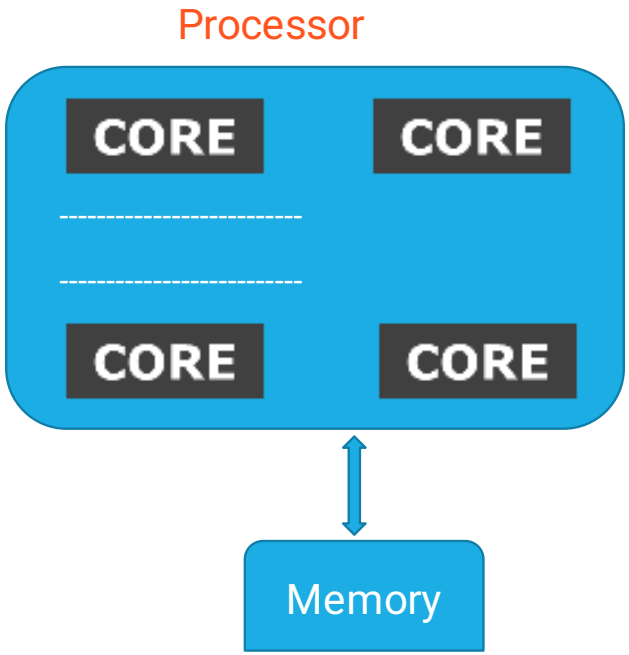
Introduction

- OpenMP is one of the most common parallel programming models in use today.
- It is relatively easy to use which makes a great language to start with when learning to write parallel software.

Basic Architecture



Older processor had only single CPU core to execute instructions



Processors have 4 or more independent CPU cores to execute instructions

Sequential Program Execution

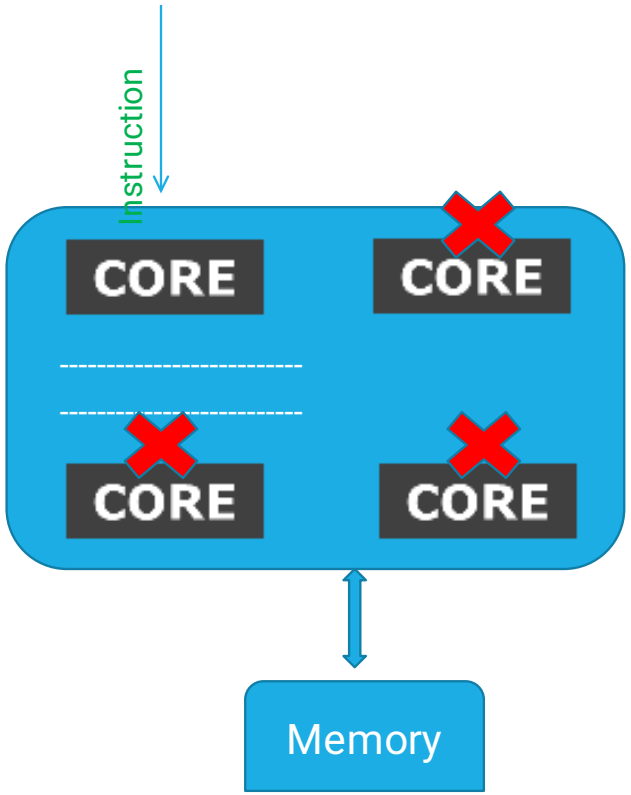
When you run sequential program

Instructions executed in serial

Other cores are idle

Waste of available resource…
We want all cores to be used to execute program.

HOW ?



OpenMP : An API for writing Multithreaded Application

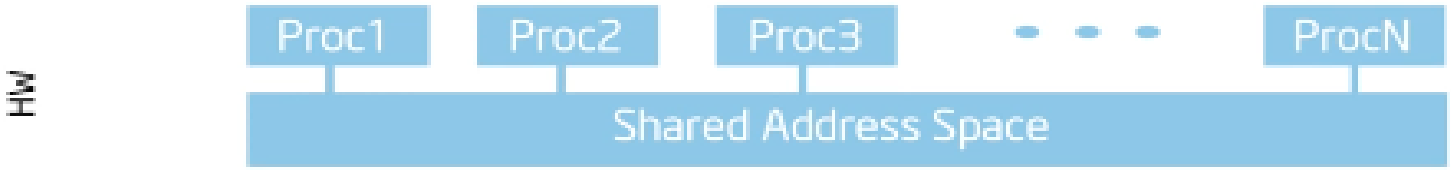
- Set of compiler directives and library routines for parallel application programmers .
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++ .
- Standardise last 20 years of SMP practice.

OpenMP Follows..

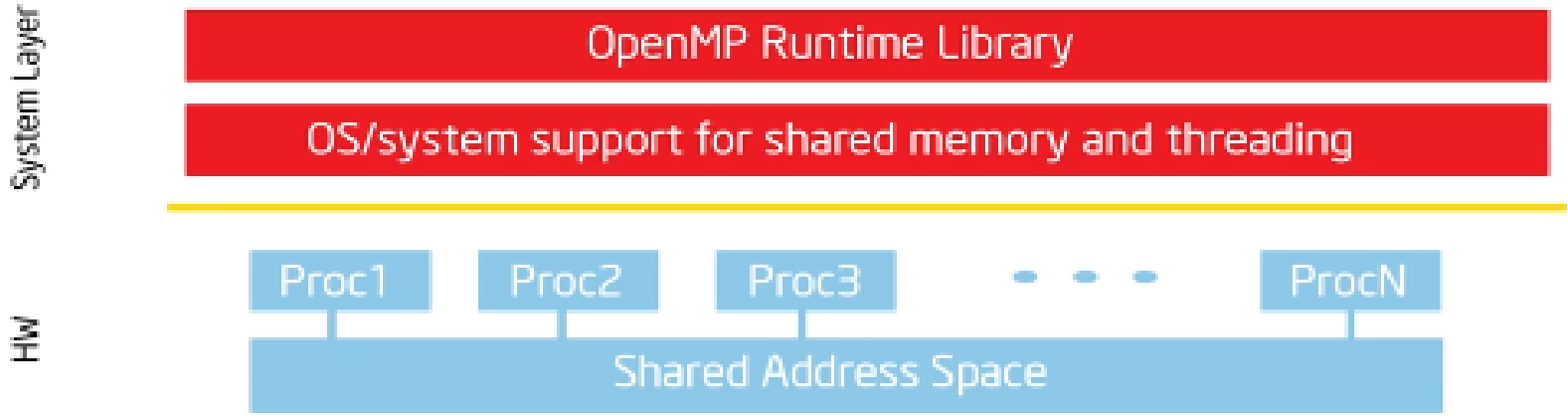
Shared Memory

Explicit Parallelism

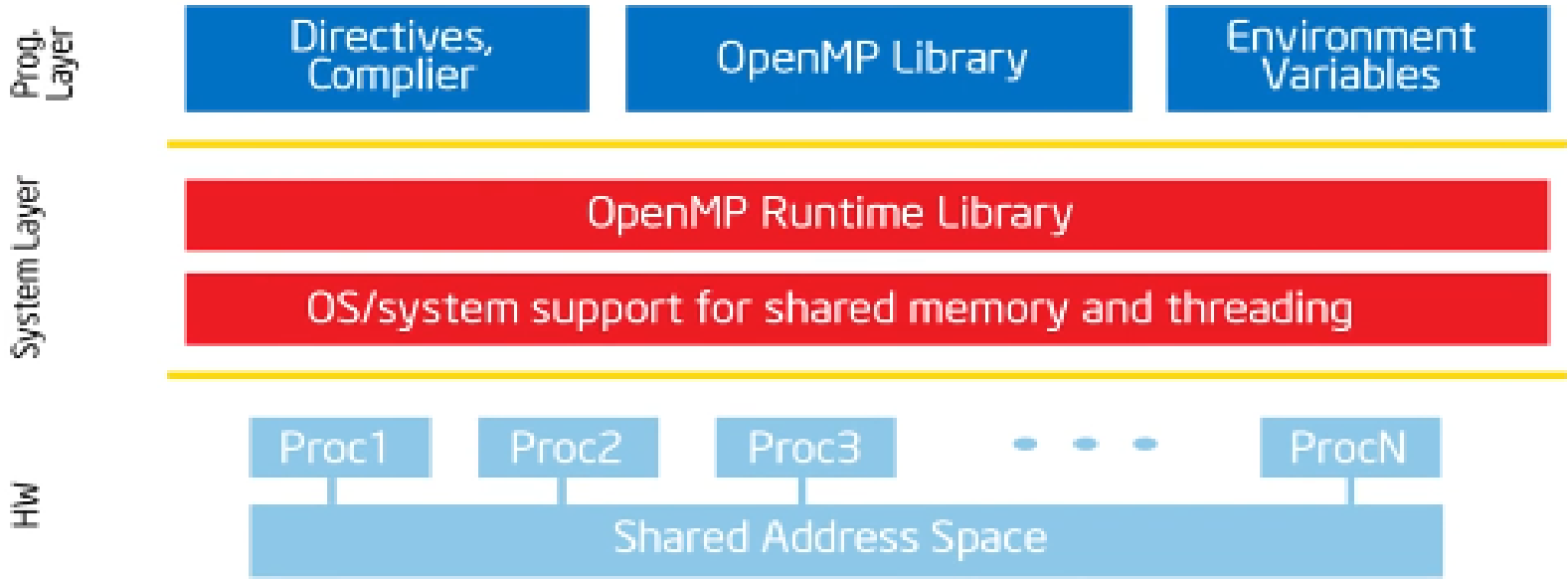
OpenMP Basic Defs: Solution Stack



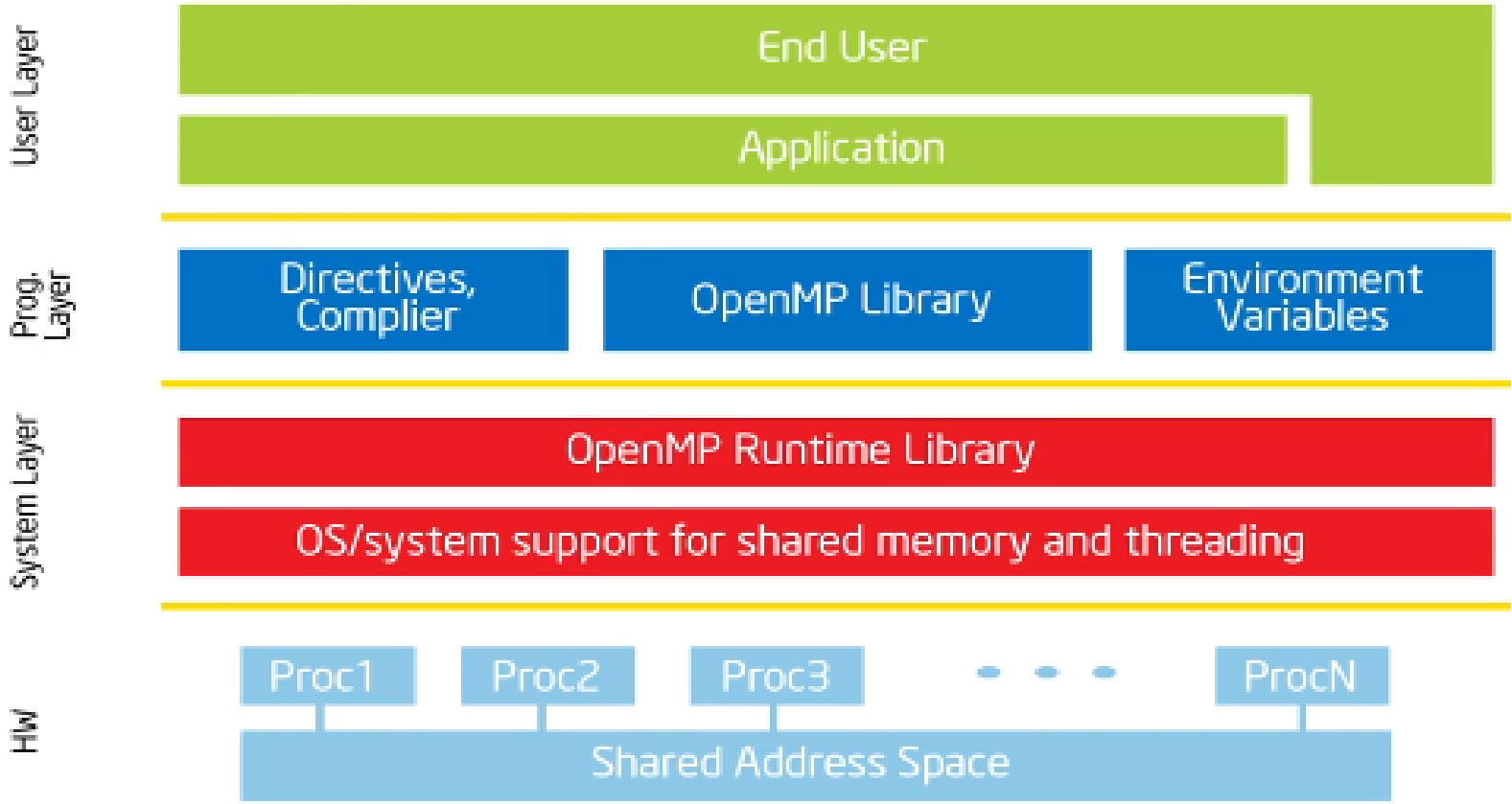
OpenMP Basic Defs: Solution Stack



OpenMP Basic Defs: Solution Stack



OpenMP Basic Defs: Solution Stack



Compiler Notes

- Linux and MAC with gcc:
  - gcc - fopenmp   <program name> -o <executable>
  - export OMP_NUM_THREADS=4
  - ./<executable-name>
- Linux and MAC with PGI:
  - pgcc – mp <program name> – o <executable>
  - export OMP_NUM_THREADS=4
  - ./<executable-name>
- Linux with Intel Compiler:
  - icc - qopenmp <program name> -o output_file <executable>
  - export OMP_NUM_THREADS=4
  - ./<executable-name>

Why choose OpenMP ?

- Portable
  - standardized for shared memory architectures
- Simple and Quick
  - Relatively easy to do parallelization for small parts of an application at a time
  - incremental parallelization
  - supports both fine grained and coarse grained parallelism
- Compact API
  - simple and limited set of directives

When to use OpenMP ?

- Target platform is multicore or multiprocessor.
- Application is cross-platform
- Parallelizable loop
- Last-minute optimization

OpenMP Core Syntax

- Most of the constructs in OpenMP are compiler directives.
  #pragma omp construct [clause [clause]…]
  - Ex. #pragma omp parallel num_threads(4)
- Function prototypes and types in the file: #include < omp.h >
  - Most OpenMP constructs apply to a “structured block”.
- Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.

