HUST-USYD Summer School on Parallel Programming Practice – Lecture 4

Bing Bing Zhou (bing.zhou@sydney.edu.au)

School of Computer Science, University of Sydney

THE UNIVERSITY OF
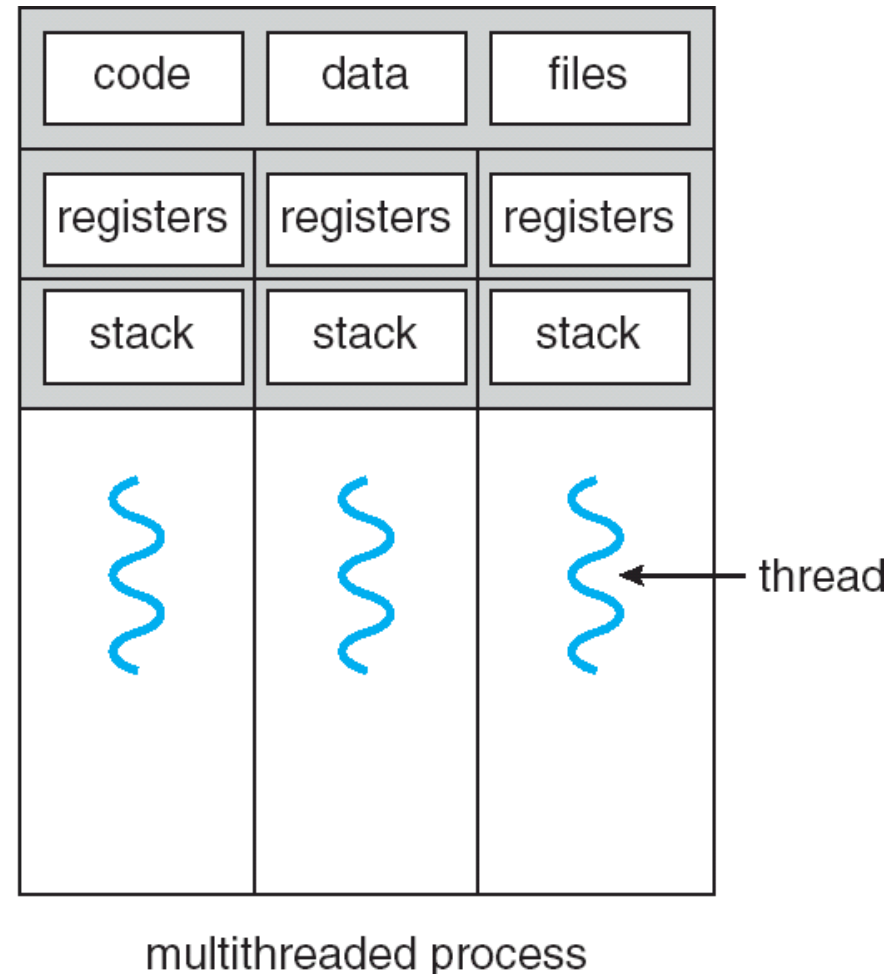SYDNEY

Outline

- Review of Homework 2
- Program shared memory platform
- Example: matrix multiplication
  - Matrix partitioning
  - Locality
- OpenMP
  - Execution model
  - General code structure
  - Basic omp constructs and directives
    - Parallel region construct
      - Example: Hello World
      - Lab exercise 1: Hello World
    - Work-sharing construct: for directive
      - Example: matrix-vector multiplication
      - Lab exercise 2: matrix multiplication

Shared Memory Platform

– Multiple threads created and running concurrently

 – A set of global variables shared by all threads
 – Threads also have their own local private variables
 – Threads communication implicitly using shared variables
 – Threads coordination explicitly by synchronization on shared variables

– Shared memory machines are usually small scale (e.g., ~100 cores for large one)

 – Need coarse grained parallelism

– Multithreading techniques used for shared-memory programming

 – Pthread – explicitly create threads and synchronize thread execution
 – OpenMP – use compiler directives and clauses – we'll study OpenMP in this course

Thread Basics

– One process and multiple threads

  – Threads in a process share the same address space and the process state, which greatly reduces the cost of context switching

  – Threads run concurrently and interact through reads/writes from/to the same locations in the shared address space

