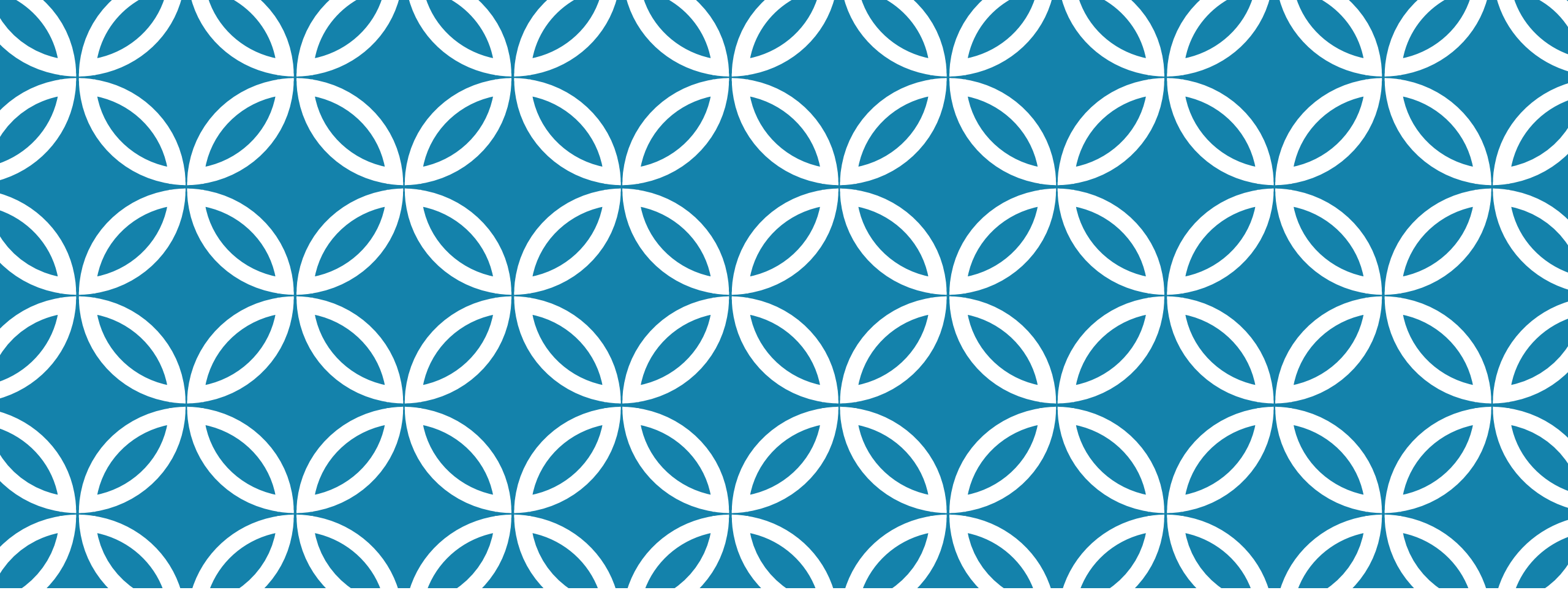
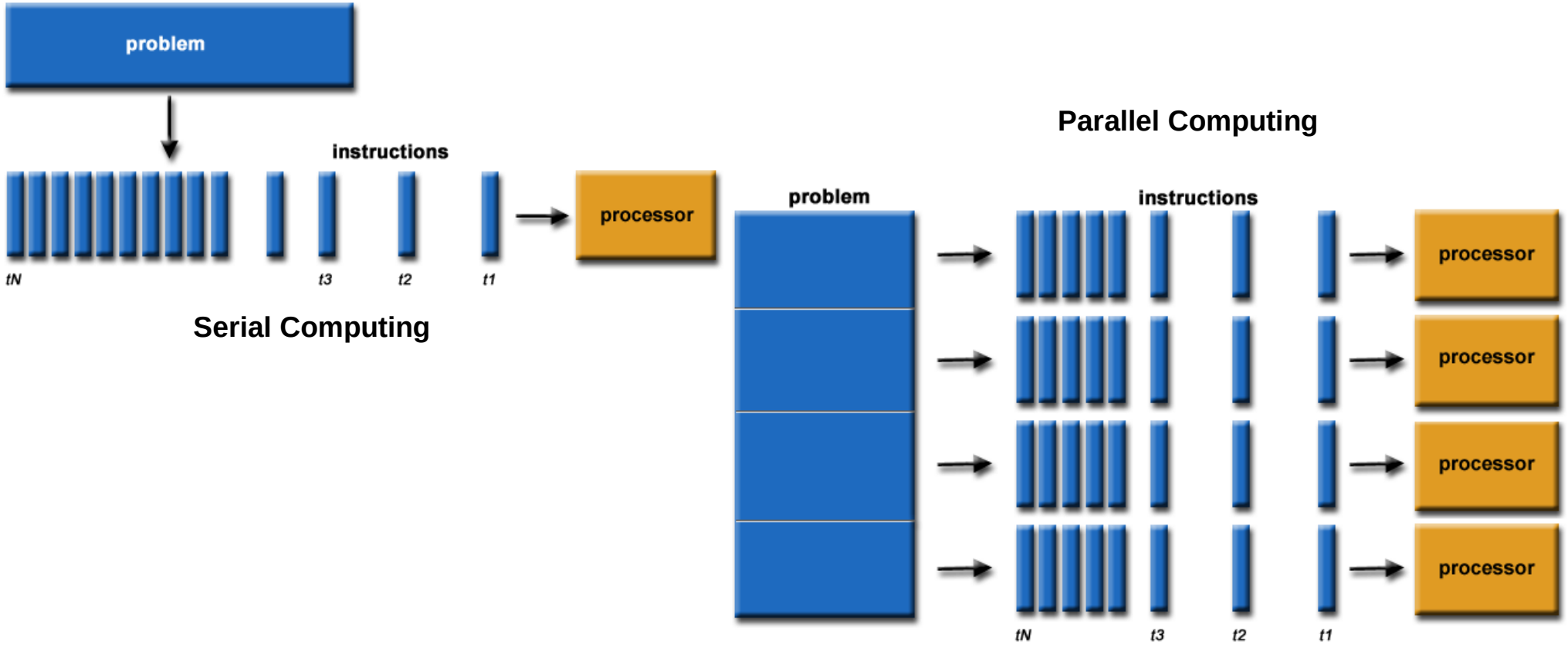


Scope of Parallel Computing

Dr. Ayaz ul Hassan Khan
ayazhk@gmail.com

Parallel Computing



Parallel Computing: Resources

The compute resources can include:

- A single computer with multiple processors;
- A single computer with (multiple) processor(s) and some specialized computer resources (GPU, FPGA …)
- An arbitrary number of computers connected by a network;
- A combination of both.

Parallel Computing: The computational problem

The computational problem usually demonstrates characteristics such as the ability to be:

- Broken apart into discrete pieces of work that can be solved simultaneously;
- Execute multiple program instructions at any moment in time;
- Solved in less time with multiple compute resources than with a single compute resource.

WHY Parallel?