Serial Program

```c
#include <stdio.h>
int main ()
{
printf("Hello World \n");
}
```

Parallel Program

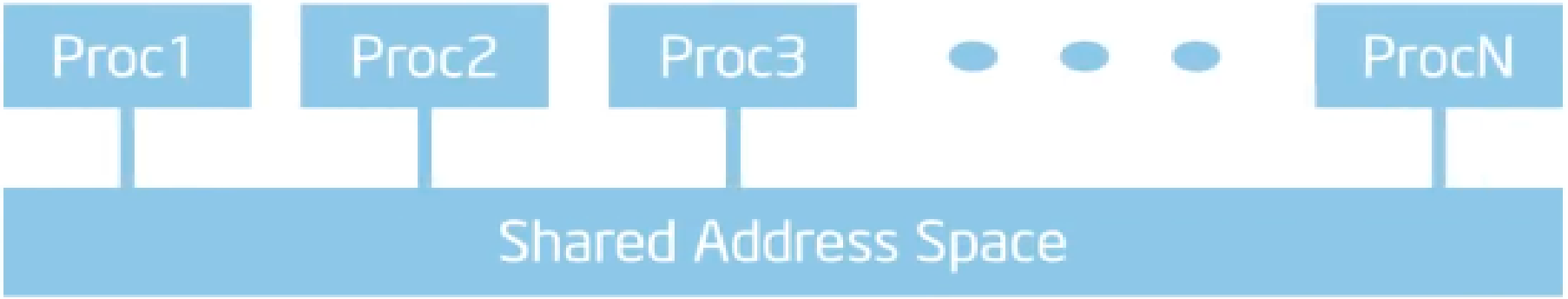
```c
#include<stdio.h>
#include<omp.h>
int main()
{
    #pragma omp parallel
    {
    int ID = omp_get_thread_num();
    printf("Hello(%d)\n",ID);
    printf("World(%d)\n",ID);
    }
    return 0;
}
```