  – Cannot assume any order of execution

    • Any such order must be established using synchronization mechanisms

  – Issues on deadlock, starvation and performance

  – All general-purpose modern operating systems support threads

code	data	files
registers	registers	registers
stack	stack	stack

← thread

multithreaded process

Program Shared Memory Platform

- Global variables shared by all threads
  - Seems algorithm design is easier as data don't need to be moved around
- Data partitioning unnecessary?
  - Not true! must consider ILP, memory hierarchy and cache effect
  - Data locality is very important
- Explicit synchronization
  - In shared-memory machines global data are shared by multiple threads
  - Synchronization must be used to
    - Protect access to shared data and so to prevent race condition
    - Impose order constraints
  - Need to prevent possible deadlock when synchronization is applied

Race Condition

- In shared-memory machines synchronization must be used to protect access to shared data and so to prevent race conditions

- A race condition occurs when
  - Multiple threads read and write shared data items
  - The final result becomes unpredictable
    - The output depends on who finishes the race last

- E.g., two threads, one incrementing and one decrementing a shared count at the same time
  - If no synchronization, the value of count could be 4, or 5, or 6 after the two threads completed their operation, assuming the initial value of count is 5
   - Why?

Race Condition

- Each increment, or decrement instruction at the high level involves several machine level instructions (i.e., read-modify-write):

  - count++:
    register1 = count
    register1 = register1 + 1
    count = register1

  - count--:
    register2 = count
    register2 = register2 - 1
    count = register2

- The red and blue machine level instructions could be interleaved during the execution, making the final result non-deterministic

- To prevent race condition, need to ensure increment (or decrement) operation completed before the other can proceeds
  – mutual exclusion

Race Condition

- Each increment, or decrement instruction at the high level involves several machine level instructions (i.e., read-modify-write):
    - count++:
        register1 = count
        register1 = register1 + 1
        count = register1
    - count--:
        register2 = count
        register2 = register2 - 1
        count = register2

- The red and blue machine level instructions could be interleaved during the execution, making the final result non-deterministic

- To prevent race condition, need to ensure increment (or decrement) operation completed before the other can proceeds – mutual exclusion

Example: Matrix Multiplication

- We can draw a task dependency graph for matrix multiplication
- We can easily see from the graph that all output elements c_{ij} can be computed concurrently
- However, this is a fine-grained parallelism
- For shared-memory machines we need more coarse-grained parallelism
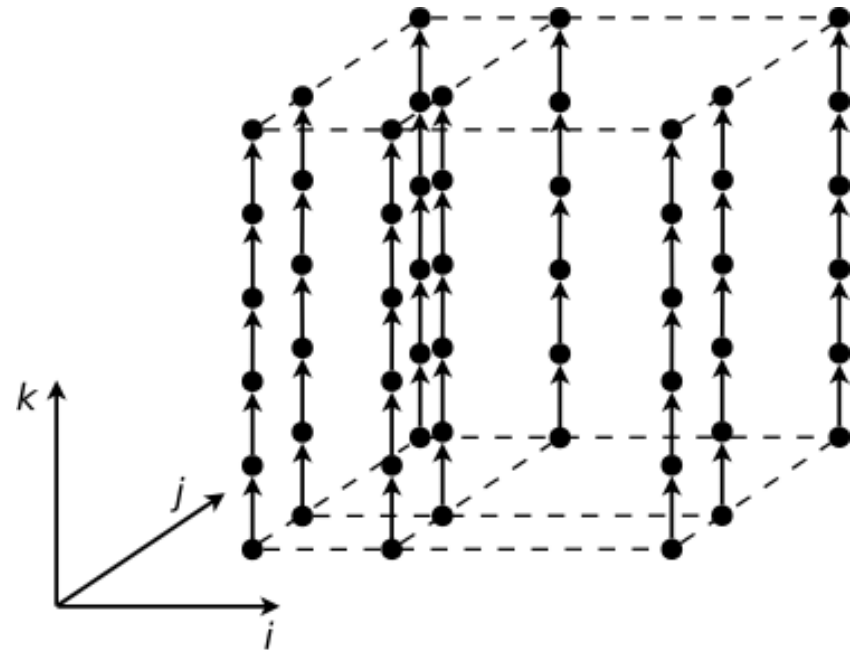
Example: Matrix Multiplication

- For coarse grained parallelism, partition output matrix C
  - No data dependency, thus no synchronization
    - Embarrassingly parallel
  - Note regular data structure & known workload
    - Static partitioning – relatively easy
- Different ways to partition matrix C and assign work
 - 1D Cyclic
  - 1D block
  - 2D block