Main Reasons:

- **SAVE TIME AND/OR MONEY**
- **SOLVE LARGER / MORE COMPLEX PROBLEMS**
- **PROVIDE CONCURRENCY**
- **TAKE ADVANTAGE OF NON-LOCAL RESOURCES**
- **MAKE BETTER USE OF UNDERLYING PARALLEL HARDWARE**
- ***parallelism is the future of computing***

Multicore Programming

Multicore or **multiprocessor** systems putting pressure on programmers, challenges include:

- **Dividing activities**
- **Balance**
- **Data splitting**
- **Data dependency**
- **Testing and debugging**

Parallelism implies a system can perform more than one task simultaneously

Concurrency supports more than one task making progress

- Single processor / core, scheduler providing concurrency

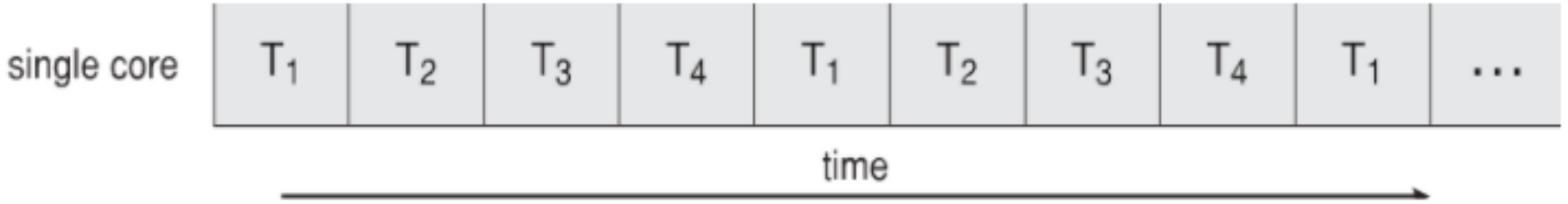
Multicore Programming (Cont.)

Types of parallelism

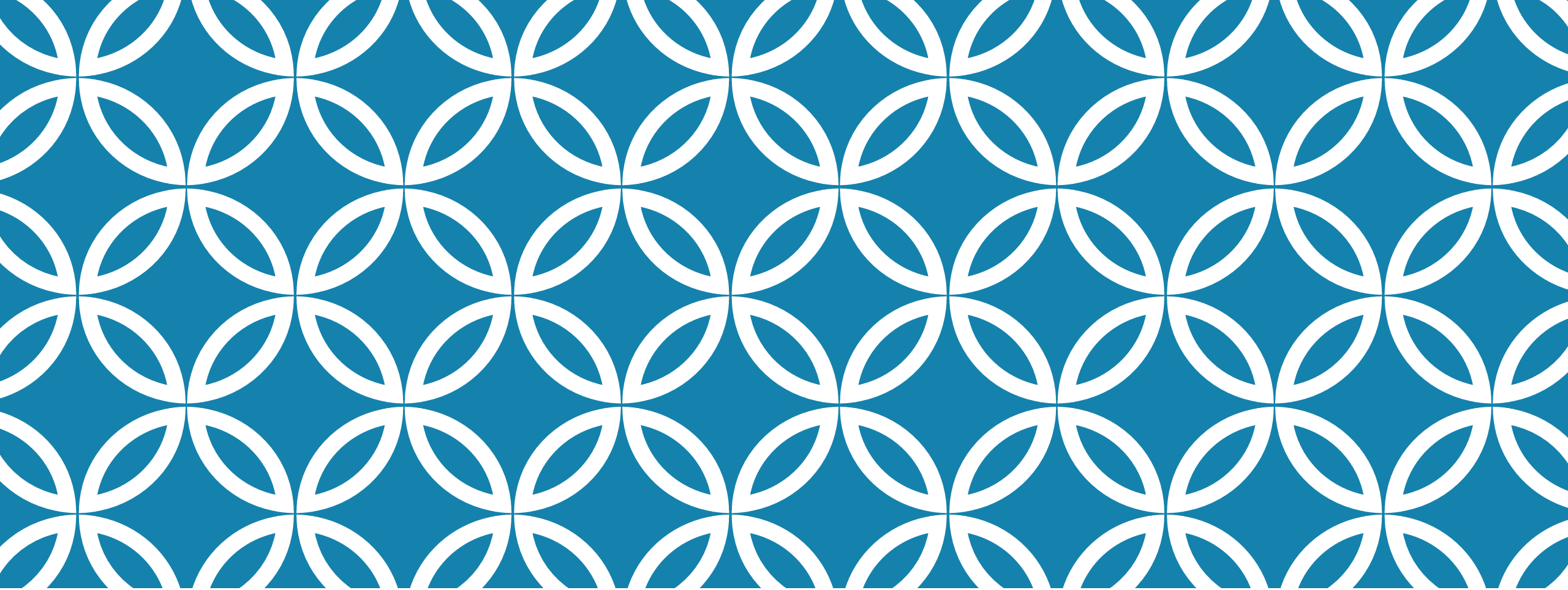
- **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
- **Task parallelism** – distributing threads across cores, each thread performing unique operation

Concurrency vs. Parallelism

Concurrent execution on single-core system:



Parallelism on a multi-core system:



OPENMP BASICS: SYNTAX, CONSTRUCTS, CLAUSES AND SECTIONS

Dr. Ayaz ul Hassan Khan
ayazhk@gmail.com

OPENMP 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.

Assumptions:

- **Participants know basic C/C++ programming**
- **Participants are new to parallel programming**
- **Participants have access to a compiler that supports OpenMP**