Shared Memory

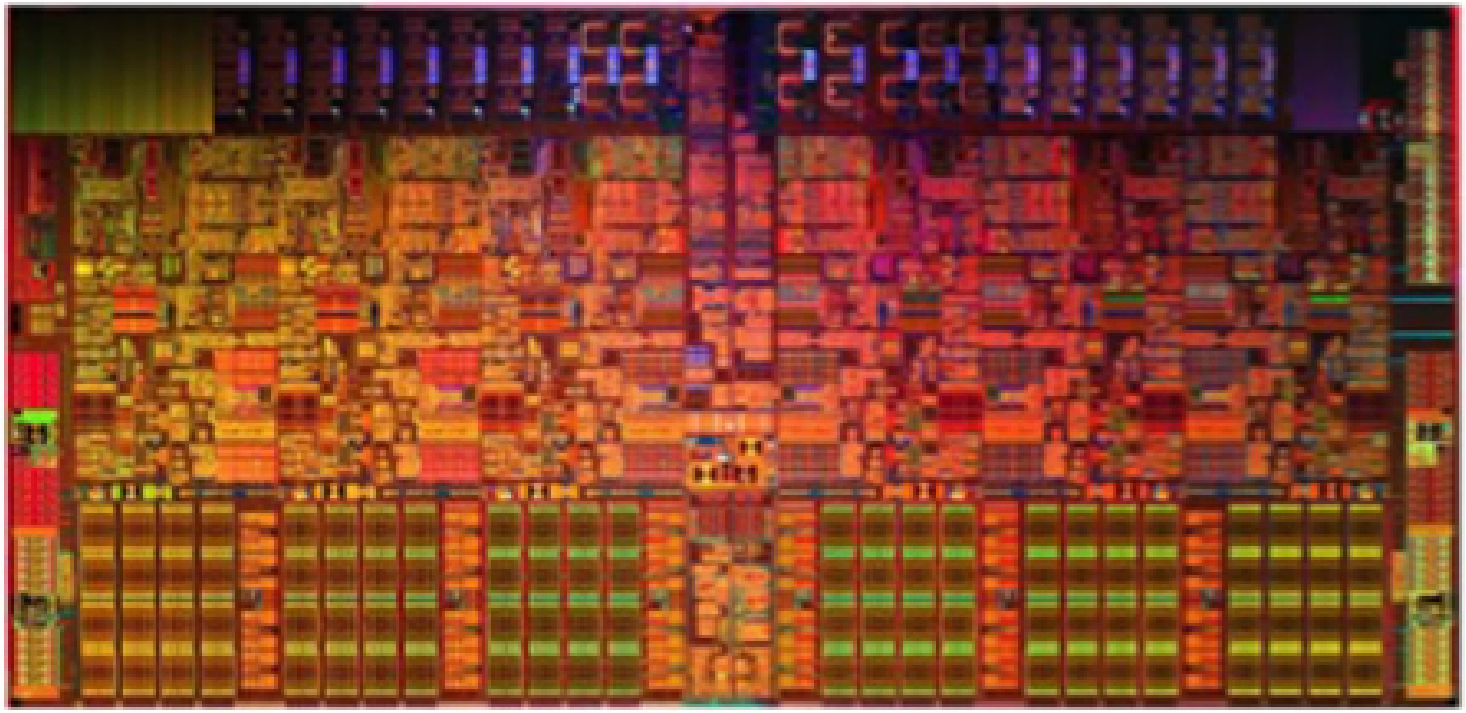
Shared Memory Machines

Shared Address Space

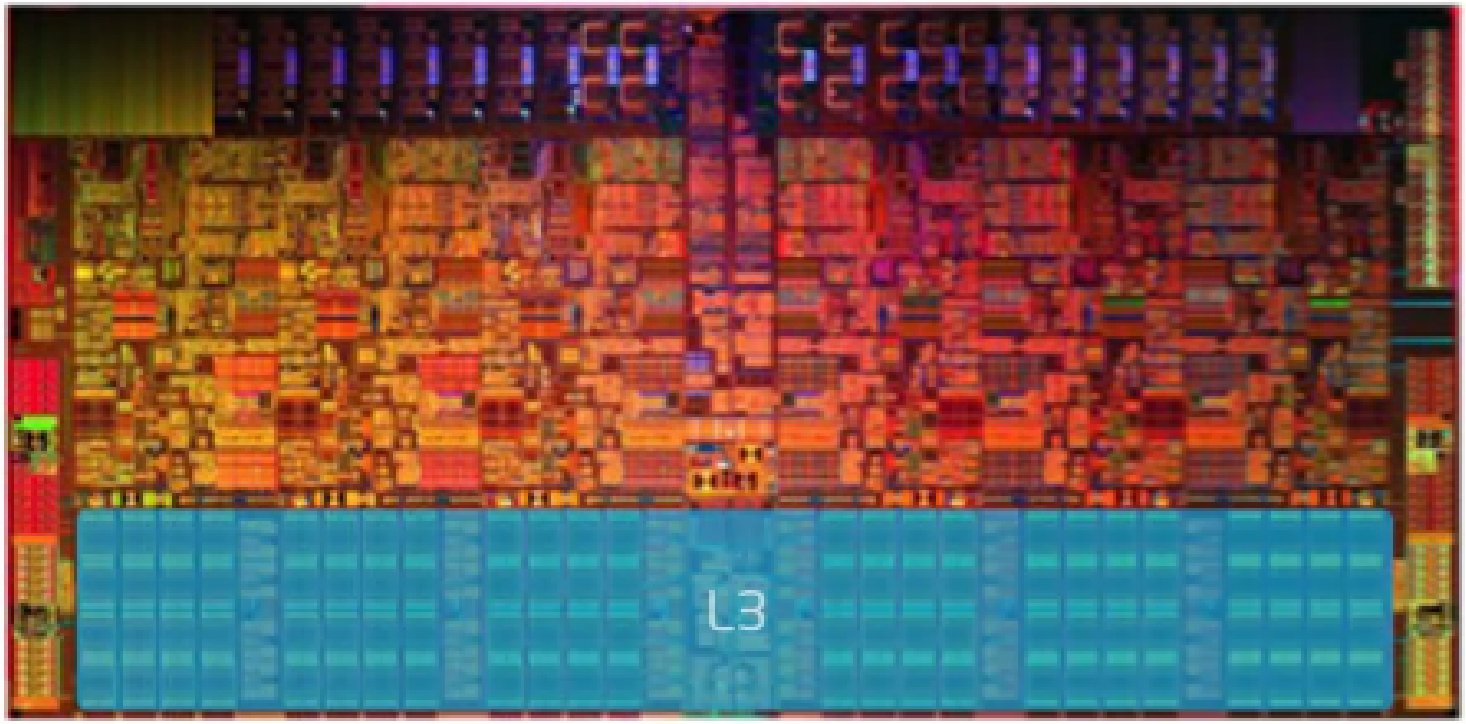
Shared Memory Machines



Intel® Core™ i7-970 Processor

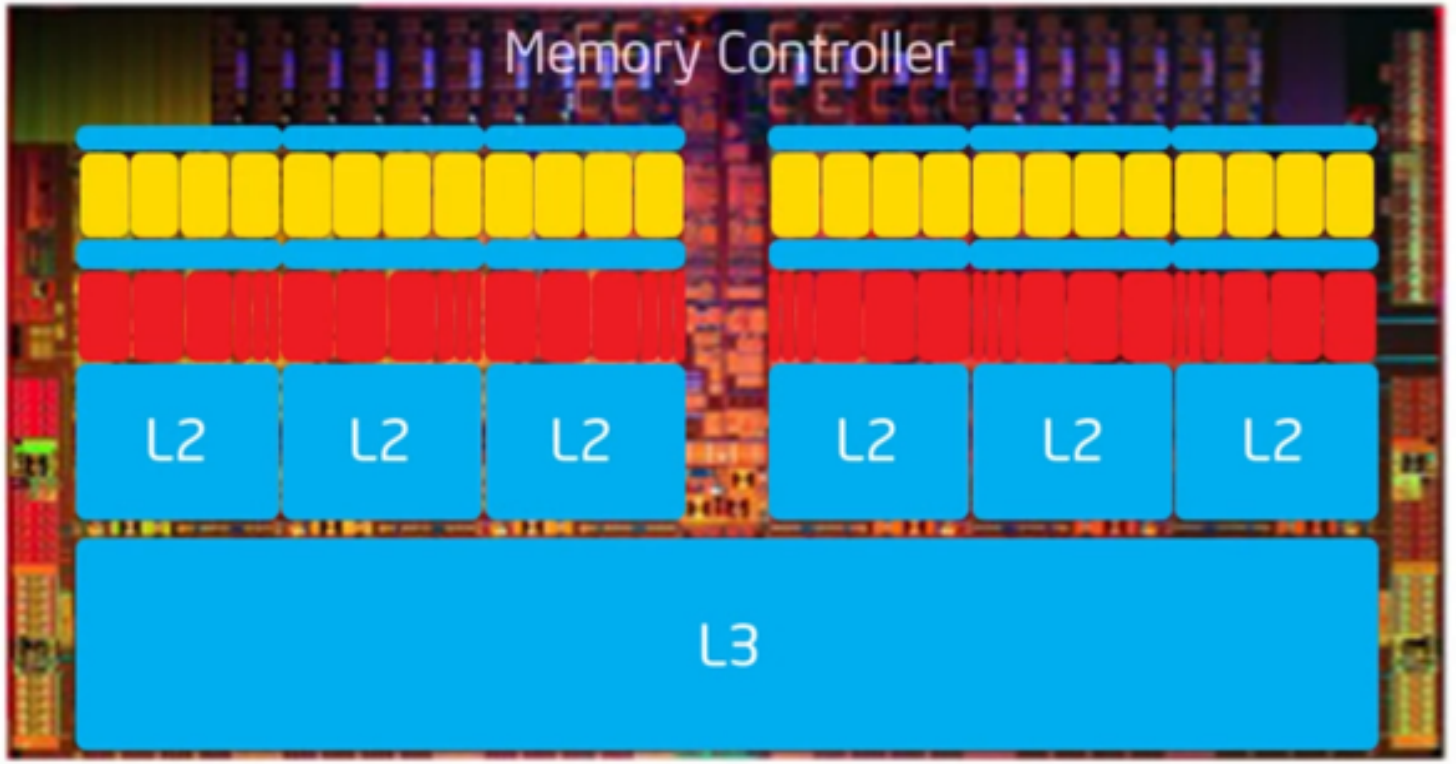


Intel® Core™ i7-970 Processor

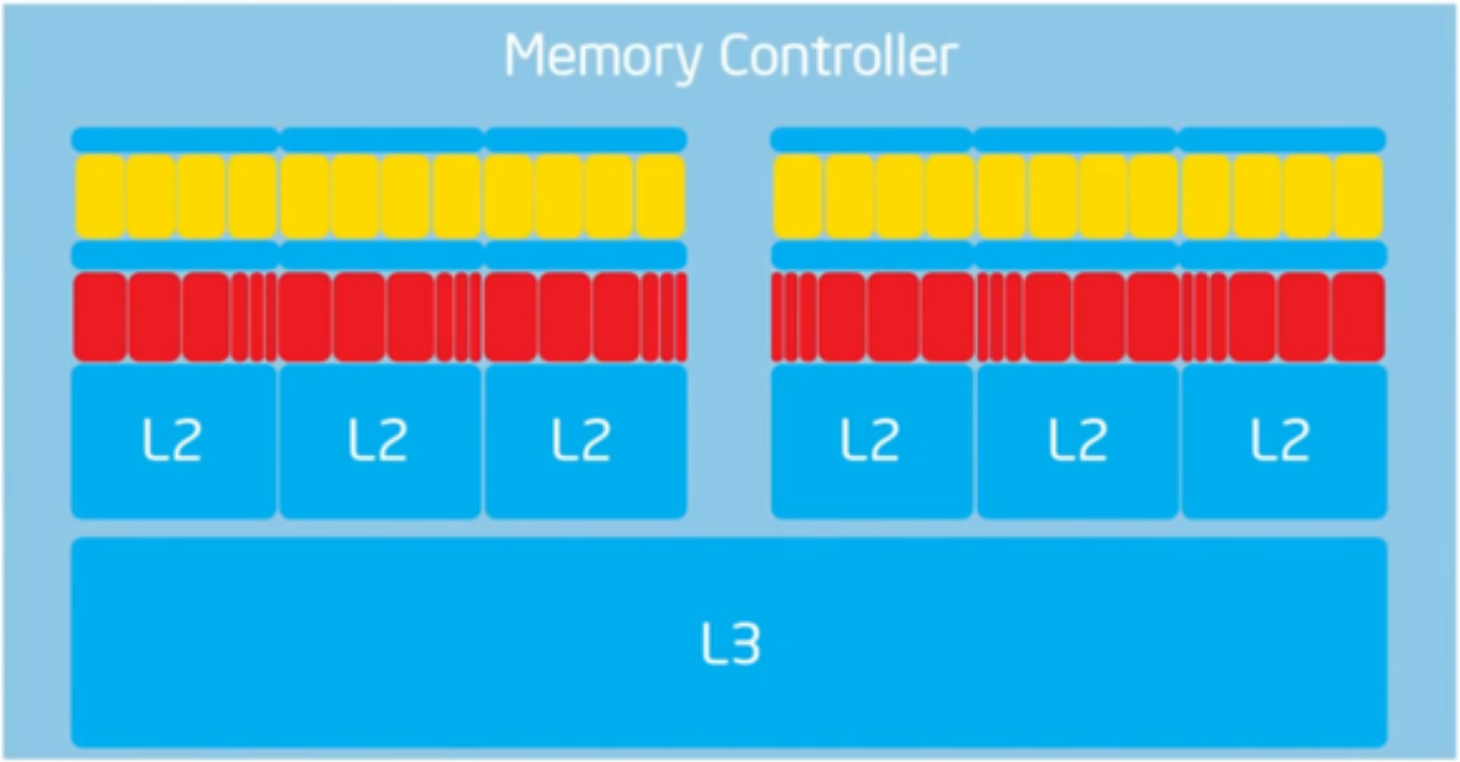


Intel® Core™ i7-970 Processor

Intel® Core™ i7-970 Processor

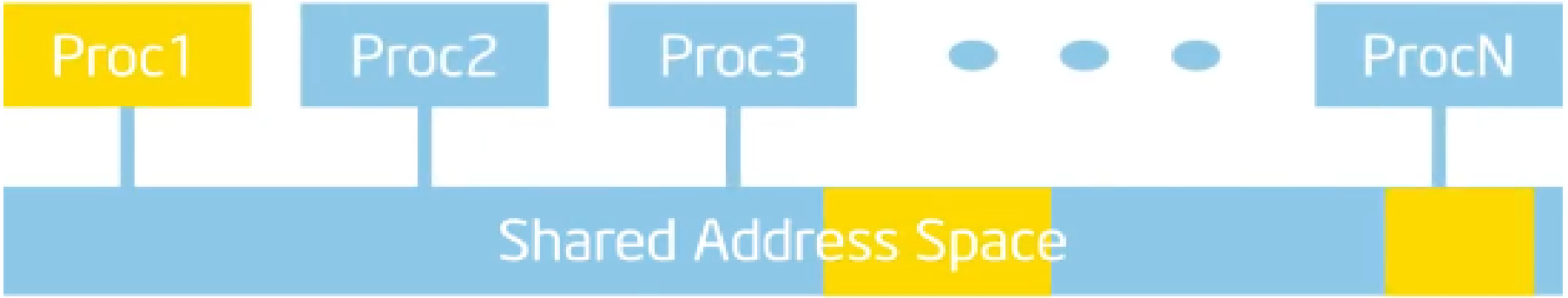


Intel® Core™ i7-970 Processor

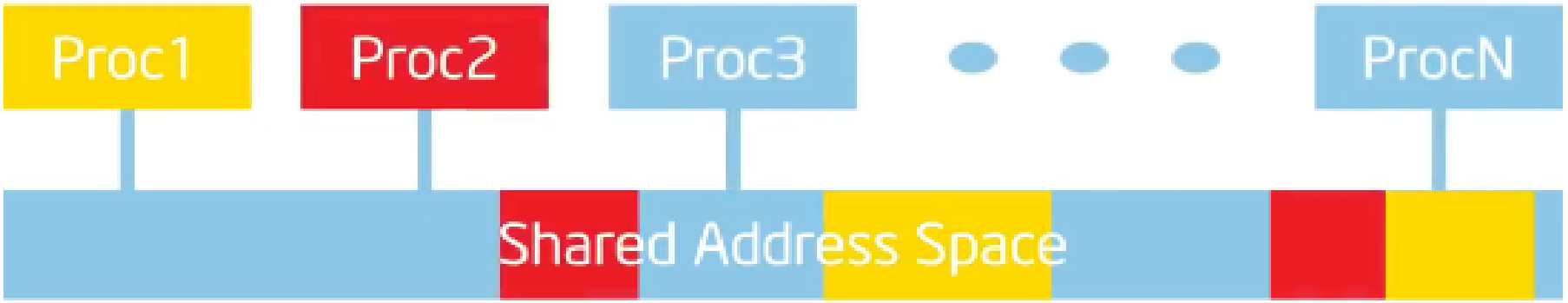


Shared Memory Machines

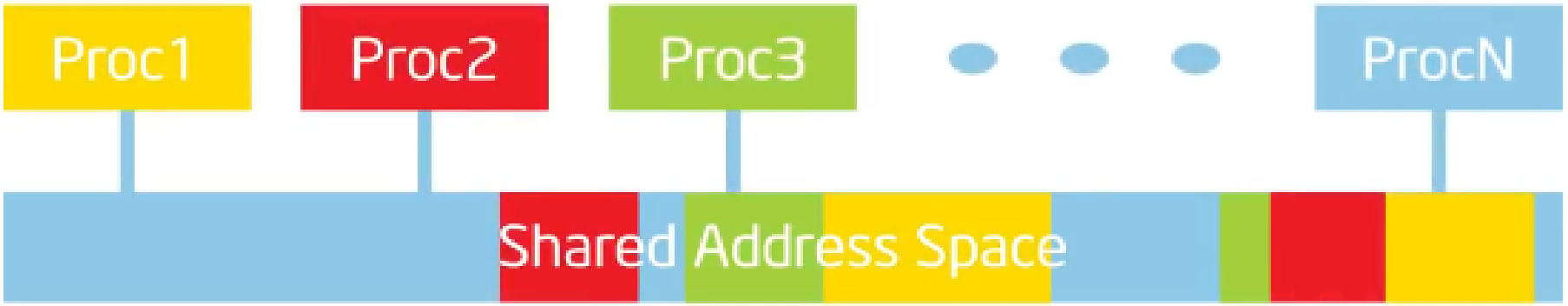
Shared Memory Machines



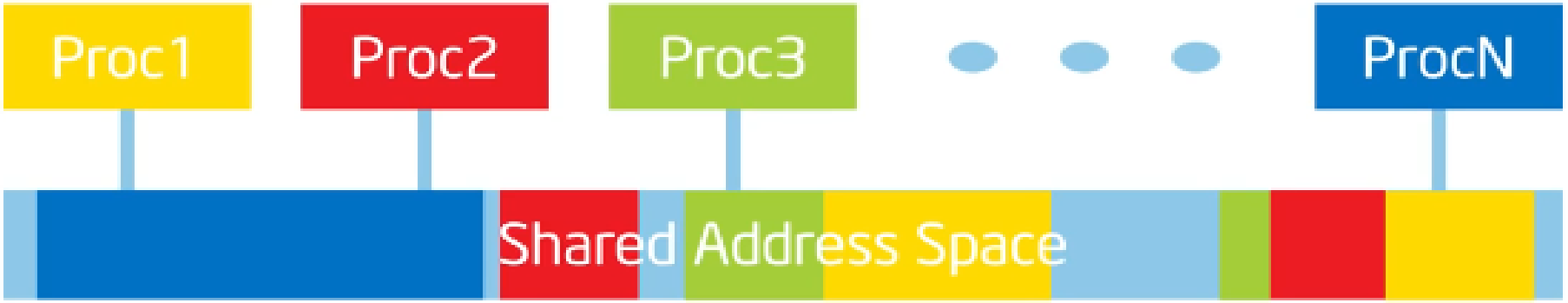
Shared Memory Machines



Shared Memory Machines



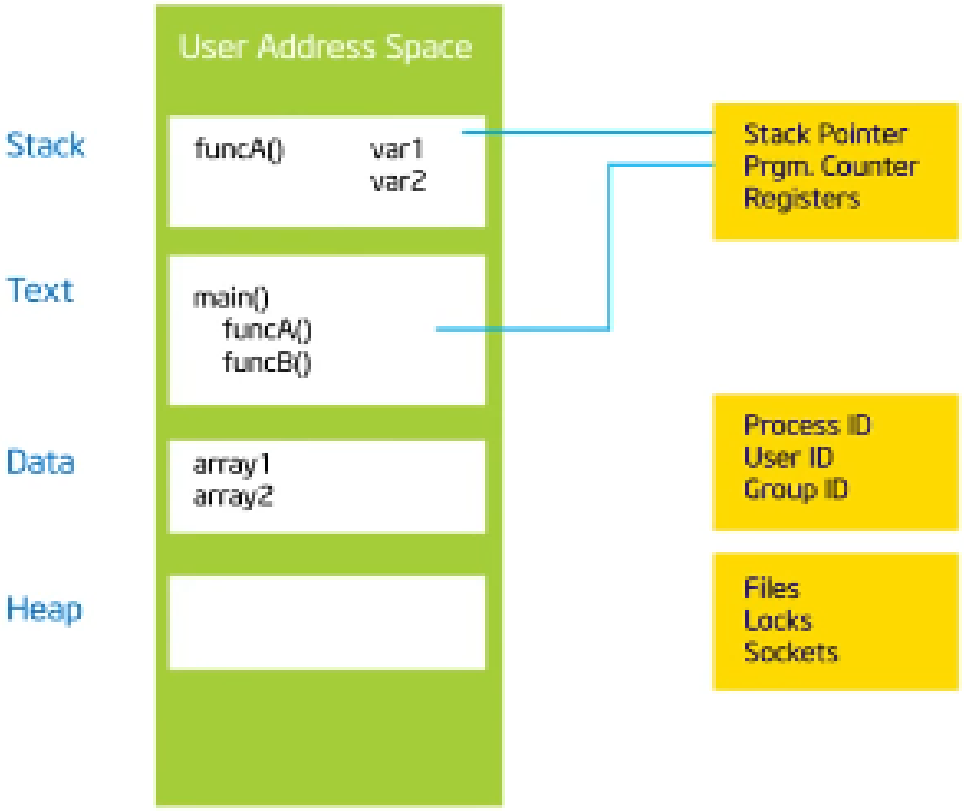
Shared Memory Machines



How do threads interact?

OpenMP is a multi- threading, shared address model.
– Threads communicate by sharing variables.

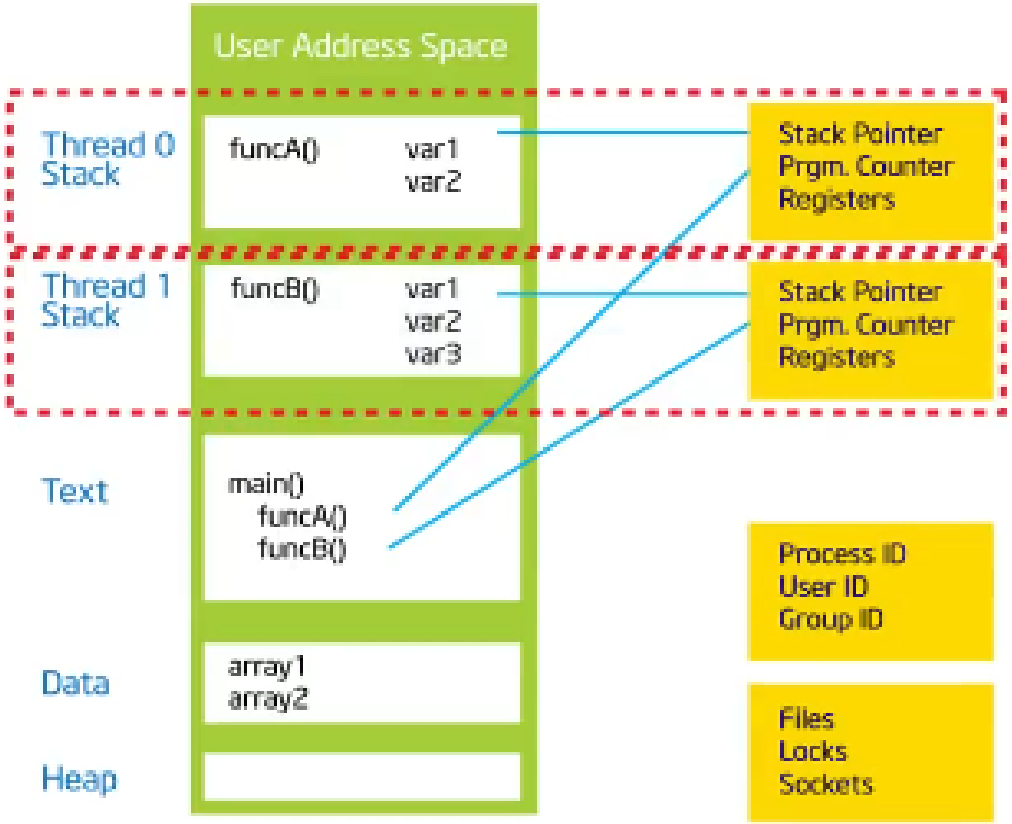
Programming Shared Memory Computers



Process:

- An instance of a program execution.
- The execution context of a running program... i.e. the resources associated with a program's execution.

Programming Shared Memory Computers



Threads:

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

Programming Shared Memory Computers

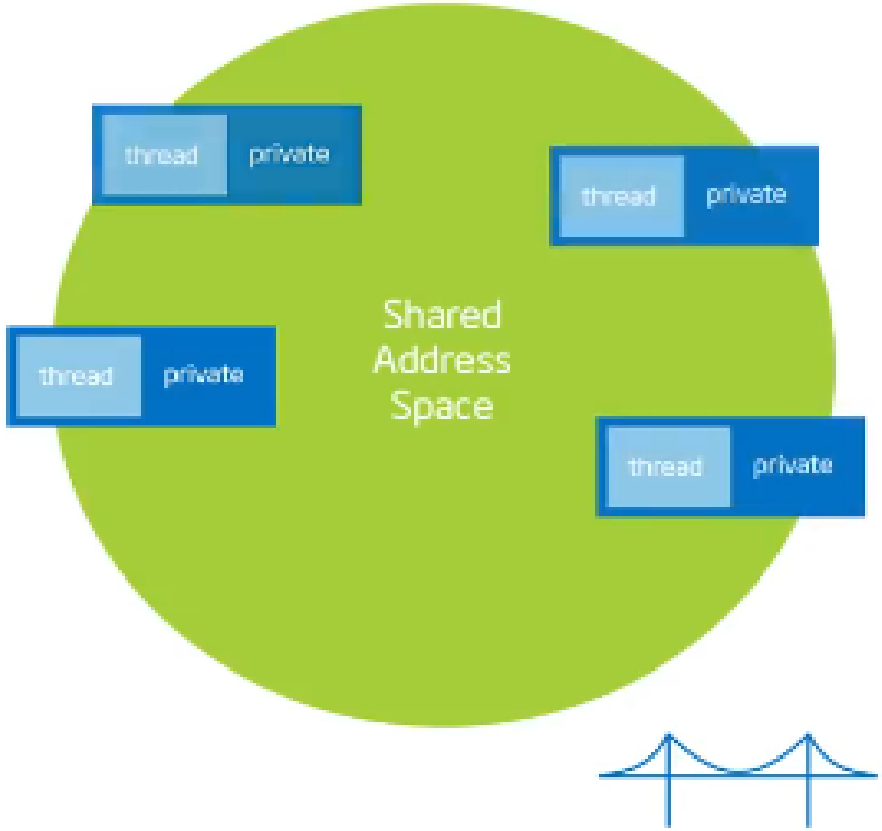
An instance of a program:

One Process and lots of 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



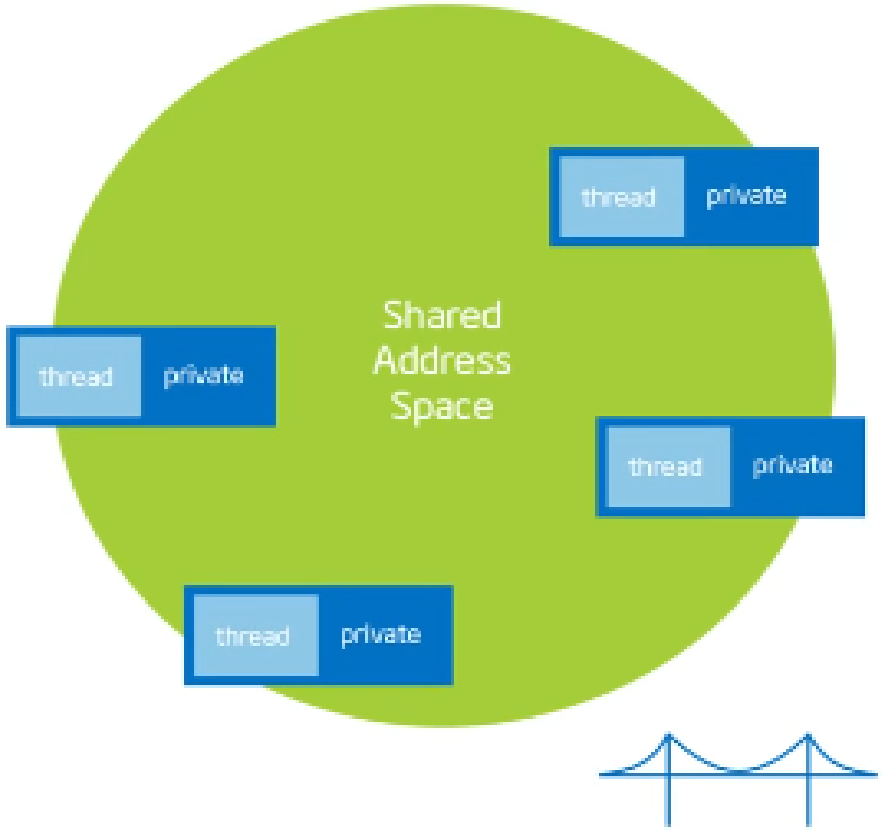
Programming Shared Memory Computers

An instance of a program:
One Process and lots of 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



Programming Shared Memory Computers

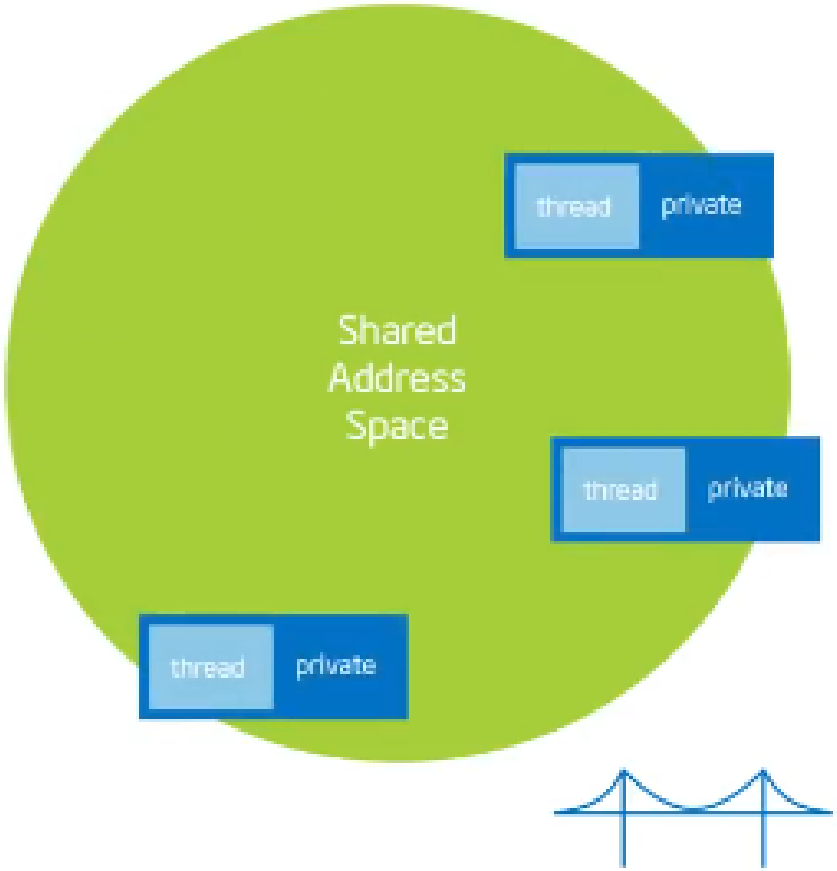
An instance of a program:

One Process and lots of 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

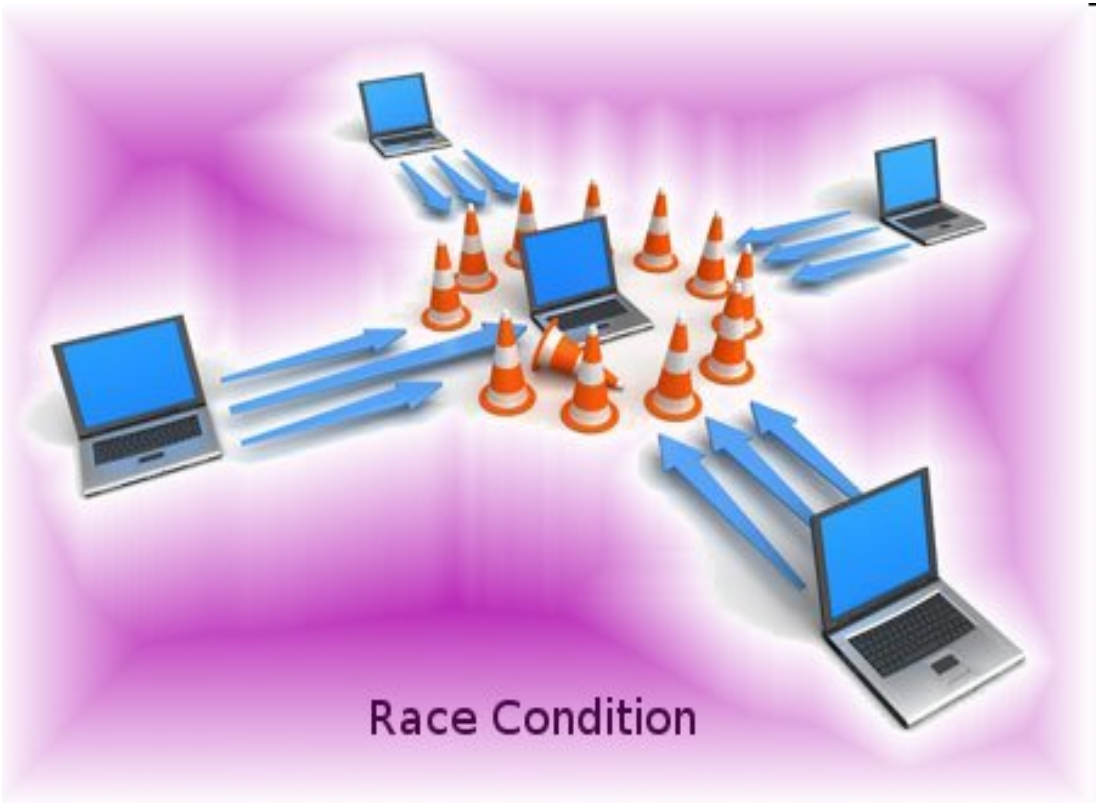


```c
#include<stdio.h>
#include<omp.h>
int main()
{
    #pragma omp parallel
    {
    int ID = omp_get_thread_num();
    printf("Hello(%d)\n",ID);
    printf("World(%d)\n",ID);
    }
    return 0;
}
```

```
$ gcc -fopenmp hello.c -o hello
$ export OMP_NUM_THREADS=4
$ ./hello
```

```
Hello(0)
World(0)
Hello(1)
World(1)
Hello(2)
World(2)
Hello(3)
World(3)
```

Race Condition



Unintended sharing of data causes race conditions:

- ✓ Race condition:

  When the program's outcome changes as the threads are scheduled differently.

- ✓ To control race conditions:

  Use synchronization to protect data conflicts.

- ✓ Synchronization is expensive:

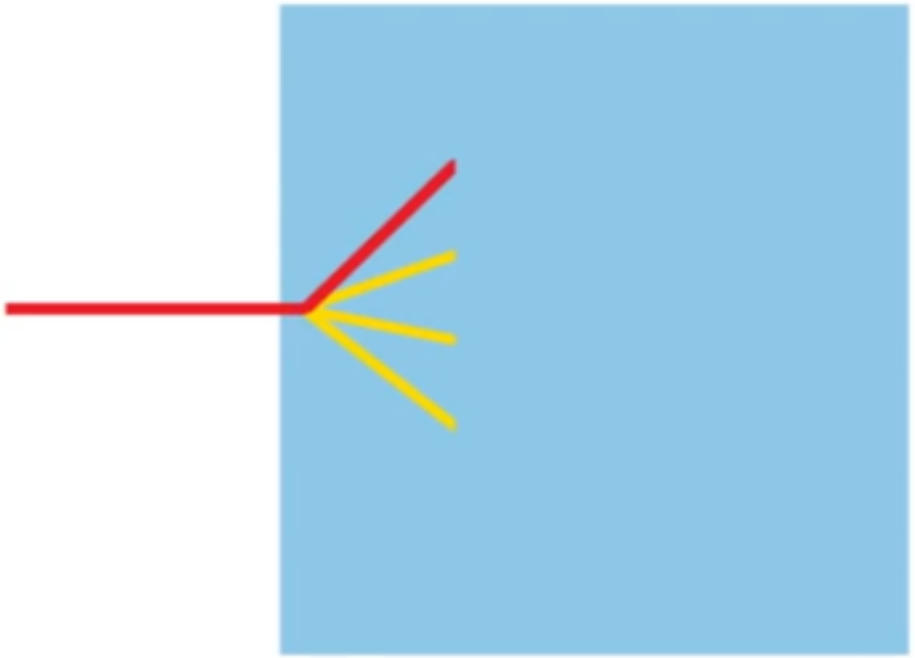
  Change how data is accessed to minimize the need for synchronization.