Example : Matrix Multiplication

– 1D Cyclic Partitioning:
– In 1D cyclic partitioning, rows are assigned to the threads one by one in a cyclic manner until all the rows are assigned

t_0

t_1

t_2

t_3

t_0

t_1

t_2

t_3
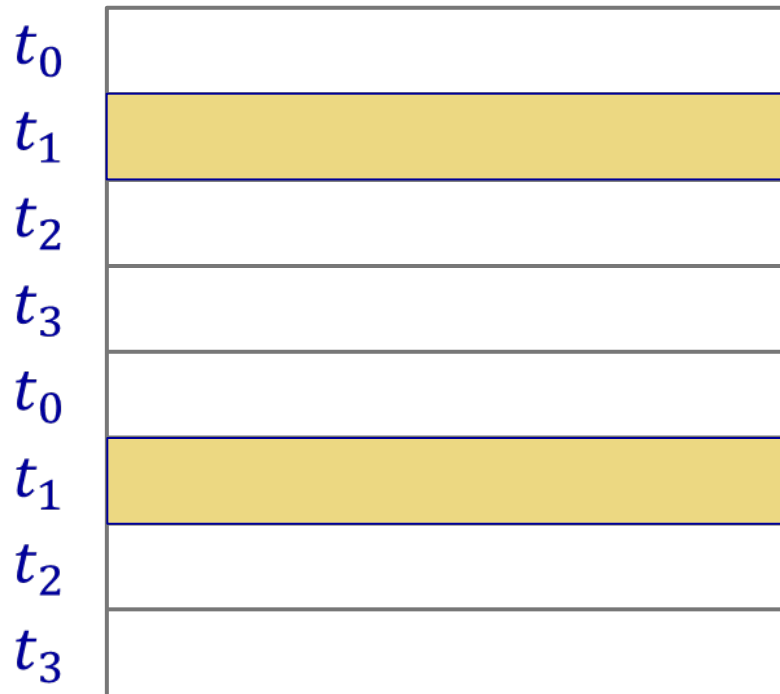
Example: Matrix Multiplication

– 1D Block Partitioning:

– In 1D block partitioning, rows are first grouped into blocks and then each block is assigned one thread
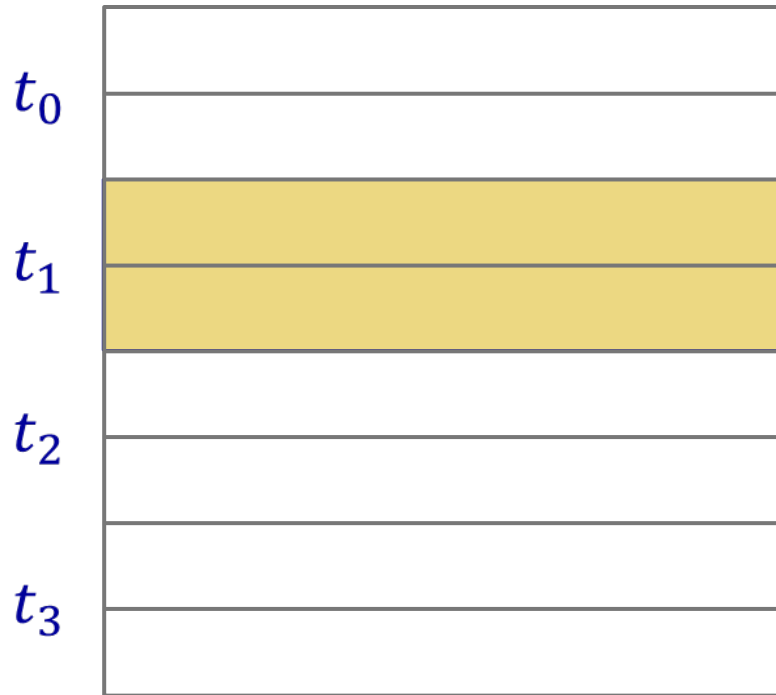
Example: Matrix Multiplication

– 2D Block Partitioning:

– In 2D block partitioning, both rows and columns are first grouped into blocks and then each block (smaller 2D matrix) is assigned one thread

t_{00}	t_{01}	t_{02}	t_{03}
t_{10}	t_{11}	t_{12}	t_{13}
t_{20}	t_{21}	t_{22}	t_{23}
t_{30}	t_{31}	t_{32}	t_{33}

Example: Matrix Multiplication

- Which one is better, 1D or 2D partitioning?
- We have to consider input matrices A and B
- Locality is very important
    - Increase opportunities for ILP
   - Increase computational intensity (memory hierarchy)

Example: *Matrix Multiplication*

- Locality

- 1D cyclic: one row in C needs one row in A and whole B
  - Essentially MV multiplication – low computational intensity
 - If B is larger than cache size – a lot of cache misses