OPENMP OVERVIEW

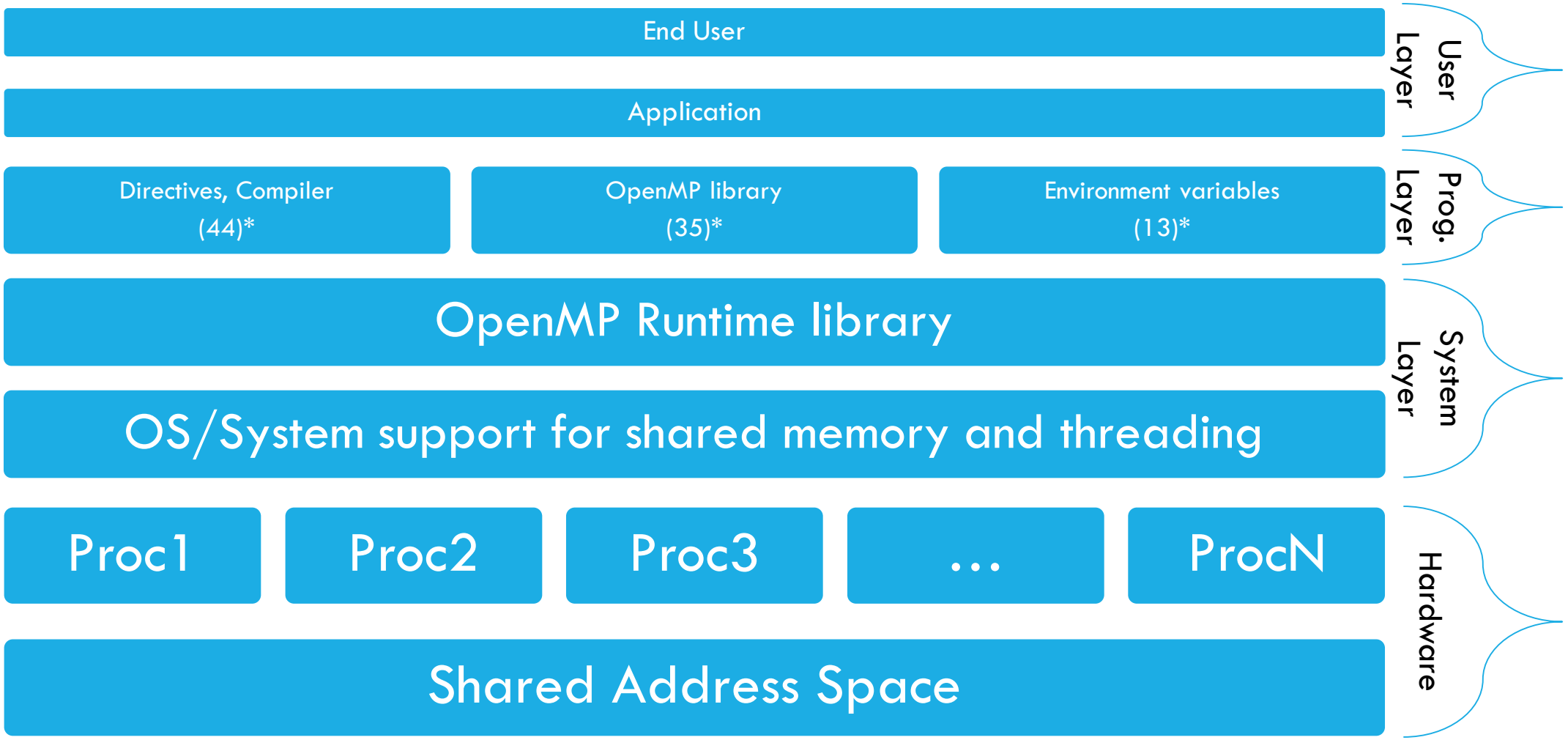
An API for writing multithreaded applications

A set of compiler directives and library routines for parallel application programmers

Greatly simplifies writing multi-threaded (MT) programs in Fortran, C, and C++

Standardizes last 28 years of SMP practice

OPENMP SOLUTION STACK



* As of version 4.0

OPENMP CORE SYNTAX

Most of the constructs in OpenMP are compiler directives.

#pragma omp *construct [clause [clause]…]*

Example

- ***#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.**
- **It's OK to have an exit() within the structured block.**

EXERCISE: HELLO WORLD

Write a multithreaded program where each thread prints “Hello World”

COMPILER DIRECTIVES

OpenMP compiler directives are used for various purposes:

- Spawning a parallel region
- Dividing blocks of code among threads
- Distributing loop iterations between threads
- Serializing sections of code
- Synchronization of work among threads

RUN-TIME LIBRARY ROUTINES:

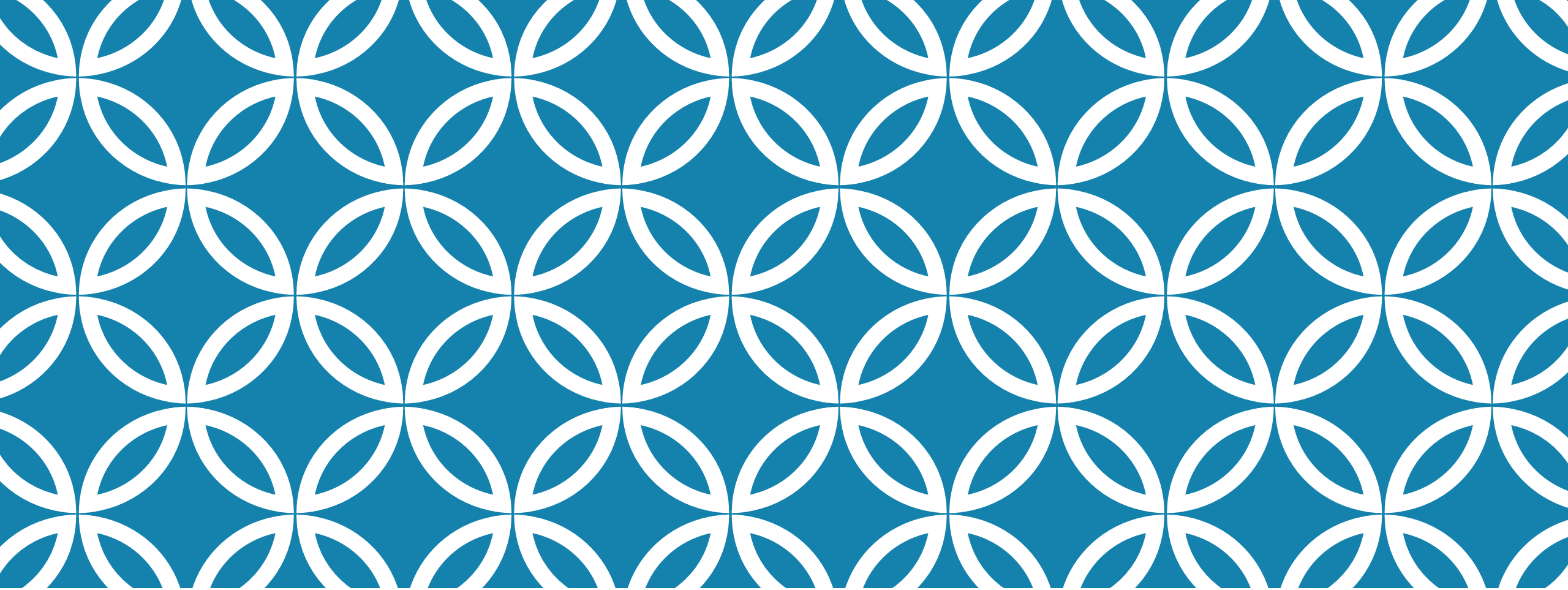
These routines are used for a variety of purposes:

- Setting and querying the number of threads
- Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
- Setting and querying the dynamic threads feature
- Querying if in a parallel region, and at what level
- Setting and querying nested parallelism
- Setting, initializing and terminating locks and nested locks
- Querying wall clock time and resolution

ENVIRONMENT VARIABLES:

These environment variables can be used to control such things as:

- Setting the number of threads
- Specifying how loop iterations are divided
- Binding threads to processors
- Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
- Enabling/disabling dynamic threads
- Setting thread stack size
- Setting thread wait policy



OPENMP DATA OFFLOADING SUPPORT

Dr. Ayaz ul Hassan Khan
ayazhk@gmail.com

DATA ENVIRONMENT: DEFAULT STORAGE ATTRIBUTES

Shared Memory programming model:

- Most variables are shared by default

Global variables are SHARED among threads

- Fortran: COMMON blocks, SAVE variables, MODULE variables
- C: File scope variables, static
- Both: dynamically allocated memory (ALLOCATE, malloc, new)

But not everything is shared...