Fork-Join Parallelism

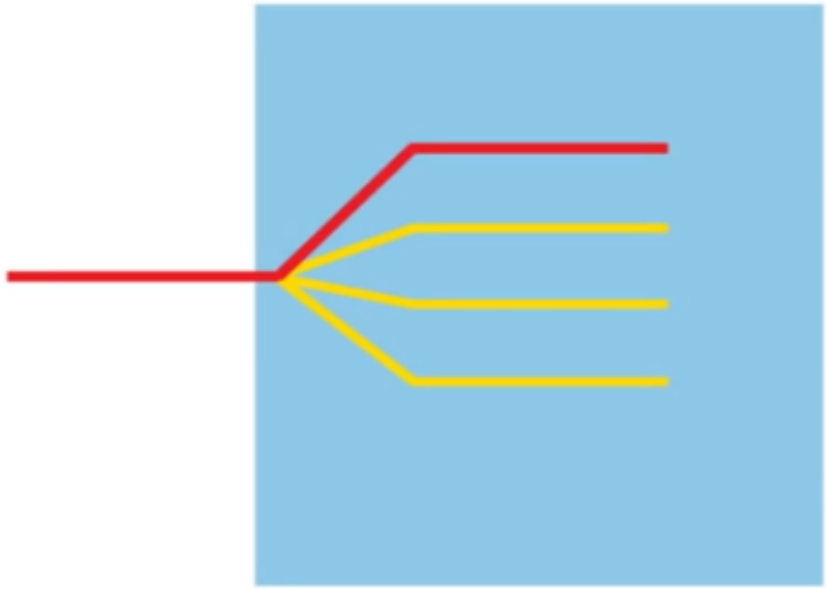
Fork-Join Parallelism

Fork-Join Parallelism

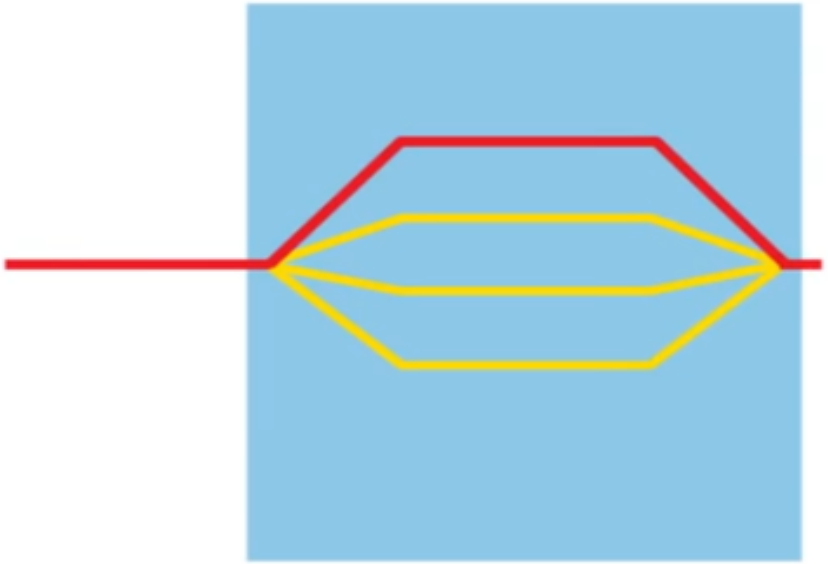
Fork-Join Parallelism

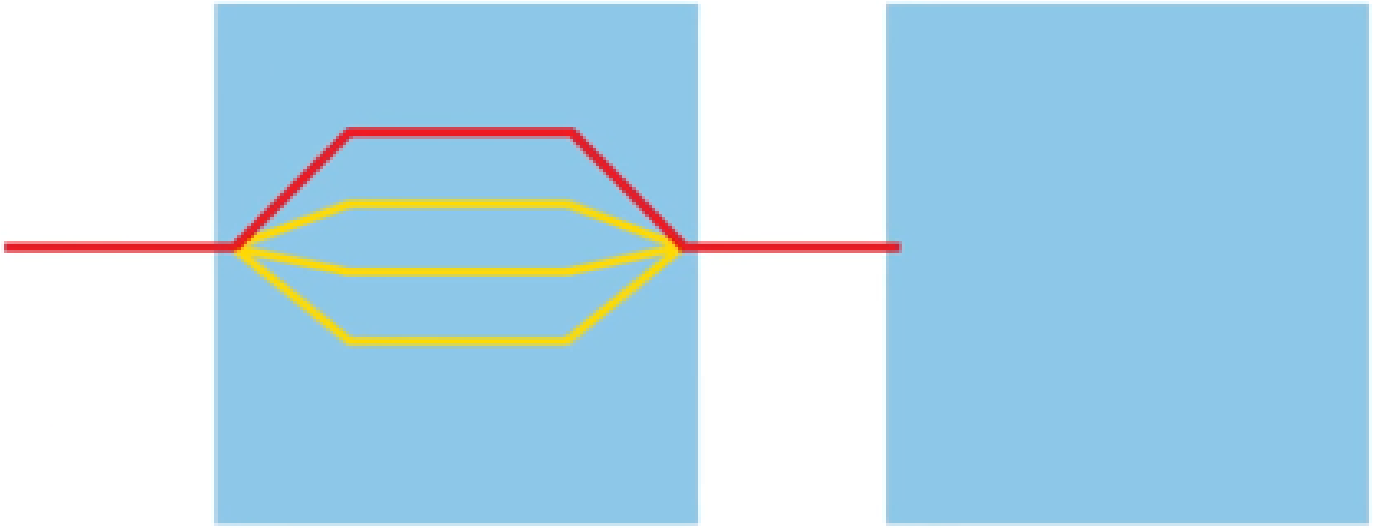


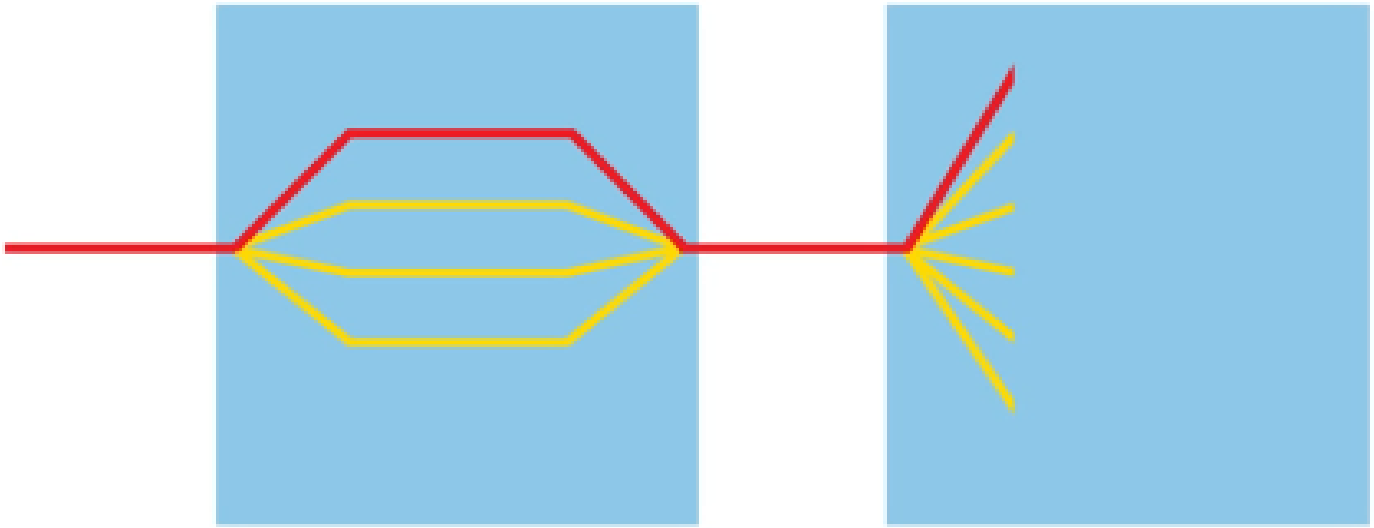
Fork-Join Parallelism

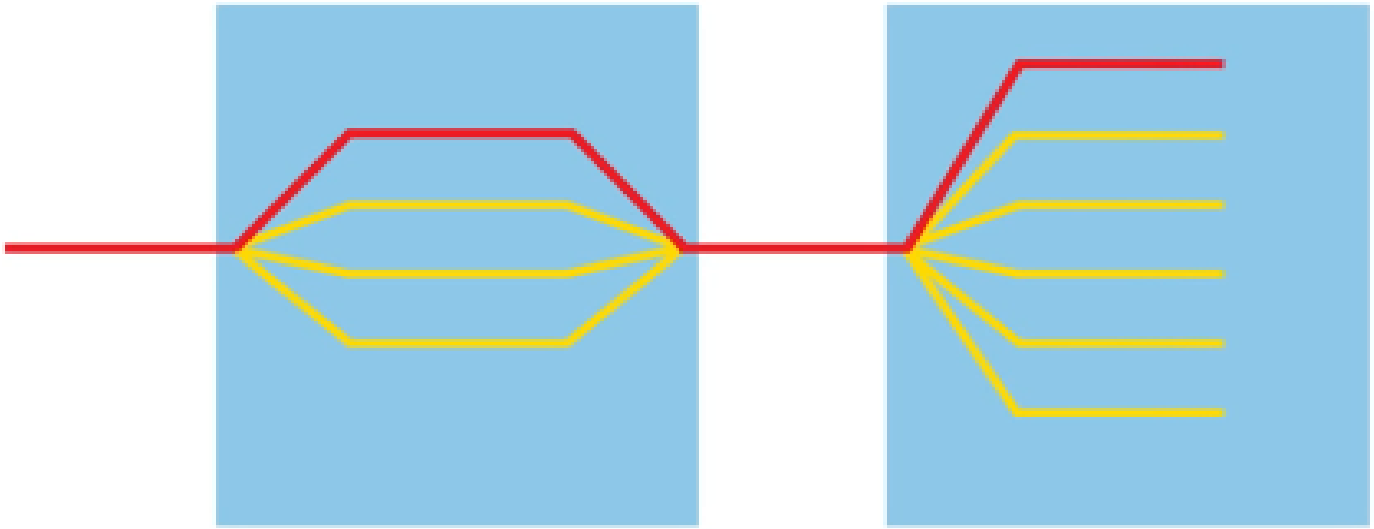


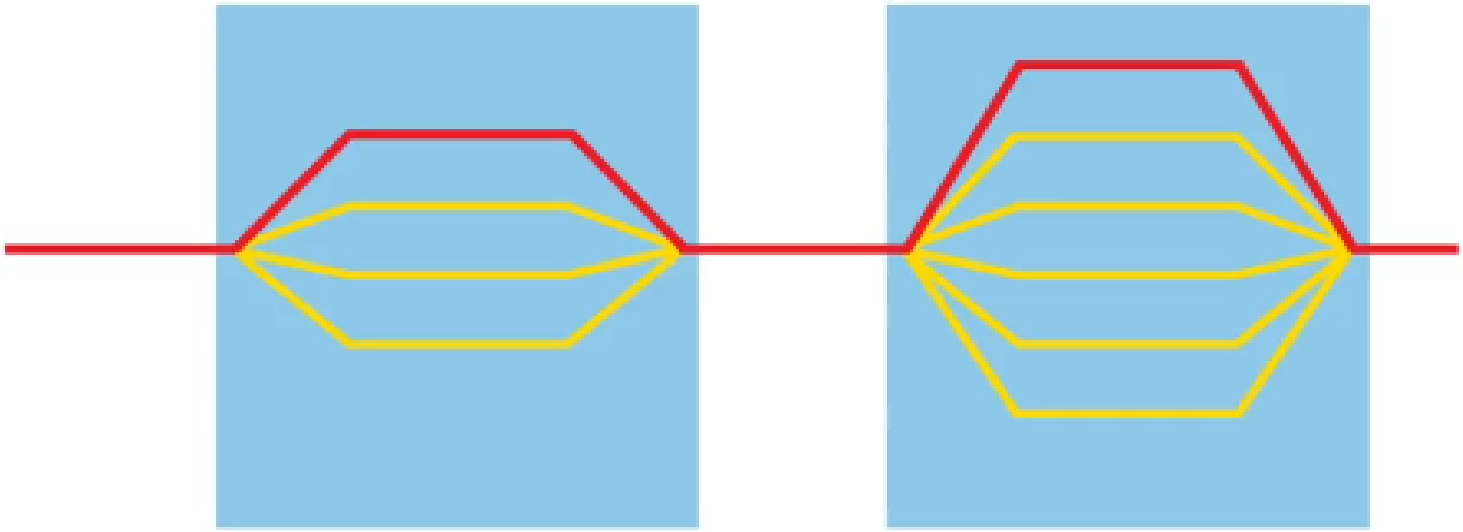
Fork-Join Parallelism

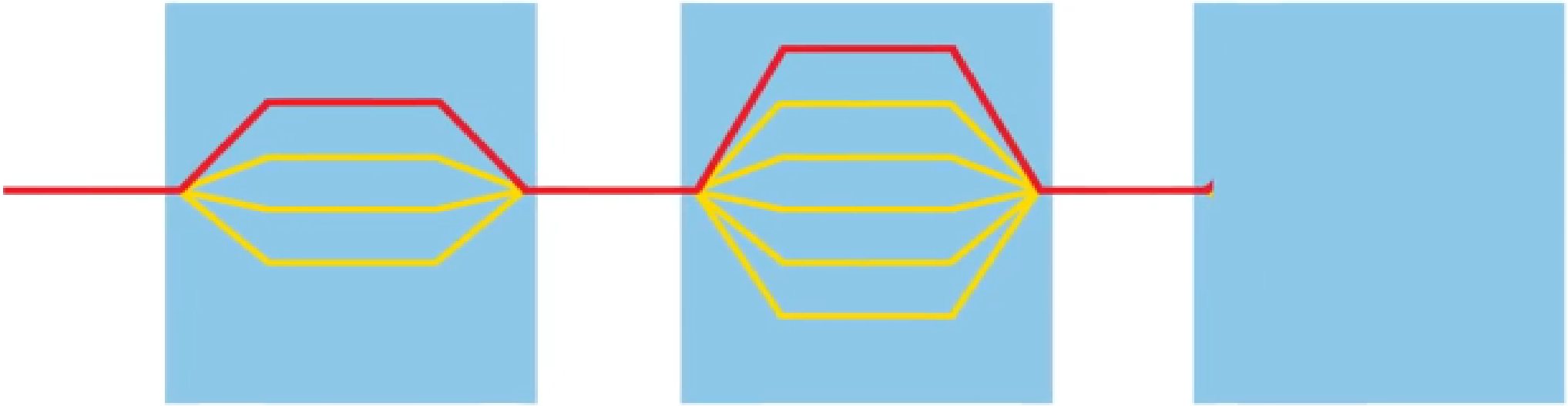


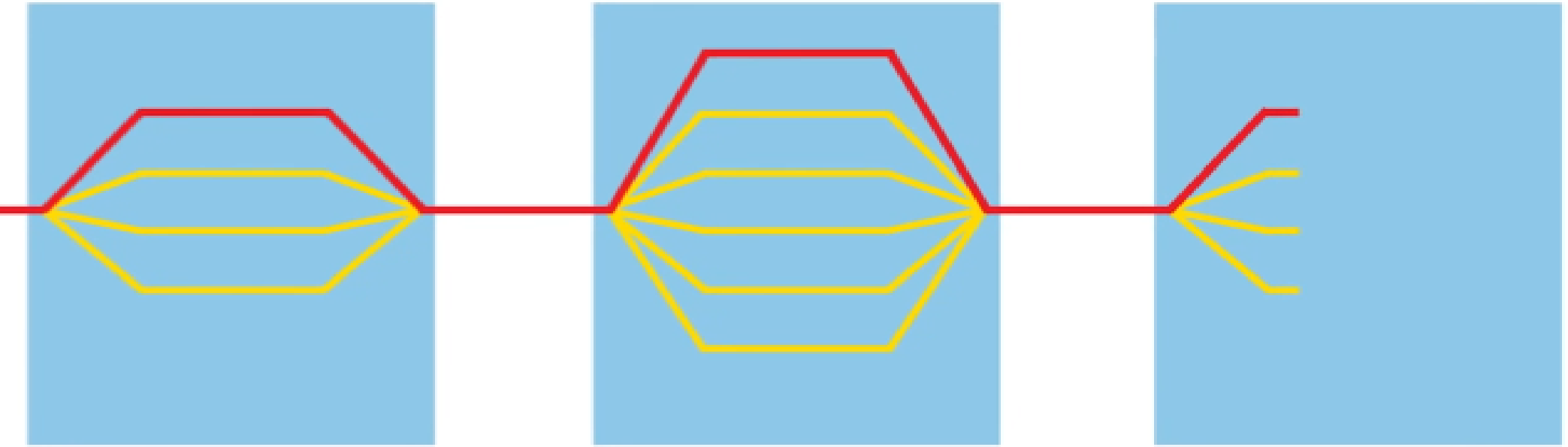


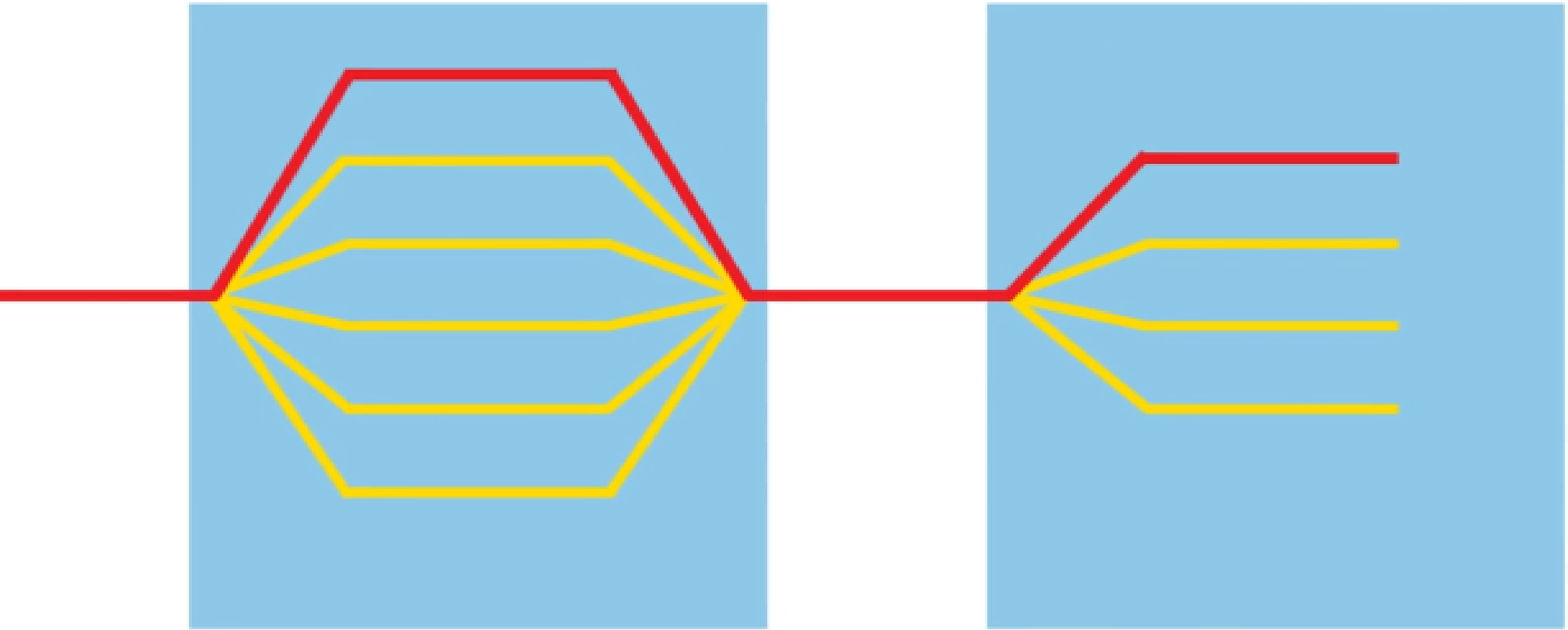


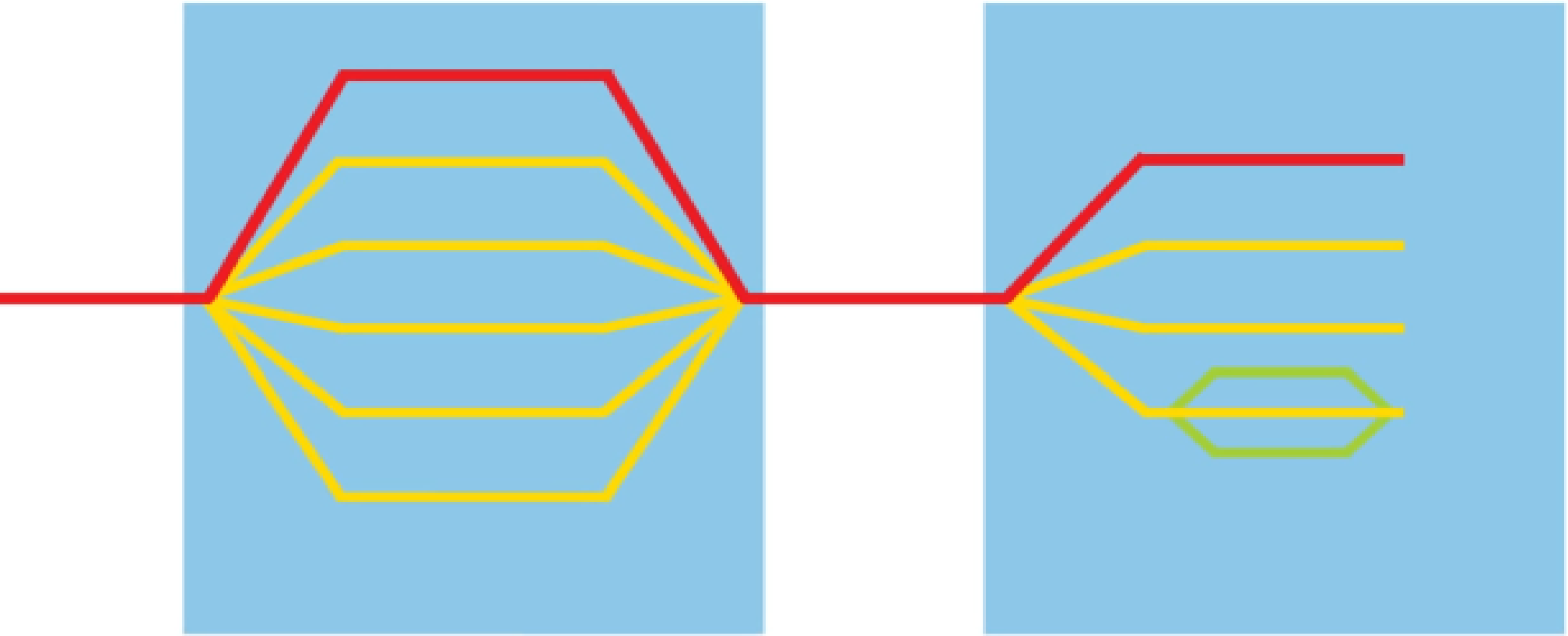


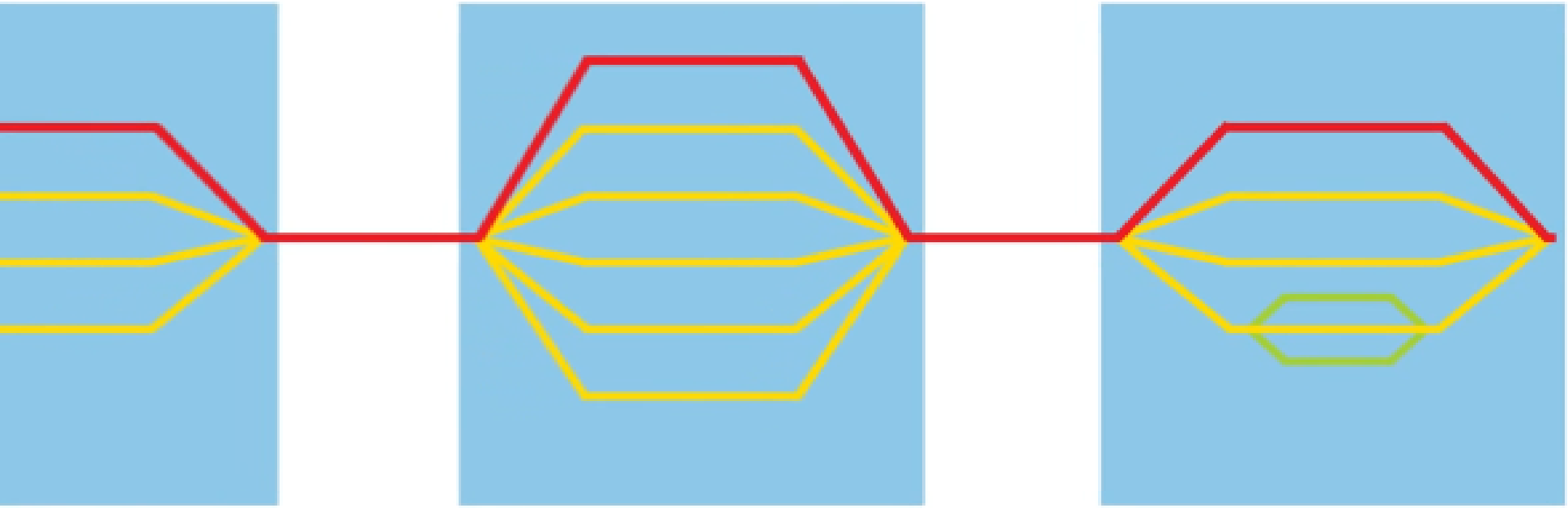


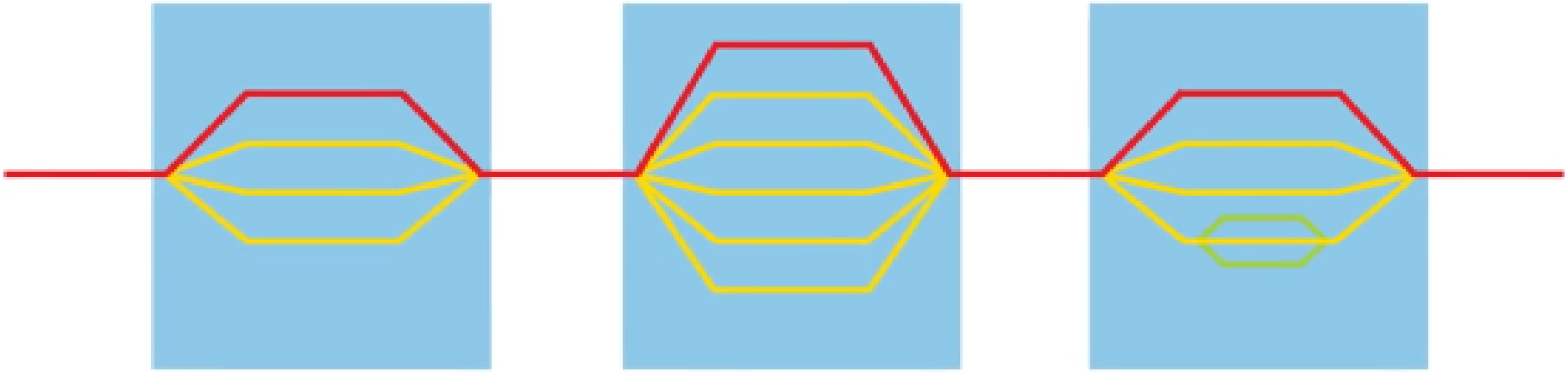






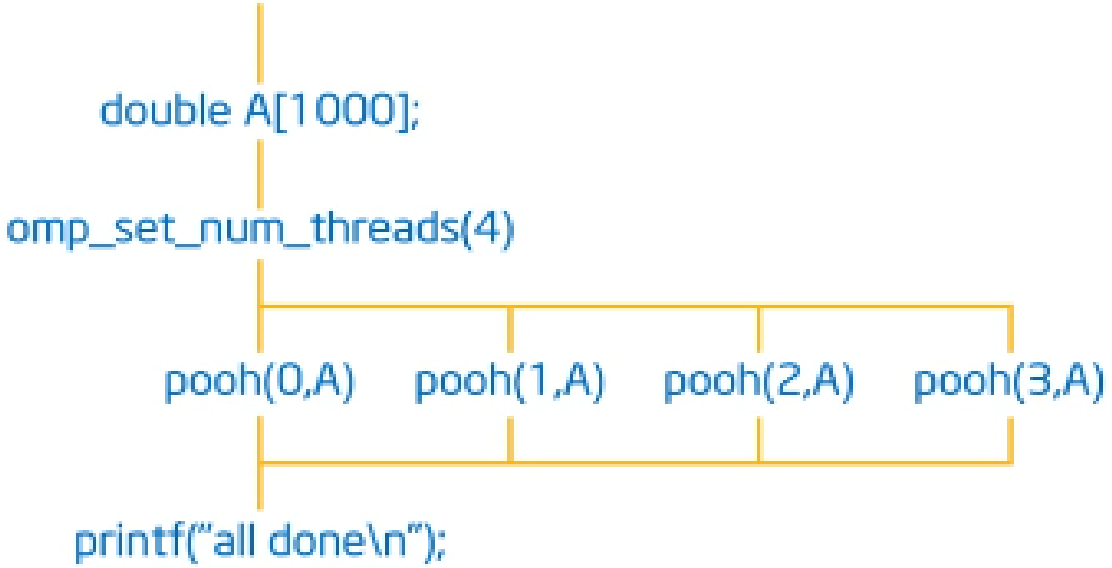






You create threads in OpenMP
with the parallel construct