C = A X B
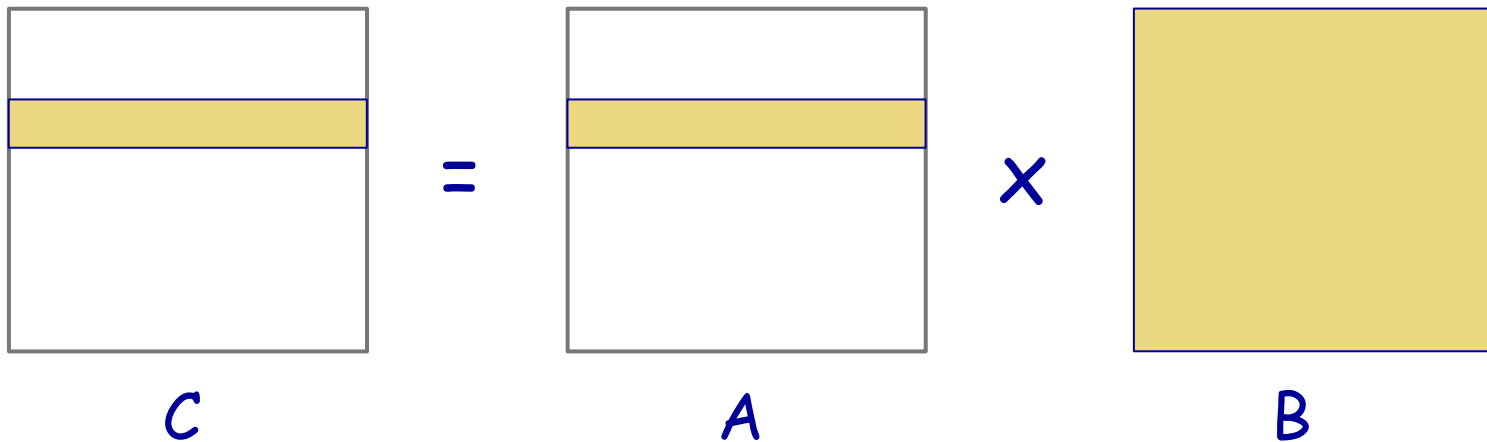
Example: Matrix Multiplication

– Locality

– 1D block: better

- Matrix multiplication – higher computational intensity
- But still needs whole B – a lot of cache misses if B is large
- To improve – further partition row blocks – 2D blocking!

C   =   A   X   B
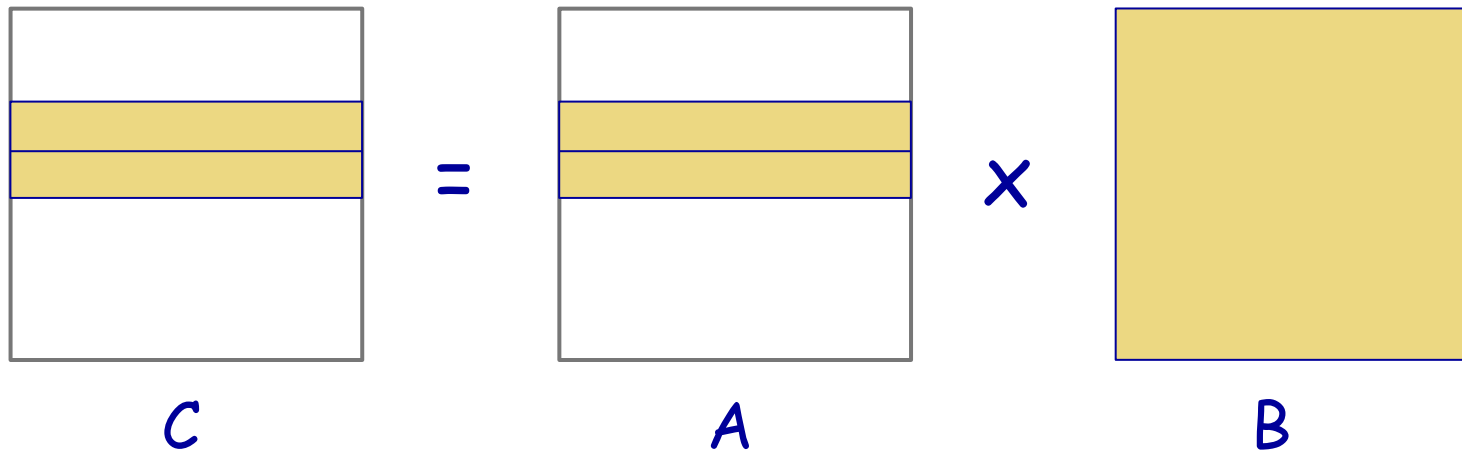
Example: Matrix Multiplication