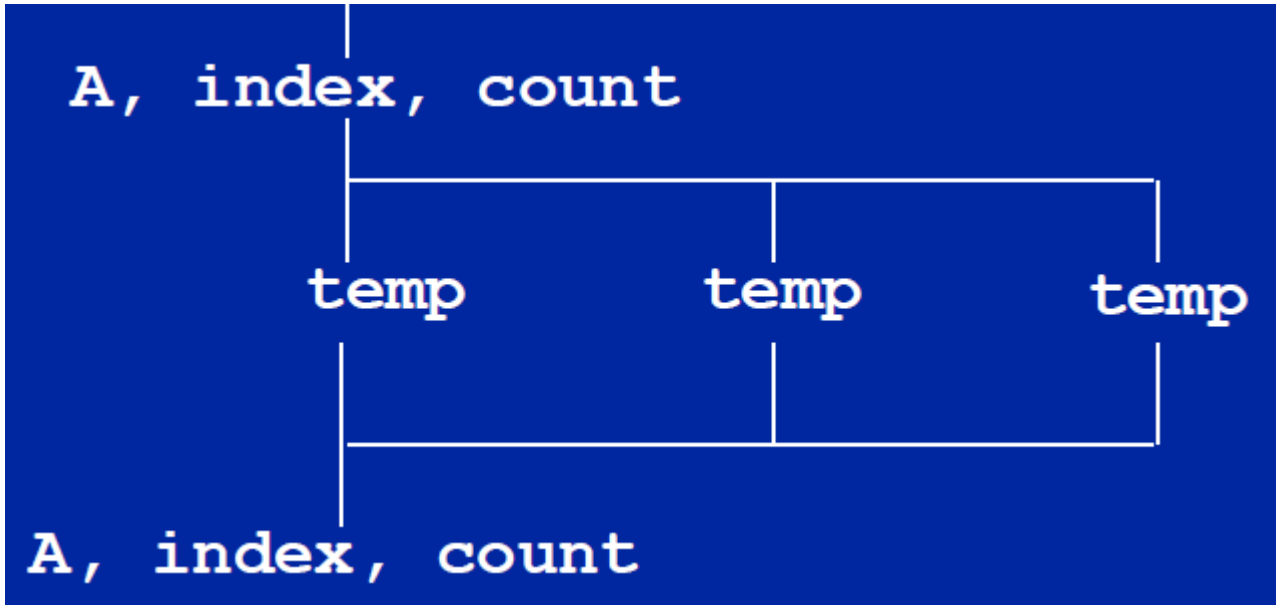
- Stack variables in subprograms(Fortran) or functions(C) called from parallel regions are PRIVATE
- Automatic variables within a statement block are PRIVATE.

DATA SHARING: EXAMPLE

```
double A[10];
int main() {
int index[10];
#pragma omp parallel
        work(index);
printf("%d\n", index[0]);;
}
```

```
extern double A[10];
void work(int *index) {
double temp[10];
static int count;
...
}
```

A, index and count are shared by all threads. temp is local to each thread



DATA SHARING: CHANGING STORAGE ATTRIBUTES

One can selectively change storage attributes for constructs using the following clauses*

- **SHARED**
- **PRIVATE**
- **FIRSTPRIVATE**

The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop with:

- **LASTPRIVATE**

The default attributes can be overridden with:

- **DEFAULT (PRIVATE | SHARED | NONE)**
  - **DEFAULT (PRIVATE) is Fortran only**

***All data clauses apply to parallel constructs and worksharing constructs except "shared" which only applies to parallel constructs.**

DATA SHARING: PRIVATE CLAUSE

private(var) creates a new local copy of var for each thread.

- **The value of the private copies is uninitialized**
- **The value of the original variable is unchanged after the region**

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

tmp was not initialized

tmp is 0 here

DATA SHARING: FIRSTPRIVATE CLAUSE

Variables initialized from shared variable

C++ objects are copy-constructed

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

Each thread gets its own copy of incr with an initial value of 0

DATA SHARING: LASTPRIVATE CLAUSE

Variables update shared variable using value from last iteration

C++ objects are updated as if by assignment

```
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;
}
```

"x" has the value it held for the "last sequential" iteration (i.e., for i=(n-1))

DATA SHARING: A DATA ENVIRONMENT TEST

Consider this example of PRIVATE and FIRSTPRIVATE

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

Are A,B,C local to each thread or shared inside the parallel region?

What are their initial values inside and values after the parallel region?

Inside this parallel region …

- **"A" is shared by all threads;; equals 1**
- **"B" and "C" are local to each thread.**
  - **B's initial value is undefined**
  - **C's initial value equals 1**

Following the parallel region …

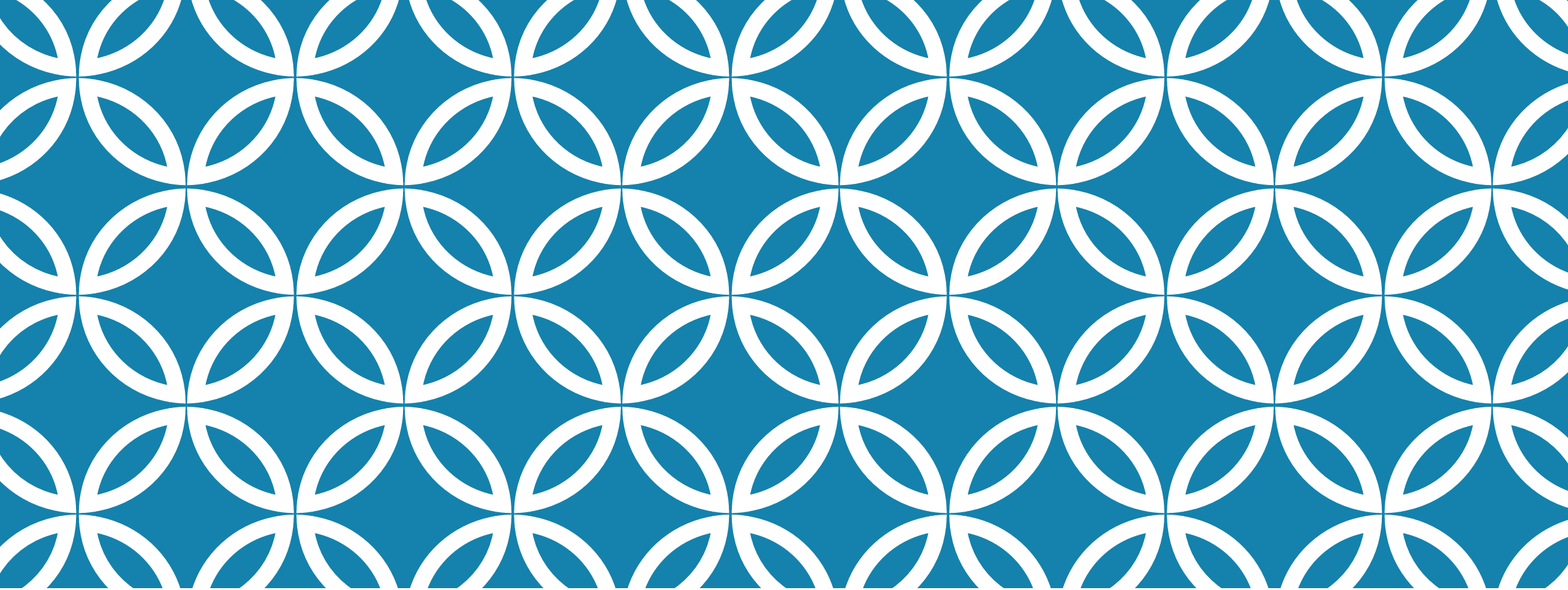
- **B and C revert to their original values of 1**
- **A is either 1 or the value it was set to inside the parallel region]**

DATA SHARING EXERCISE: MANDELBROT SET AREA

The supplied program (mandel.c) computes the area of a Mandelbrot set.