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
printf("all done\n");
```




```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
printf("all done\n");
```

How OpenMP Interacts with Lower Level Runtime ?

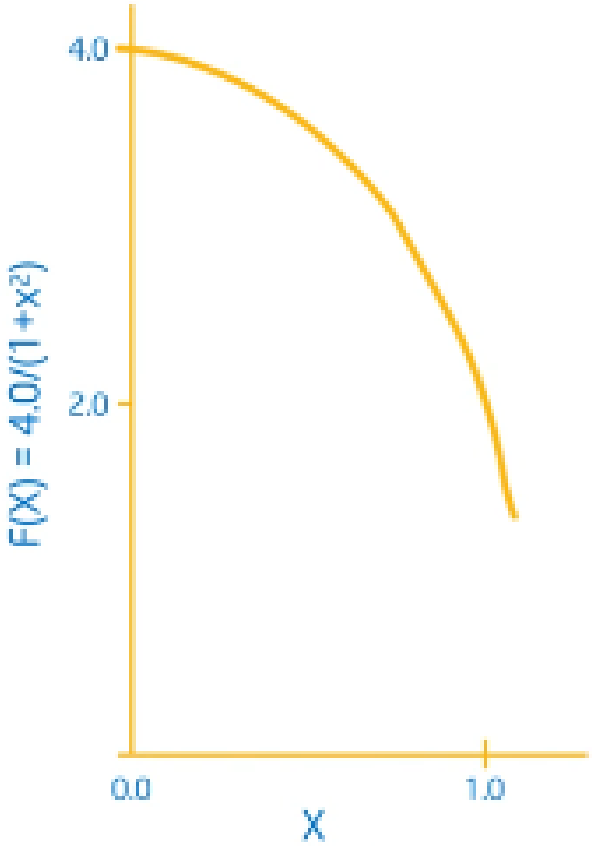
```
#pragma omp parallel num_threads(4)
{
      foobar ();
}
```

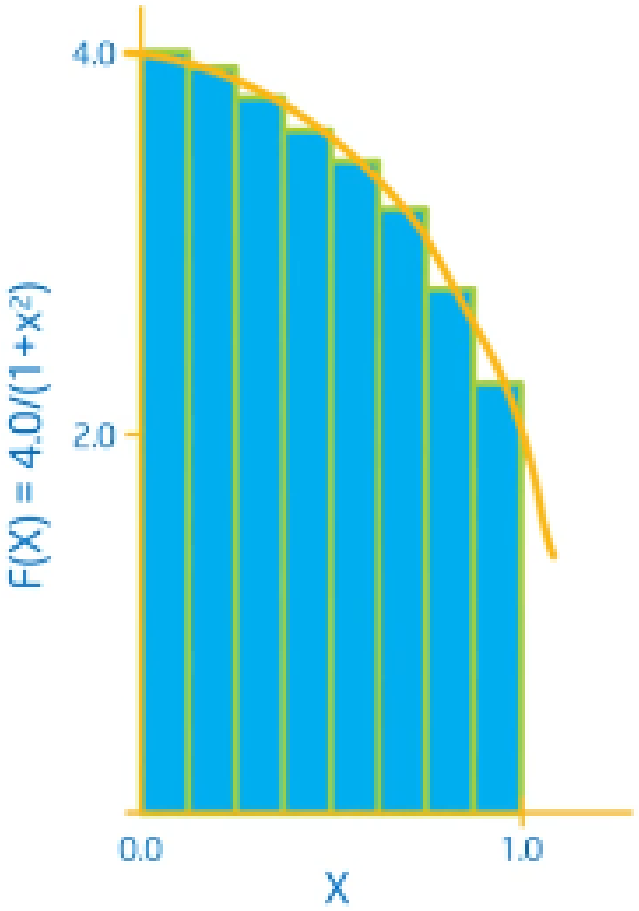
```
void thunk ()
{
      foobar ();
}

pthread_t tid[4];
for (int i = 1; i < 4; ++i)
   pthread_create (
         &tid[i],0,thunk, 0);
think();

for (int i = 1; i < 4; ++i)
   pthread_join (tid[i]);
```

```
#pragma omp parallel num_threads(4)
{
      foobar ();
}
```

Pi Program

$$\int_0^1 \frac{4.0}{(1+X^2)} \qquad \int_0^1 \frac{4.0}{(1+X^2)}\, dx = \pi$$