- Locality
- 2D block:
  - Matrix multiplication – high computational intensity
  - Only need one block row of A and one block column of B
  - Block row of A and block column of B can be further partitioned into small blocks
    - Block matrix multiplication – each time only need three small blocks
    - Fit into the cache to minimize cache misses

$$C = A \times B$$
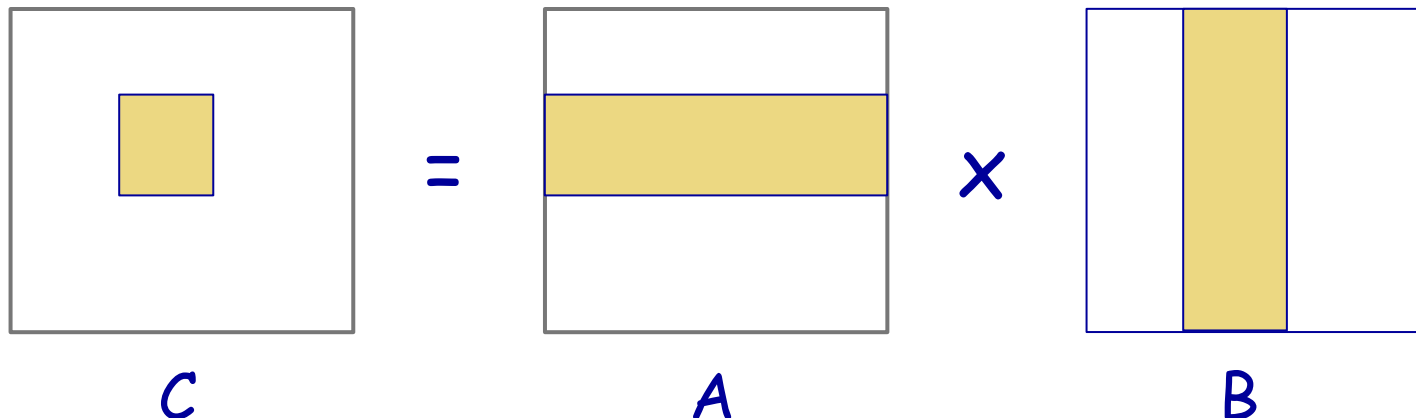
Example: Matrix Multiplication

– Matrix Multiplication on SM Machine

  – There is no data dependency – embarrassingly parallel

  – Linear speedup is achievable

  – The focus should be on sequential computation on each core to achieve high efficiency