The program has been parallelized with OpenMP, but we were lazy and didn't do it right.

Find and fix the errors (hint … the problem is with the data environment).



OPENMP THREAD SAFETY AND SHARING

Dr. Ayaz ul Hassan Khan
ayazhk@gmail.com

OPENMP: HOW DO THREADS INTERACT?

OpenMP is a multi-threading, shared address model.

- **Threads communicate by sharing variables.**

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 so:

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

EXERCISE: PI PROGRAM

Create a parallel version of the pi program using a parallel construct.

Pay close attention to shared versus private variables.

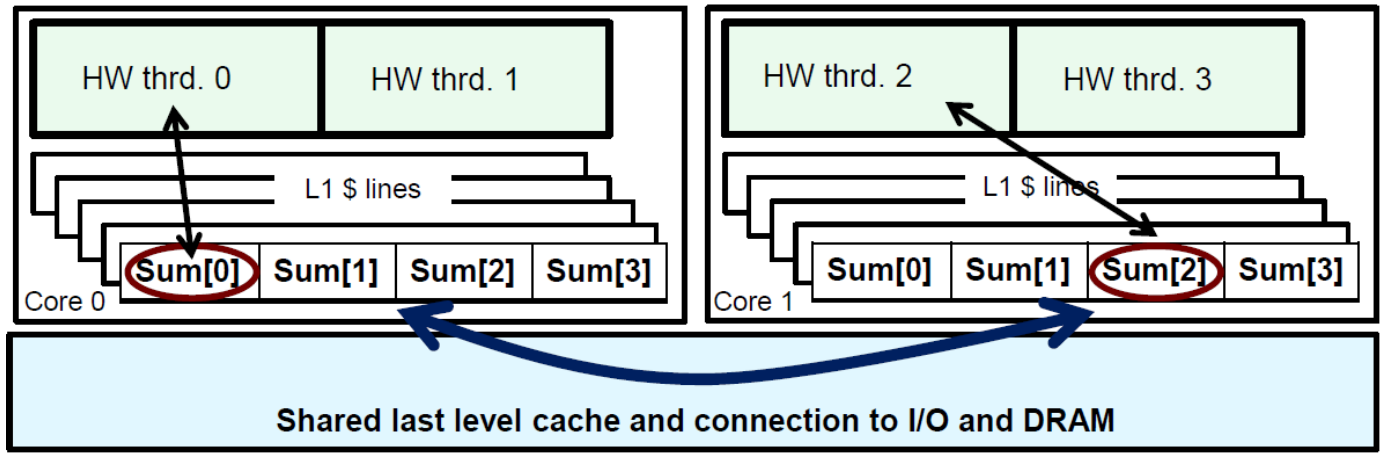
In addition to a parallel construct, you will need the runtime library routines

- **int omp_get_num_threads();**
- **int omp_get_thread_num();**
- **double omp_get_wtime();**

FALSE SHARING: POOR SCALING

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

- Results in poor scalability

Solution: Pad arrays so elements you use are on distinct cache lines.

EXERCISE: MODIFY PI PROGRAM TO ELIMINATE FALSE SHARING BY PADDING

EXERCISE: MODIFY PI PROGRAM TO ELIMINATE FALSE SHARING BY PADDING

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.

The two most common forms of synchronization are:

- **Barrier**: each thread wait at the barrier until all threads arrive.
- **Mutual exclusion**: Define a block of code that only one thread at a time can execute.

High Level Synchronization:

- Critical
- Atomic
- Barrier
- Ordered

Low Level Synchronization:

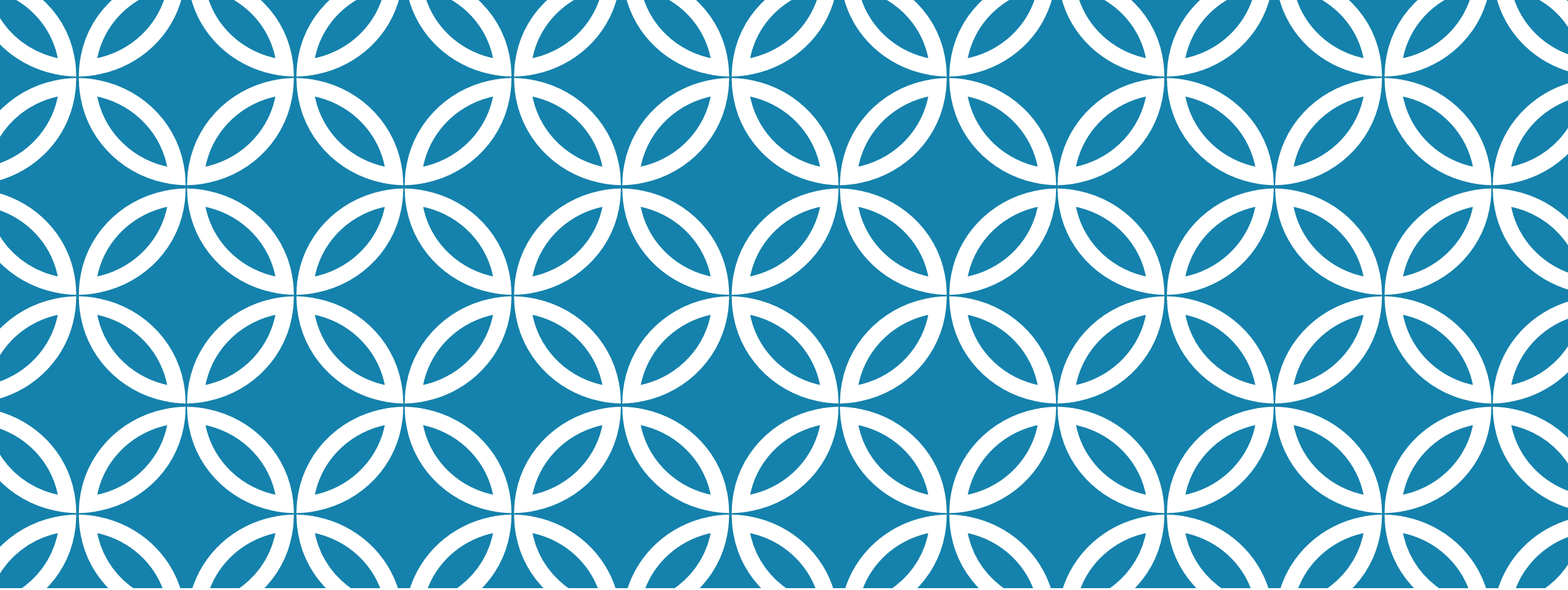
- Flush
- Locks (both simple and nested)

EXERCISE: USE CRITICAL SECTION TO REMOVE FALSE SHARING

In previous exercise, you probably used an array to create space for each thread to store its partial sum.