```c
#include<stdio.h>
#include<omp.h>

static long num_steps = 1000000;
double step;
int main()
{
        int i;
        double x,pi,sum;
        step = 1.0/(double) num_steps;

        double before = omp_get_wtime();

        for(i=0 ; i<num_steps; i++)
        {
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;

        double after = omp_get_wtime();

        printf("PI : %lf Time : %lf\n",pi,after - before);
}
```

```c
#include<stdio.h>
#include<omp.h>
static long num_steps = 1000000;
double step;
int main()
{
    int i;
    double x;                                   // Store the middile of each step rectangle
    double pi;                                  // Store the Final Output
    double sum_up_height=0.0;
    step = 1.0/(double) num_steps;              //Get Each Step Size
    double before = omp_get_wtime();            //Time Before Executing Loop
    for(i=0 ; i<num_steps; i++)                 // Iterate the loop up to no. of steps
    {
        x = (i+0.5)*step;                       // Middle of each step rectangle
        sum_up_height = sum_up_height + 4.0/(1.0+x*x);     // Adding the height to create a big rectangle
    }
    pi = step * sum_up_height;                  // Get the area of big rectangle
    double after = omp_get_wtime();             // Time After Executing Loop

    printf("PI : %lf Time : %lf\n",pi,after - before);
}
```

In addition to a parallel construct, you will need the runtime library routines

- ✓ int omp_get_num_threads();

  Number of threads in a team;

- ✓ intomp_get_thread_num();

  Thread ID and Rank

- ✓ double omp_get_wtime();

  Time in Seconds since a fixed point in the past.

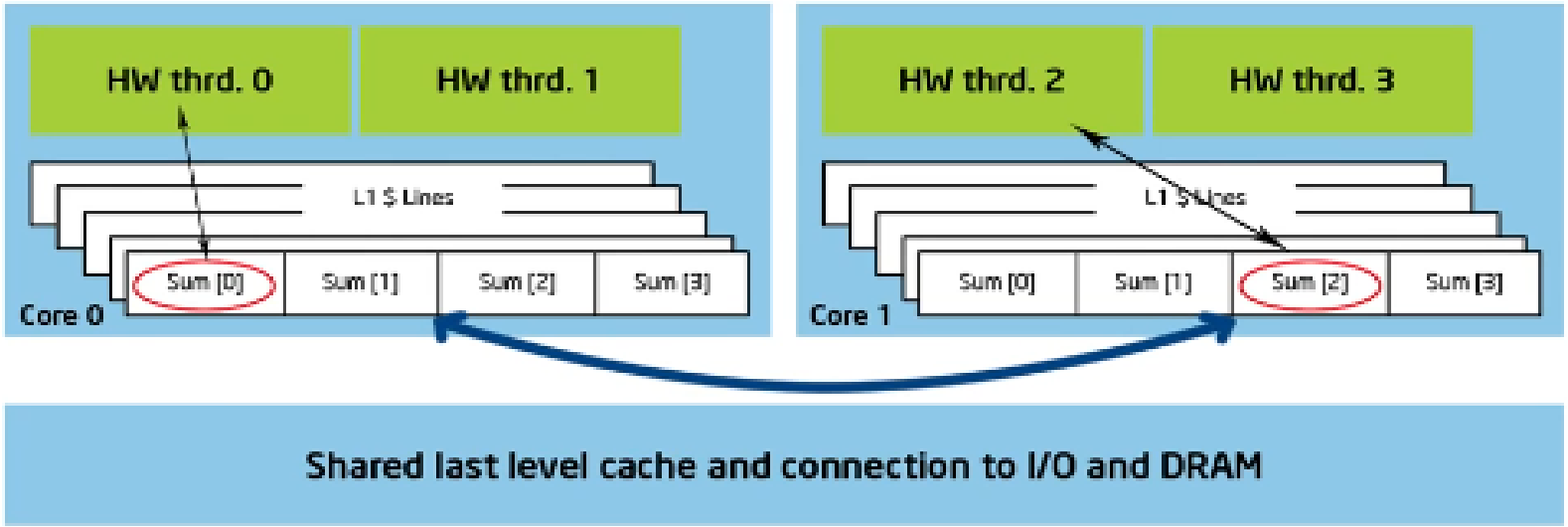
Serial Execution Time : ?

```c
#include<stdio.h>
#include<omp.h>
static long num_steps = 1000000;
double step;
#define NUM_THREADS 4
void main()
{
    int i ,team;
    double pi,sum_up_height[NUM_THREADS];
    step = 1.0/(double)num_steps;
    omp_set_num_threads(NUM_THREADS);
    double before = omp_get_wtime();
    #pragma omp parallel
    {
        int i,id,nthreads;
        double x;
        id = omp_get_thread_num();
        nthreads = omp_get_num_threads();
        if(id == 0)
            team = nthreads;
        for(i=id,sum_up_height[id]=0.0; i< num_steps; i=i+nthreads)
        {
            x=(i+0.5)*step;
            sum_up_height[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0,pi=0.0;i<team;i++)
    {
        pi += sum_up_height[i] * step;
    }
    double after = omp_get_wtime();
    printf("PI : %lf ,Time : %lf\n",pi,after - before);
}
```

Thread	Time
1	?
2	?
3	?
4	?

False Sharing

If independent data elements happen to sit on the same cache line, each update will cause the cache lines to "slosh back and forth" between threads. This is called "**false sharing**"



If you promote scalars to an array to support creation of an SPMD program, the array elements are contiguous in memory and hence share cache lines. This results in poor scalability.

```c
#include<stdio.h>
#include<omp.h>
static long num_steps = 1000000;
double step;
#define PAD 8 //assum_up_heighte 64 byte L1 cache line size
#define NUM_THREADS 4
void main()
{
    int i ,team;
    double pi,sum_up_height[NUM_THREADS][PAD];
    step = 1.0/(double)num_steps;
    omp_set_num_threads(NUM_THREADS);
    double before = omp_get_wtime();
    #pragma omp parallel
    {
        int i,id,nthreads;
        double x;
        id = omp_get_thread_num();
        nthreads = omp_get_num_threads();
        if(id == 0)
            team = nthreads;
        for(i=id,sum_up_height[id][0]=0.0; i< num_steps; i=i+nthreads)
        {
            x=(i+0.5)*step;
            sum_up_height[id][0] += 4.0/(1.0+x*x);
        }
    }
    for(i=0,pi=0.0;i<team;i++)
    {
        pi += sum_up_height[i][0] * step;
    }
    double after = omp_get_wtime();
    printf("PI : %lf ,Time : %lf\n",pi,after - before);
}
```

Thread	Time
1	?
2	?
3	?
4	?

Do we really need to pad our arrays?

- Padding arrays requires deep knowledge of the cache architecture.

  Move to a machine with different sized .

  Cache lines and your software performance falls apart.

- There has got to be a better way to deal with false sharing.

Synchronization

Bringing one or more threads to a well defined

and

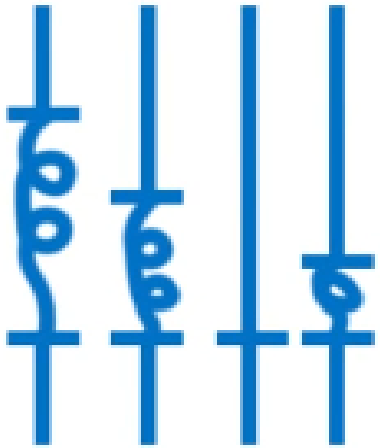
known point in their execution.

High level synchronization:

- ✓ critical
- ✓ atomic
- ✓ barrier
- ✓ ordered

Low level synchronization

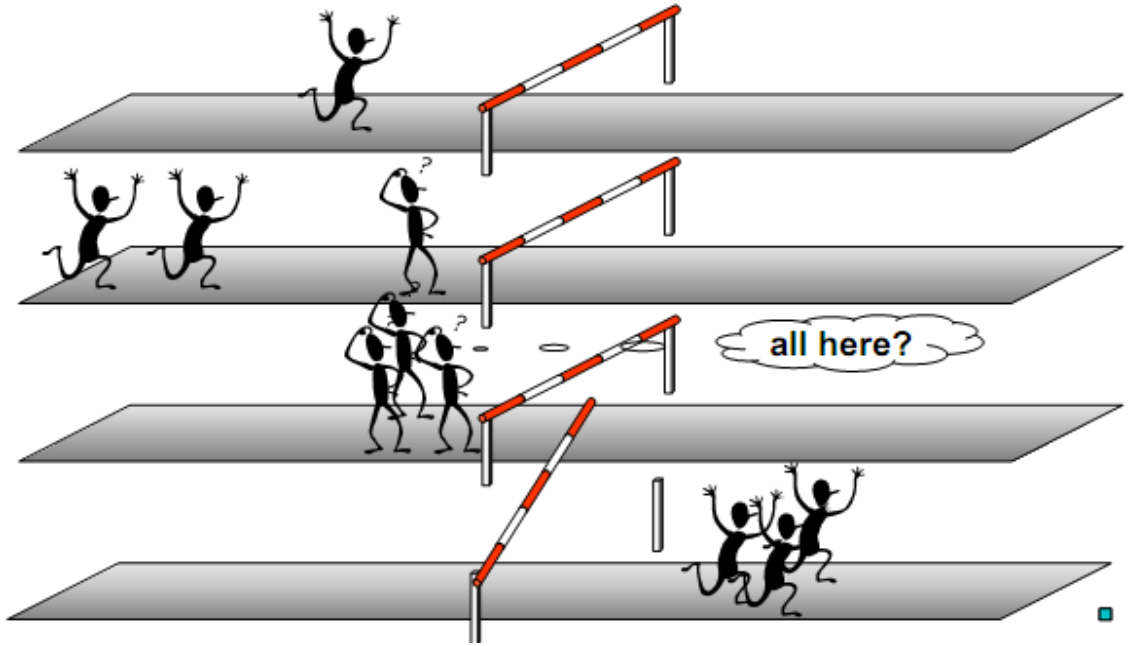
- ✓ flush
- ✓ locks (both simple and nested)

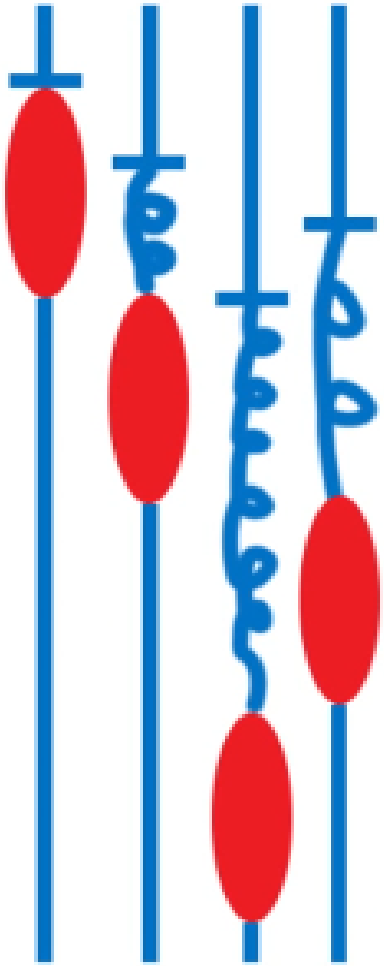


Barrier:

Each thread wait at the barrier
until all threads arrive

```
#pragma omp parallel

{
        int id=omp_get_thread_num();
        A[id] = big_cal1(id);
#pragma omp barrier

        B[id] = big_calc2(id, A);
}
```





Mutual Exclusion:

Define a block of code that only one thread at a time can execute

Critical

```
float res;

#pragma omp parallel

[       float B; int i, id, nthrds;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        for(i=id; i<niters; i+=nthrds){
            B = big_job(i);

#pragma omp critical
                res += consume (B);
        }
}
```

```c
#include<stdio.h>
#include<omp.h>
static long num_steps = 1000000;
double step;
#define NUM_THREADS 4
void main()
{
    double pi=0.0;
    int team;
    step = 1.0/(double)num_steps;
    omp_set_num_threads(NUM_THREADS);
    double before = omp_get_wtime();
    #pragma omp parallel
    {
        int i,id,nthreads;
        double x,sum_up_height;
        id = omp_get_thread_num();
        nthreads = omp_get_num_threads();
        if(id == 0)
            team = nthreads;
        for(i=id,sum_up_height=0.0; i< num_steps; i=i+team)
        {
            x=(i+0.5)*step;
            sum_up_height += 4.0/(1.0+x*x);
        }
        #pragma omp critical
           pi += sum_up_height * step;
    }
    double after = omp_get_wtime();
    printf("PI : %lf ,Time : %lf\n",pi,after - before);
}
```

Thread	Time
1	?
2	?
3	?
4	?

```
#pragma omp parallel

{
        double tmp, B;
        B = DOIT();
        tmp = big_ugly(B);

#pragma omp atomic
          X += tmp;

}
```

The statement inside the atomic must be one of the following forms:

x binop= expr

x++

++x

x—

--x

X is an lvalue of scalar type and binop is a non-overloaded built in operator.

```c
#include<stdio.h>
#include<omp.h>
static long num_steps = 1000000;
double step;
#define NUM_THREADS 4
void main()
{
    double pi=0.0;
    int team;
    step = 1.0/(double)num_steps;
    omp_set_num_threads(NUM_THREADS);
    double before = omp_get_wtime();
    #pragma omp parallel
    {
        int i,id,nthreads;
        double x,sum;
        id = omp_get_thread_num();
        nthreads = omp_get_num_threads();
        if(id == 0)
            team = nthreads;
        for(i=id,sum=0.0; i< num_steps; i=i+team)
        {
            x=(i+0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
                sum = sum * step;
        #pragma omp atomic
           pi += sum ;
    }
    double after = omp_get_wtime();
    printf("PI : %lf ,Time : %lf\n",pi,after - before);
}
```

Thread	Time
1	?
2	?
3	?
4	?

Work Sharing

- Loop Construct
- Section Construct
- Single Construct
- Task Construct

Loop Construct

```
#pragma omp parallel

{
#pragma omp for
        for (I=0; I<N; I++){
                NEAT_STUFF(I);
        }
}
```

Loop Worksharing Constructs

Sequential Code

```
for(i=0;i<N;i++)  { a[i] = a[i] + b[i];}
```

OpenMP Parallel Region

```
#pragma omp parallel
{
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
        Nthrds = omp_get_num_threads();
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        if (id == Nthrds-1)iend = N;
        for(i=istart;i<iend;i++)  { a[i] = a[i] + b[i];}
}
```

OpenMP Parallel Region and
a Worksharing for Construct

```
#pragma omp parallel
#pragma omp for
        for(i=0;i<N;i++)  { a[i] = a[i] + b[i];}
```

(OpenMP makes the loop control index on a parallel loop private to a thread)

Loop Worksharing Constructs:

The Schedule Clause

The schedule clause affects how loop iterations are mapped onto threads

schedule(static [,chunk])	Deal-out blocks of iterations of size "chunk" to each thread.
schedule(dynamic[,chunk])	Each thread grabs "chunk" iterations off a queue until all iterations have been handled.
schedule(guided[,chunk])	Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size "chunk" as the calculation proceeds.
schedule(runtime)	Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library).
schedule(auto)	Schedule is left up to the runtime to choose (does not have to be any of the above).

When To Use The Schedule Clause

Static

- ✓ Predetermined and predictable by Programmer.
- ✓ Least work at runtime.
- ✓ Scheduling done at compile time.

Dynamic

- ✓ Unpredictable, highly variable work per iteration.
- ✓ Most work at Runtime.
- ✓ Complex scheduling logic at runtime.

Runtime

schedule(runtime)

omp_set_schedule()

omp_get_schedule()

Added a new schedule kind AUTO which gives full freedom to the runtime to determine the scheduling of iterations to threads.