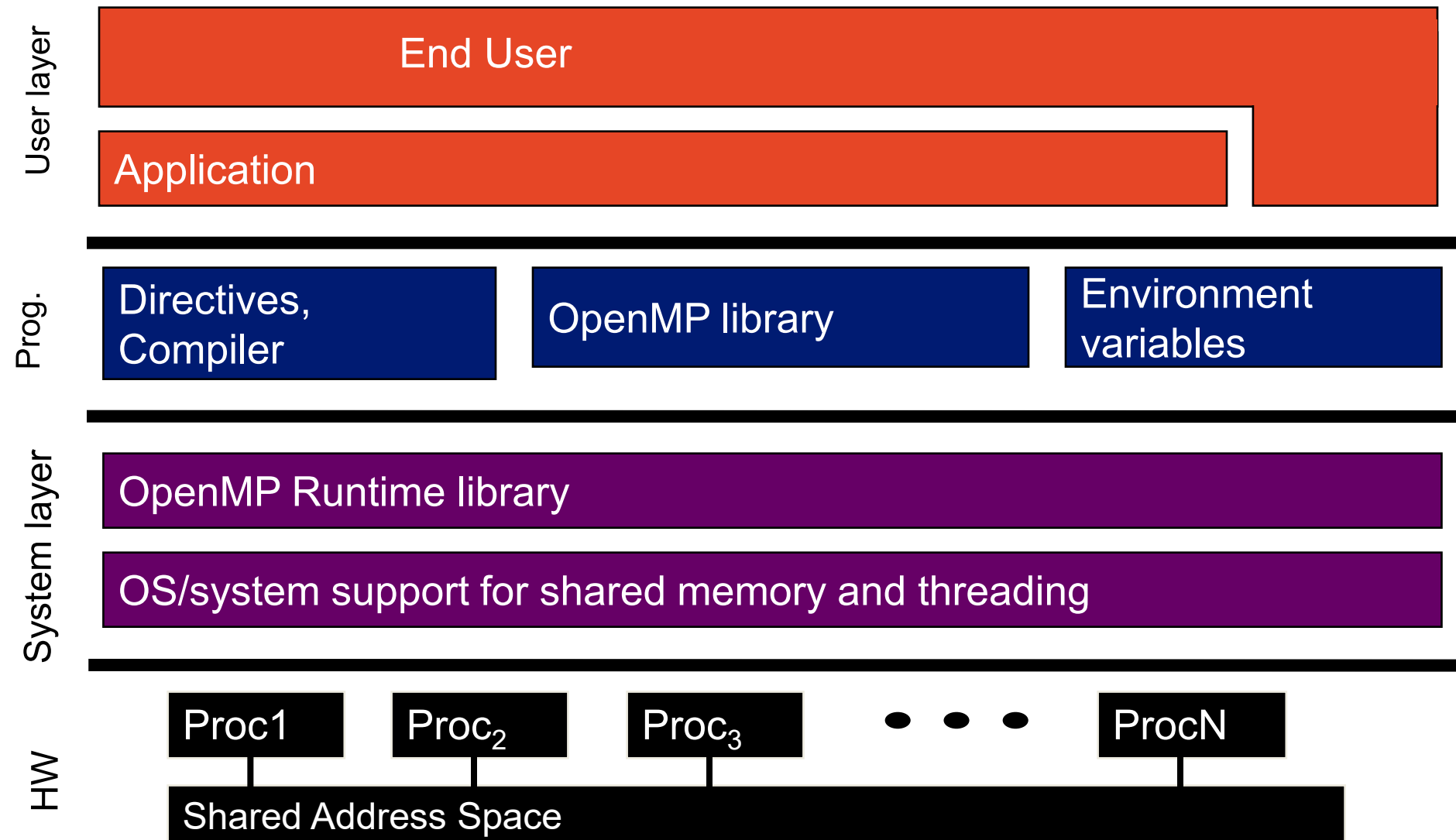   • e.g., cache effect, memory hierarchy, loop unrolling and ILP

Shared-Memory Programming with OpenMP

- **Many materials on OpenMP used in this course are derived from:**
  - https://hpc-tutorials.llnl.gov/openmp/
  - https://cvw.cac.cornell.edu/OpenMP/
  - https://sites.google.com/lbl.gov/cs267-spr2018/
  - https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf
- OpenMP is an acronym for Open Multi-Processing
  - openmp.org – Talks, examples, forums, etc
- OpenMP is a directive-based Application Programming Interface (API) for developing parallel programs on shared memory architectures
- High-level API for programming in C/C++ and Fortran
  - Preprocessor (compiler) directives
    - Compiler directives specify what type of code should be generated, using commands starting with pragma built into comment
    - #pragma omp construct [clause [clause …]]
  - Library Calls
    - #include <omp.h>
  - Environment Variables

Motivation

- Thread libraries, e.g., Pthread are hard to use
  - Pthread threads have many library calls for initialization, synchronization, thread creation, condition variables, etc
  - Synchronization between threads introduces a new dimension of program correctness

- How about to develop a parallel compiler which automatically convert sequential programs into parallel ones?
  - Tried many years with very limited success

- This is when explicit parallelization through OpenMP directives comes into the picture
  - Parallelize sequential programs with relatively few annotations that specify parallelism and independence
  - Only a small API that hides cumbersome threading calls with simpler directives

OpenMP Basic Solution Stack

User layer

End User

Application

Prog.

Directives, Compiler

OpenMP library

Environment variables

System layer

OpenMP Runtime library

OS/system support for shared memory and threading

HW

Proc1   Proc$_2$ Proc$_3$ • • •   ProcN

Shared Address Space

A Programmer's View of OpenMP

– OpenMP is a portable, threaded, shared-memory programming specification with "light" syntax
  – Requires compiler support (C, C++ or Fortran)

– OpenMP will:
  – Allow a programmer to separate a program into serial regions and parallel regions, rather than explicitly create concurrently-executing threads
  – Hide stack management
  – Provide some synchronization constructs

– OpenMP will not:
  – Parallelize automatically
  – Guarantee speedup
  – Provide freedom from data races