Modify your "pi program" from previous exercise to avoid false sharing due to the sum array.

EXERCISE: USE ATOMIC CLAUSE TO REMOVE FALSE SHARING



PREPARING THE DEVELOPMENT ENVIRONMENT

Dr. Ayaz ul Hassan Khan
ayazhk@gmail.com

INTEL HPC

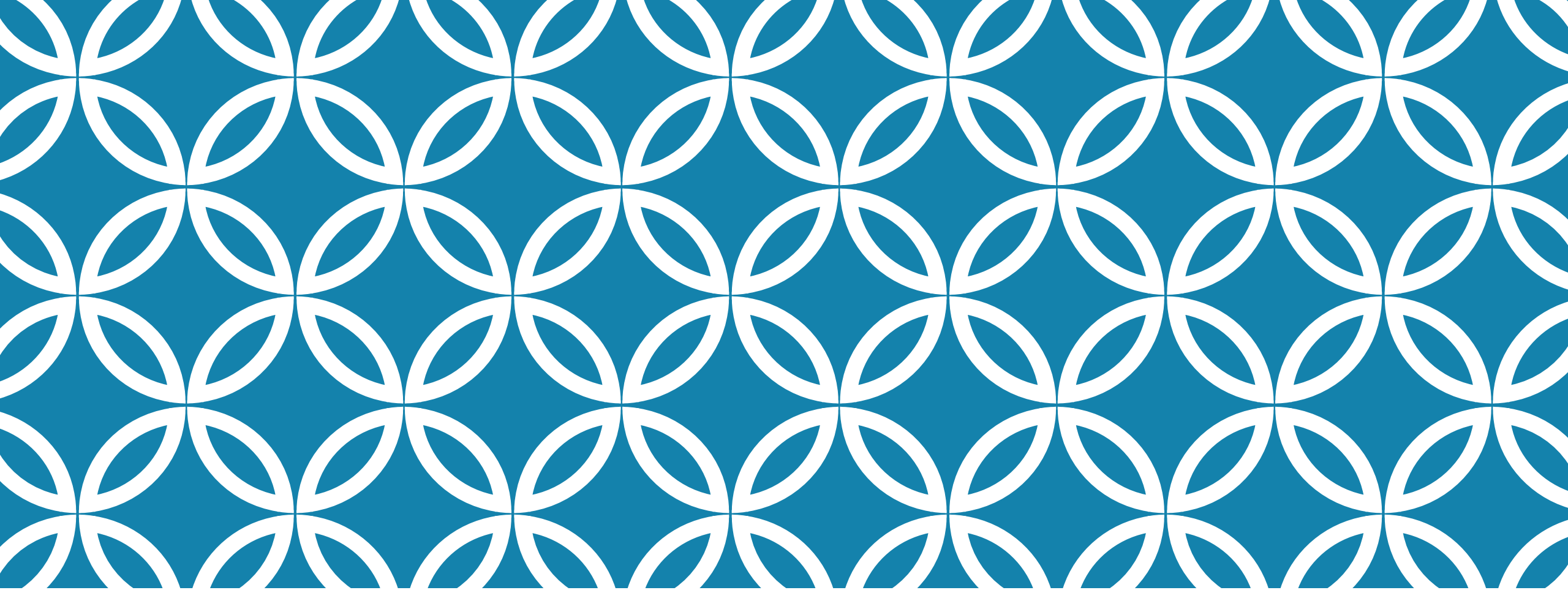
Intel HPC Toolkit:

https://www.intel.com/content/www/us/en/developer/tools/oneapi/hpc-toolkit-download.html

Intel DevCloud:

https://www.intel.com/content/www/us/en/developer/tools/devcloud/overview.html

DO REGISTER YOURSELF TO INTEL DEVCLOUD!



PARALLEL PROGRAMMING WITH OPENMP

Dr. Ayaz ul Hassan Khan
ayazhk@gmail.com

SPMD VS WORKSHARING

A parallel construct by itself creates an SPMD or “Single Program Multiple Data” program

- each thread redundantly executes the same code

How do you split up pathways through the code between threads within a team?

- This is called worksharing

THE LOOP WORKSHARING CONSTRUCTS

The loop worksharing construct splits up loop iterations among the threads in a team

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

The variable I is made "private" to each thread by default. You could do this explicitly with a "private(I)" clause

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];}

EXERCISE: PI WITH LOOPS

Go back to the serial pi program and parallelize it with a loop construct

Your goal is to minimize the number of changes made to the serial program.

REDUCTION

How do we handle this case?

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

We are combining values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can't be trivially removed

This is a very common situation … it is called a "reduction".

Support for reduction operations is included in most parallel programming environments.

REDUCTION

OpenMP reduction clause:

- reduction (op : list)

Inside a parallel or a work-sharing construct:

- 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.

The variables in “list” must be shared in the enclosing parallel region.

```
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 positive number
max	Most negative number

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

EXERCISE: PI WITH LOOP AND REDUCTION

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).

LOOP WORK-SHARING CONSTRUCTS: THE SCHEDULE CLAUSE

Schedule Clause	When To Use
STATIC	Pre-determined and predictable by the programmer
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead
AUTO	When the runtime can “learn” from previous executions of the same loop

EXERCISE: MATRIX MULTIPLICATION

Parallelize the matrix multiplication program in the file matmul.c

Can you optimize the program by playing with how the loops are scheduled?

NESTED LOOPS

For perfectly nested rectangular loops we can parallelize multiple loops in the nest with the collapse clause:

- #pragma omp parallel for collapse(2)

Will form a single loop of length NxM and then parallelize that.

Useful if N is O(no. of threads) so parallelizing the outer loop makes balancing the load difficult.

EXERCISE: MATRIX MULTIPLICATION WITH COLLAPSE CLAUSE

EXERCISE: LINKED LIST (HARD)

Consider the program linked.c

- Traverses a linked list computing a sequence of Fibonacci numbers at each node.

Parallelize this program using loop worksharing constructs

Once you have a correct program, optimize it.

OPENMP TASKS

Tasks are independent units of work.

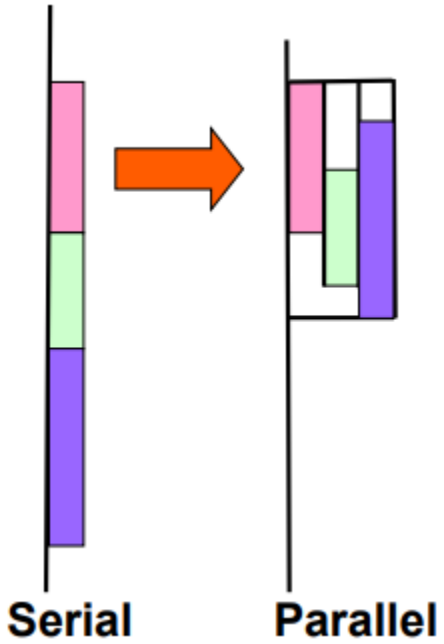
Tasks are composed of:

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

Threads perform the work of each task.

The runtime system decides when tasks are executed

- Tasks may be deferred
- Tasks may be executed immediately



WHEN ARE TASKS GUARANTEED TO COMPLETE

Tasks are guaranteed to be complete at thread barriers:

- #pragma omp barrier

or task barriers

- #pragma omp taskwait

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

Multiple foo tasks created here – one for each thread

All foo tasks guaranteed to be completed here

One bar task created here

bar task guaranteed to be completed here

DATA SCOPING WITH TASKS: FIBONACCI EXAMPLE

n is private in both tasks

x,y are private variables in each task

A task's private variables are undefined outside the task

What's wrong here?