```
   double res[MAX]; int i;
#pragma omp parallel
{
   #pragma omp for
   for (i=0;i<MAX; i++){
       res[i] = huge():
   }
}
```

```
   double res[MAX]; int i;
#pragma omp parallel for
   for (i=0;i<MAX; i++){
       res[i] = huge():
   }
```

Working With Loops

Find the compute-intensive loops

Make the loop iterations independent so they can safely execute in any order without loop-carried dependencies

Place the appropriate OpenMP directive and test

```
int i, j, A[MAX];
j = 5;
for (i=0;i<MAX; i++)[
      j +=2;
      A[i] = big(j);
}
```

(OpenMP makes the loop control index on a parallel loop private to a thread)

Working With Loops

Find the compute-intensive loops

Make the loop iterations independent so they can safely execute in any order without loop-carried dependencies

Place the appropriate OpenMP directive and test

```
int i, j, A[MAX];
#pragma omp parallel for
for (i=0;i<MAX; i++)[
        int j = 5 + 2*(i+1);
        A[i] = big(j);
}
```

(OpenMP makes the loop control index on a parallel loop private to a thread)

```
double  ave=0.0,A[MAX];   int i;
for (i=0;i<MAX; i++){
    ave + = A[i];
}
ave = ave/MAX;
```

Reduction

OpenMP reduction clause:

reduction (op : list)

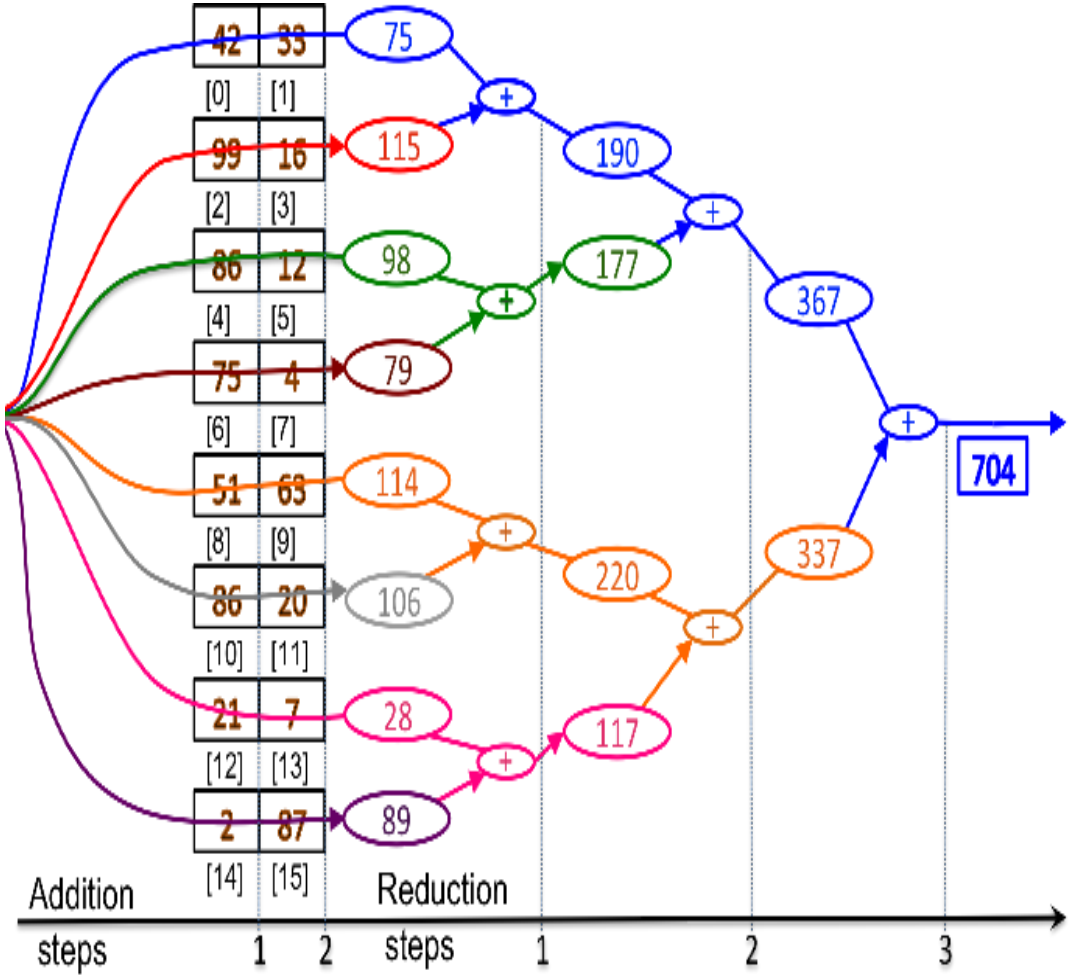
A local copy of each list variable is made and initialized depending on the "op"

e.g. 0 for "+"

Updates occur on the local copy

Local copies are reduced into a single value and combined with the original global value

```
double  ave=0.0, A[MAX];   int i;
#pragma omp parallel for reduction (+:ave)
    for (i=0;i<MAX; i++){
     ave + = A[i];
}
ave = ave/MAX;
```



OpenMP: Reduction operands/initial-values

Many different associative operands can be used with reduction.
Initial values are the ones that make sense mathematically.

Operator	Initial Value
+	0
*	1
-	0
min	Largest pos num
max	Most neg num

C/C++ only

Operator	Initial Value
&	~0
|	0
^	0
&&	1
||	0

Fortran only

Operator	Initial Value
.AND.	.true.
.OR.	.false.
.NEVQ.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.

Serial Code

```c
#include<stdio.h>
#include<omp.h>

static long num_steps = 1000000;
double step;
int main()
{
        int i;
        double x,pi,sum;
        step = 1.0/(double) num_steps;

        double before = omp_get_wtime();

        for(i=0 ; i<num_steps; i++)
        {
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;

        double after = omp_get_wtime();

        printf("PI : %lf Time : %lf\n",pi,after - before);
}
```

Parallel with Reduction Clause

```c
#include<stdio.h>
#include<omp.h>
static long num_steps = 1000000;
double step;
#define NUM_THREADS 4
void main()
{
    int i;double x,pi,sum=0.0;
    step = 1.0/(double)num_steps;
    omp_set_num_threads(NUM_THREADS);
    double before = omp_get_wtime();
    #pragma omp parallel
    {
        double x;
        #pragma omp for reduction(+:sum)
            for(i=0;i<num_steps;i++)
                {
                    x = (i+0.5)*step;
                    sum = sum + 4.0/(1.0+x*x);
                }
    }
    pi = step * sum;
    double after = omp_get_wtime();
    printf("PI : %lf ,Time : %lf\n",pi,after - before);
}
```

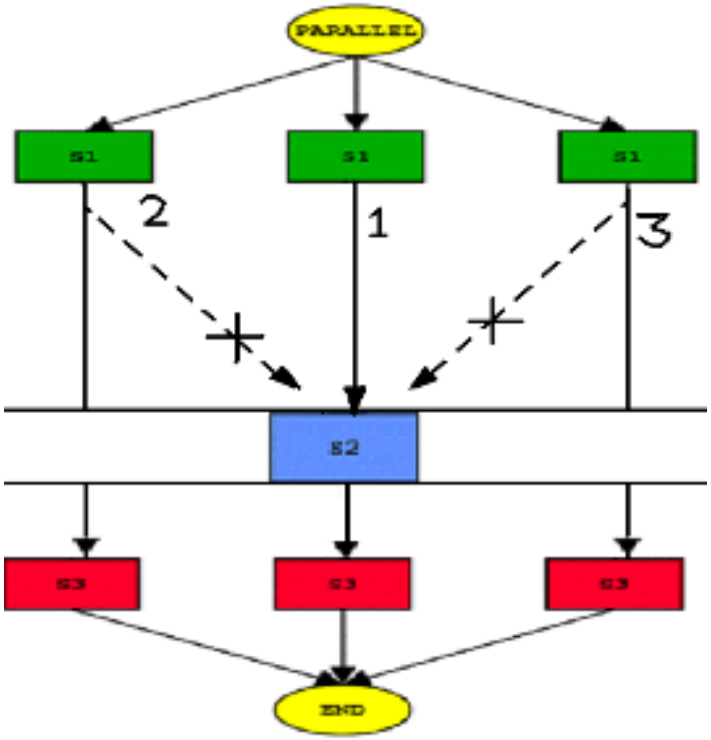
Private & Shared Clause

```
#pragma omp parallel shared (A, B, C) private(id)
{
      id=omp_get_thread_num();
      A[id] = big_calc(id);
#pragma omp barrier
#pragma omp for
      for(i=0;i<N;i++){C[i]=big_calc3(i,A);}
#pragma omp for nowait
      for(i=0;i<N;i++){ B[i]=big_calc2(C, i); }
      A[id] = big_calc4(id);
}
```

#pragma omp parallel for private(i), firstprivate(b), lastprivate(a)

Barrier (Explicit / Implicit)

Single Clause




```
#pragma omp parallel
{
        do_many_things();
#pragma omp single
        {   exchange_boundaries();   }
        do_many_other_things();
}
```

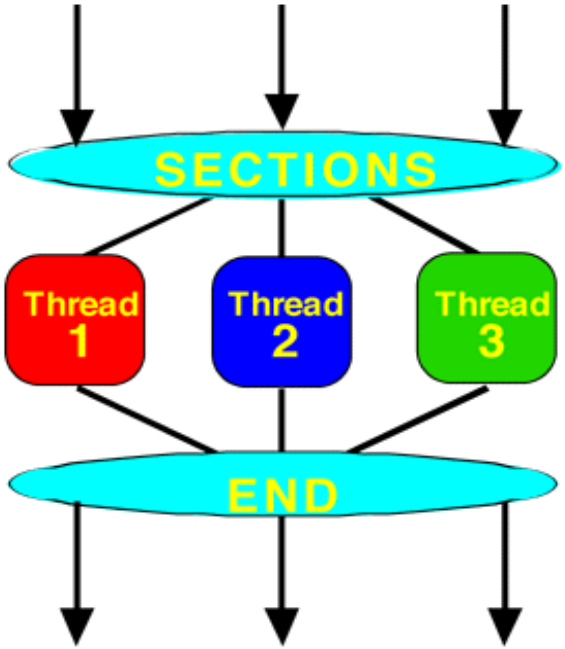
Barrier (Explicit / Implicit)

Master Clause

```
#pragma omp parallel
{
        do_many_things();
#pragma omp master
        {   exchange_boundaries();   }
#pragma omp barrier
        do_many_other_things();
}
```

Barrier (Explicit / Implicit)

Sections




```
#pragma omp parallel
{
    #pragma omp sections
    {
    #pragma omp section
        x_calculation();
    #pragma omp section
        y_calculation();
    #pragma omp section
        z_calculation();
    }
}
```

Some other clauses ……

Changing Storage Attributes

SHARED

PRIVATE

FIRSTPRIVATE

LASTPRIVATE

DEFAULT (PRIVATE | SHARED | NONE)

```
void wrong() {
    int tmp = 0;
#pragma omp parallel for private(tmp)
    for (int j = 0; j < 1000; ++j)
            tmp += j;
    printf("%d\n", tmp);
}
```

```
incr = 0;
#pragma imp parallel for firstprivate(incr)
for (i = 0; i <= MAX; i++) {
        if ((i%2)==0) incr++;
        A[i] = incr;
}
```

```
void sq2(int n, double *lastterm)
{
   double x; int i;
   #pragma omp parallel for lastprivate(x)
   for (i = 0; i < n; i++){
      x = a[i]*a[i] + b[i]*b[i];
      b[i] = sqrt(x);
   }
   *lastterm = x;
}
```

Data Sharing:
A Data Environment Test

variables: A = 1, B = 1, C = 1
#pragma omp parallel private(B) firstprivate(C)

Inside this parallel region:

B and C are local to each thread
A is shared by all threads

C's initial value equals 1
B's initial value is undefined

Following the parallel region:

B and C revert to their original values of 1
A is either 1 or the value it was set to inside the parallel region

OpenMp Constructs

OpenMP Constructs

To create a team of threads
#pragma omp parallel

To share work between threads
#pragma omp for
#pragma omp single

To prevent conflicts (prevent races)
#pragma omp critical
#pragma omp atomic
#pragma omp barrier
#pragma omp master

Data environment clauses
private (variable_list)
firstprivate (variable_list)
lastprivate (variable_list)
reduction(+:variable_list)

Runtime Library Routines

omp_set_num_threads()

omp_get_num_threads()

omp_get_thread_num()

omp_get_max_threads()

omp_in_parallel()

omp_set_dynamic()

omp_get_dynamic()

```
#include <omp.h>
void main()
{  int num_threads;
    omp_set_dynamic( 0 );
    omp_set_num_threads( omp_num_procs() );
#pragma omp parallel
  {    int id=omp_get_thread_num();
#pragma omp single
          num_threads = omp_get_num_threads();
       do_lots_of_stuff(id);
  }
}
```

```
#include <omp.h>
void main()
{  int num_threads;
    omp_set_dynamic( 0 );
    omp_set_num_threads( omp_num_procs() );
    omp_get_num_threads(): // how many??
#pragma omp parallel
  {    int id=omp_get_thread_num();
#pragma omp single
          num_threads = omp_get_num_threads();
       do_lots_of_stuff(id);
  }
}
```

Runtime Library Routines

omp_in_parallel()

omp_set_dynamic()

omp_get_dynamic()

omp_num_procs()

Environment Variables

OMP_NUM_THREADS
int_literal

OMP_STACKSIZE

OMP_WAIT_POLICY
ACTIVE | PASSIVE

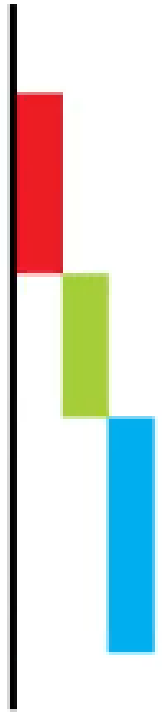
OMP_PROC_BIND
TRUE | FALSE

OpenMP Tasks

Tasks are independent units of work

OpenMP Tasks

Tasks are independent units of work



OpenMP Tasks

Tasks are independent units of work

Tasks are composed of:

- **code** to execute
- **data** environment
- **internal control variables** (ICV)

The runtime system decides when tasks are executed

```
#pragma omp parallel
{
    #pragma omp task
    foo();
    #pragma omp barrier
    #pragma omp single
    {
        #pragma omp task
        bar();
    }
}
```

Lock Routines

omp_init_lock()

omp_set_lock()

omp_unset_lock()

omp_destroy_lock()

omp_test_lock()

Good Things about OpenMP

- Incrementally Parallelization of sequential code.
- Leave thread management to compiler.
- Directly supported by compiler.

Limitation

- Requires compiler which supports OpenMP
- Internal details are hidden
- Run efficiently in shared-memory multiprocessor platforms only but not on distributed memory
- Limited performance by memory architecture

References

- https://computing.llnl.gov/tutorials/openMP/
- http://wiki.scinethpc.ca/wiki/images/9/9b/Ds-openmp.pdf
- www.openmp.org/
- http://openmp.org/sc13/OpenMP4.0_Intro_YonghongYan_SC13.pdf

Thank You