OpenMP Execution Model

– Fork-Join Parallelism
  – Primary (or Master) thread spawns a team of threads as needed
  – Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program
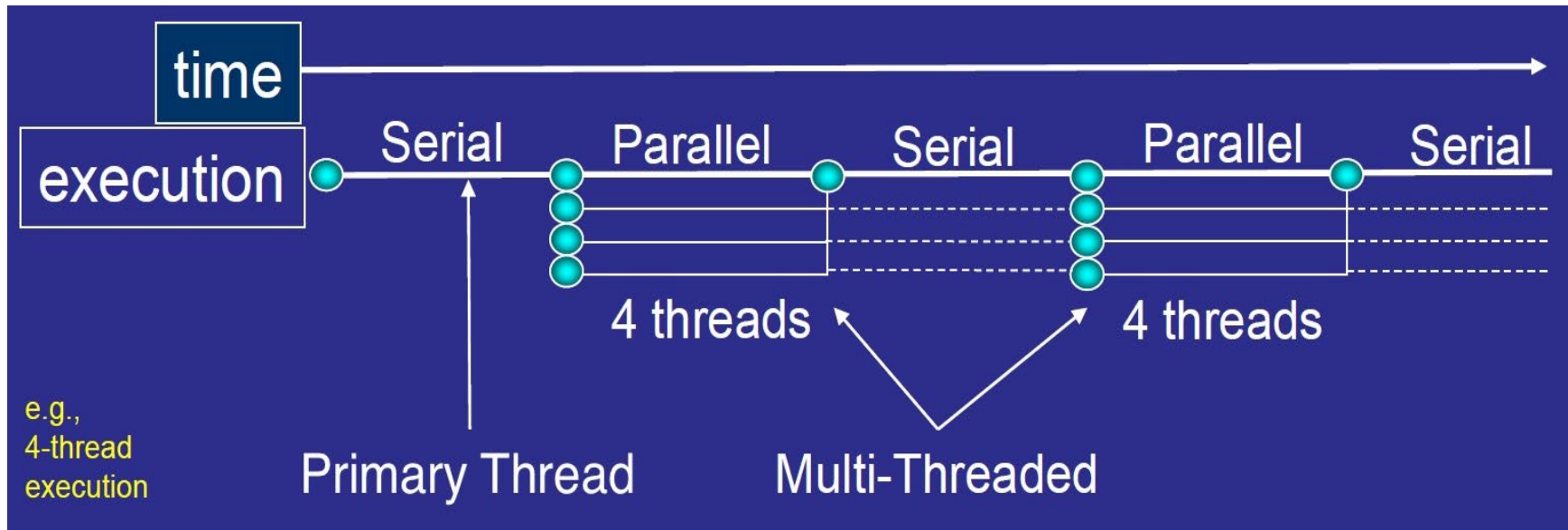
C / C++ - General Code Structure

```
#include <omp.h>
main ()  {
int var1, var2, var3;

//Serial code
    .
//Beginning of parallel section. Fork a team of threads.
//Specify variable scoping
#pragma omp parallel private(var1, var2) shared(var3)
{

//Parallel section executed by all threads
        .
//Other OpenMP directives
        .
//Run-time Library calls
        .
 } //All threads join master thread and disband

//Resume serial code
    .
 }
```

The OpenMP Common Core: Most OpenMP programs only use these 19 items

OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads
int omp_get_thread_num() int omp_get_num_threads()	Create threads with a parallel region and split up the work using the number of threads and thread ID
double omp_get_wtime()	Speedup and Amdahl's law. False Sharing and other performance issues
setenv OMP_NUM_THREADS N	Internal control variables. Setting the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution.
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies
reduction(op:list)	Reductions of values across a team of threads
schedule(dynamic [,chunk]) schedule (static [,chunk])	Loop schedules, loop overheads and load balance
private(list), firstprivate(list), shared(list)	Data environment
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive)
#pragma omp single	Workshare with a single thread
#pragma omp task #pragma omp taskwait	Tasks including the data environment for tasks.