```
int fib ( int n )
{
        int x,y;
        if ( n < 2 ) return n;
        #pragma omp task
        x = fib(n-1);
        #pragma omp task
        y = fib(n-2);
        #pragma omp
taskwait
        return x+y
}
```

DATA SCOPING WITH TASKS: FIBONACCI EXAMPLE - SOLUTION

n is private in both tasks

x,y are shared variables

- we need both values to compute the sum

```
int fib ( int n )
{
        int x,y;
        if ( n < 2 ) return n;
        #pragma omp task
shared(x)
        x = fib(n-1);
        #pragma omp task
shared(y)
        y = fib(n-2);
        #pragma omp taskwait
        return x+y
}
```

EXERCISE: TASKS IN OPENMP

Consider the program linked.c

- Traverses a linked list computing a sequence of Fibonacci numbers at each node.

Parallelize this program using tasks.

Compare your solution's complexity to an approach without tasks.

TASK DEPENDENCES

Task dependences as a way to define task-execution constraints

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
        #pragma
        std::cout<

        #pragma

        #pragma omp   task
        x++;
}
```

Task dependences can help us to remove “strong” synchronizations, increasing the look ahead and, frequently, the parallelism!!!!

```
int x = 0;
#pragma omp parallel
#pragma omp single

        ragma omp   task depend(in: x)
        :cout<< x << std::endl;

                #pragma omp   task depend(inout: x)
                x++;
}
```

	Without Depend Clause				With Depend Clause		
T1	C	E			C	E	
T2			C	E		C	E

DEPEND CLAUSE

A task cannot be executed until all its predecessor tasks are completed

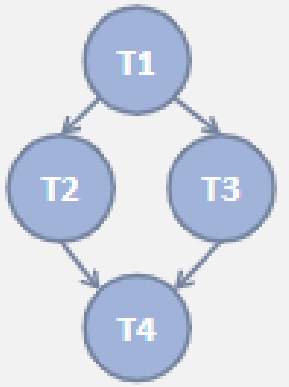
If a task defines an *in* dependence over a list-item

- the task will depend on all previously generated sibling tasks that reference that list-item in an *out* or *inout* dependence

If a task defines an *out/inout* dependence over list-item

- the task will depend on all previously generated sibling tasks that reference that list-item in an *in*, *out* or *inout* dependence

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x) //T1
  { ... }

  #pragma omp task depend(in: x)    //T2
  { ... }

  #pragma omp task depend(in: x)    //T3
  { ... }

  #pragma omp task depend(inout: x) //T4
  { ... }
}
```

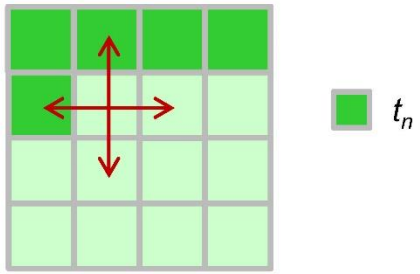


Use Case: intro to Gauss-Seidel

```
void serial_gauss_seidel(int tsteps, int size, int (*p)[size]) {
  for (int t = 0; t < tsteps; ++t) {
    for (int i = 1; i < size-1; ++i) {
      for (int j = 1; j < size-1; ++j) {
        p[i][j] = 0.25 * (p[i][j-1] * // left
                          p[i][j+1] * // right
                          p[i-1][j] * // top
                          p[i+1][j]); // bottom
      }
    }
  }
}
```

Access pattern analysis

For a specific t, i and j

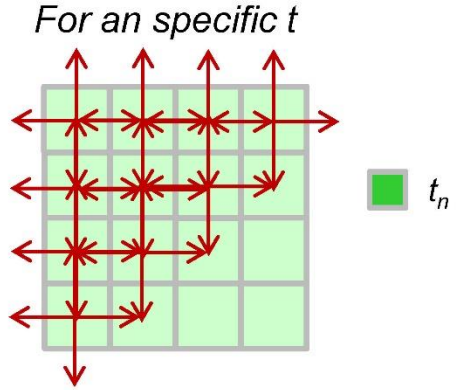


Each cell depends on:
- two cells (north & west) that are computed in the current time step, and
- two cells (south & east) that were computed in the previous time step

Use Case: Gauss-Seidel (2)

```
void serial_gauss_seidel(int tsteps, int size, int (*p)[size]) {
  for (int t = 0; t < tsteps; ++t) {
    for (int i = 1; i < size-1; ++i) {
      for (int j = 1; j < size-1; ++j) {
        p[i][j] = 0.25 * (p[i][j-1] * // left
                          p[i][j+1] * // right
                          p[i-1][j] * // top
                          p[i+1][j]); // bottom
      }
    }
  }
}
```

1st parallelization strategy



We can exploit the wavefront to obtain parallelism!!

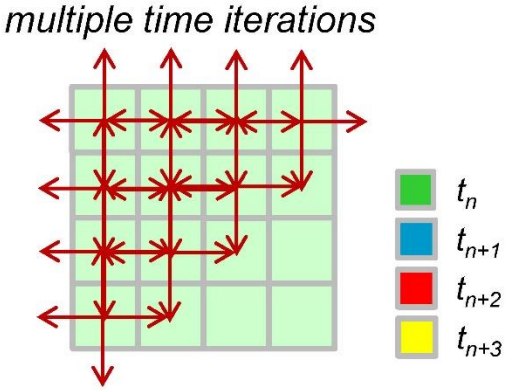
Use Case: Gauss-Seidel (3)

```
void gauss_seidel(int tsteps, int size, int TS, int (*p)[size]) {
  int NB = size / TS;
  #pragma omp parallel
  for (int t = 0; t < tsteps; ++t) {
    // First NB diagonals
    for (int diag = 0; diag < NB; ++diag) {
      #pragma omp for
      for (int d = 0; d <= diag; ++d) {
        int ii = d;
        int jj = diag - d;
         for (int i = 1+ii*TS; i < ((ii+1)*TS); ++i)
           for (int j = 1+jj*TS; i < ((jj+1)*TS); ++j)
             p[i][j] = 0.25 * (p[i][j-1] * p[i][j+1] *
                               p[i-1][j] * p[i+1][j]);
      }
    }
    // Lasts NB diagonals
    for (int diag = NB-1; diag >= 0; --diag) {
      // Similar code to the previous loop
    }
  }
}
```

Use Case: Gauss-Seidel (4)