Parallel Region Construct

#pragma omp parallel [clause ...] newline

- When a thread reaches a parallel directive, it creates a team of threads and becomes the master of the team
- The master is a member of that team and has thread number (or ID) 0 within that team

- Starting from the beginning of this parallel region, all threads will execute the same code in the region

- There is an implied barrier at the end of a parallel section

- Only the master thread continues execution past this point

Parallel Region Construct

#pragma omp parallel [clause ...] newline

– How many threads created?
  – Implementation default - usually the number of CPUs , or cores on a node
  – Use the library function omp_set_num_threads()
  – Set the number of threads using the num_threads clause
    • E.g., #pragma omp parallel num_threads(3)

– To get the number of available cores on a node
  – Use function omp_get_num_procs()

– To check how many threads created
  – Use function omp_get_num_threads()

– Created threads are numbered from 0 (master thread) to nthreads -1
  – To get each thread ID, use function omp_get_thread_num()

Parallel Region Construct

#pragma omp parallel [clause ...] newline

- Shared and private variables
  - Any variable that is **declared outside** a parallel region is **shared**
    - There is only one copy of the variable and all threads refer to the same variable – care is needed whenever such a variable is referred
  - Any variable that is **declared inside** a parallel region is **private**
    - Each thread has its own copy of the variable which is safe to use
- Shared and private clauses
  - private(list) clause declares variables in its list to be private to each thread
    - A new object of the same type is declared once for each thread in the team
  - shared(list) clause declares variables in its list to be shared among all threads in the team

Example: Hello World

– A sequential program

```
#include <stdio.h>
int  main()
{



        printf(" hello world\n");



}
```

Example: Hello World

– OpenMP program

```c
#include <omp.h>
#include <stdio.h>
int  main()
{
    #pragma omp parallel
    {


        printf(" hello world\n");


    }

}
```

OpenMP include file

Parallel region with default number of threads

Compile OpenMP programs with –fopenmp switch:

gcc –fopenmp

End of the Parallel region

Example: Hello World

– OpenMP program

```c
#include <omp.h>
#include <stdio.h>
int  main()
{

  #pragma omp parallel
  {
    int tid;

    tid = omp_get_thread_num();


    printf("Hello World from thread = %d\n", tid);
  }
}
```

OpenMP include file

Parallel region with default number of threads

Local variable

Each thread gets its own ID

End of the Parallel region

Example: Hello World

– OpenMP program

```
#include <omp.h>
#include <stdio.h>
int  main()
{
   int tid;
   #pragma omp parallel private(tid)
    {
      tid = omp_get_thread_num();
      printf("Hello World from thread = %d\n", tid);

    }
}
```

Now tid is a local variable

Lab exercise 1: Hello World

– Modify the above simple "Hello World" OpenMP program

  – Using OpenMP function or clause to set the number of threads (instead of the default number)

  – Only master thread in the parallel region gets (using OpenMP function) and then prints out the number of threads created by the program

Work-Sharing Construct: for Directive

#pragma omp for [clause ...] newline

– The for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team

– The for directive must be inside a parallel region

– E.g.,

```
#pragma omp parallel
{
#pragma omp for
    for (i=0; i<N; i++){
        c(i);
    }
}
```

The loop control index i is made "private" to each thread by default.

Threads wait here until all threads are finished with the parallel for loop before any proceed past the end of the loop
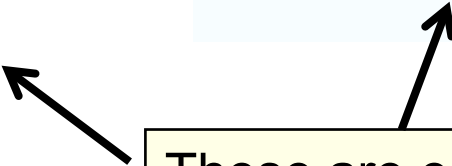
Work-Sharing Construct: for Directive

– OpenMP shortcut: Put the "parallel" and the worksharing directive on the same line

```
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();
    }
```

These are equivalent

Work-Sharing Construct: for Directive

How is work divided and shared?

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];}
}
```

Block partition

OpenMP parallel region and a worksharing for construct

Block partition by default

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

Work-Sharing Construct: for Directive

– Working with loops:

- – Find 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);
}
```

Note: loop index "i" is private by default

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

Remove loop carried dependence

Example: matrix-vector multiplication

– To write an OpenMP program, start with a sequential routine with two loops:

int i, k;

for (i = 0; i < m; i++)

  for (k = 0; k < n; k++)

    b[i] += A[i][k] * x[k]; //assume b initialized already

– Question: Which loop to be parallelized?

Example: matrix-vector multiplication

– Parallelize the i loop:

int i, k;

#pragma omp parallel for shared (A, x, b) private (i, k)

for (i = 0; i < m; i++)

   for (k = 0; k < n; k++)

     b[i] += A[i][k] * x[k]; //assume b initialized already

Lab exercise 2: Matrix Multiplication

– Write an OpenMP program to compute matrix multiplication, assuming A and B are general dense matrices

– Use omp_get_wtime() to evaluate the performance of your code

  – it is a portable wall clock timing routine, and returns a double-precision value equal to the number of elapsed seconds since some point in the past

  – Usually used in "pairs" with the value of the first call subtracted from the value of the second call to obtain the elapsed time for a block of code

Q&A