```
void serial_gauss_seidel(int tsteps, int size, int (*p)[size]) {
  for (int t = 0; t < tsteps; ++t) {
    for (int i = 1; i < size-1; ++i) {
      for (int j = 1; j < size-1; ++j) {
        p[i][j] = 0.25 * (p[i][j-1] * // left
                          p[i][j+1] * // right
                          p[i-1][j] * // top
                          p[i+1][j]); // bottom
      }
    }
  }
}
```

2nd parallelization strategy

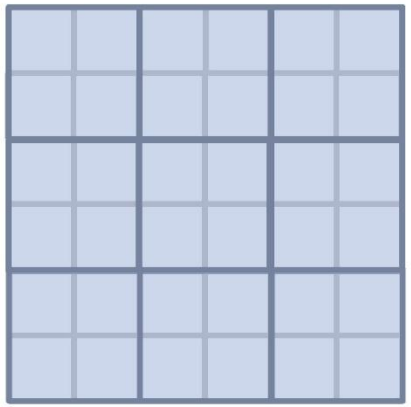


We can exploit the wavefront of multiple time steps to obtain MORE parallelism!!

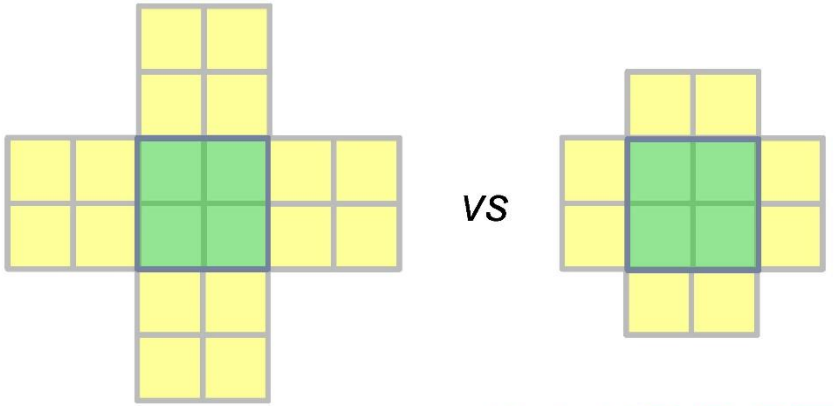
Use Case: Gauss-Seidel (5)

```
void gauss_seidel(int tsteps, int size, int TS, int (*p)[size]) {
  int NB = size / TS;

  #pragma omp parallel
  #pragma omp single
  for (int t = 0; t < tsteps; ++t)
    for (int ii=1; ii < size-1; ii+=TS)
      for (int jj=1; jj < size-1; jj+=TS) {
        #pragma omp task depend(inout: p[ii:TS][jj:TS])
            depend(in: p[ii-TS:TS][jj:TS], p[ii+TS:TS][jj:TS],
                       p[ii:TS][jj-TS:TS], p[ii:TS][jj:TS])
        {
          for (int i=ii; i<(1+ii)*TS; ++i)
            for (int j=jj; j<(1+jj)*TS; ++j)
              p[i][j] = 0.25 * (p[i][j-1] * p[i][j+1] *
                                p[i-1][j] * p[i+1][j]);
        }
      }
}
```

inner matrix region

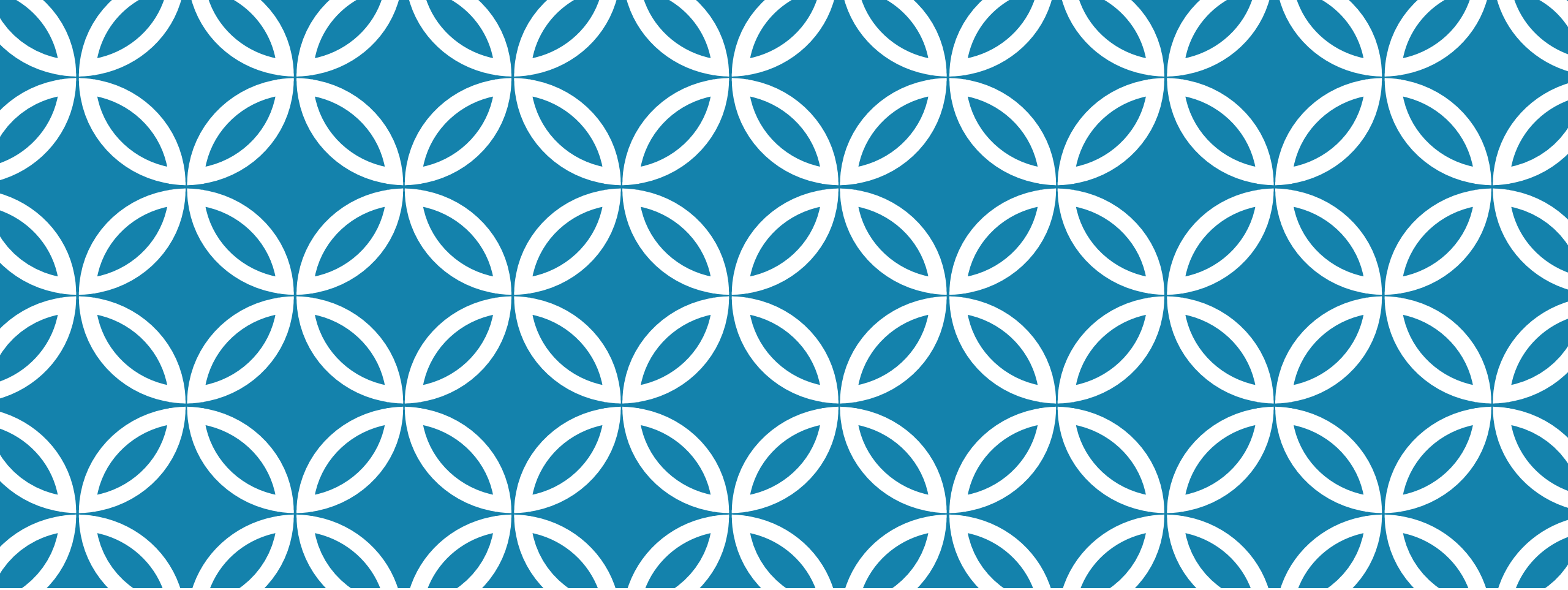


Q: Why do the input dependences depend on the whole block rather than just a column/row?



The OpenMP Common Core: Most OpenMP programs only use these 21 items

OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads.
void omp_set_thread_num() int omp_get_thread_num() int omp_get_num_threads()	Default number of threads and internal control variables. SPMD pattern: Create threads with a parallel region and split up the work using the number of threads and the thread ID.
double omp_get_wtime()	Speedup and Amdahl's law. False sharing and other performance issues.
setenv OMP_NUM_THREADS N	Setting the internal control variable for 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 (static [,chunk]) schedule(dynamic [,chunk])	Loop schedules, loop overheads, and load balance.
shared(list), private(list), firstprivate(list)	Data environment.
default(none)	Force explicit definition of each variable's storage attribute
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.



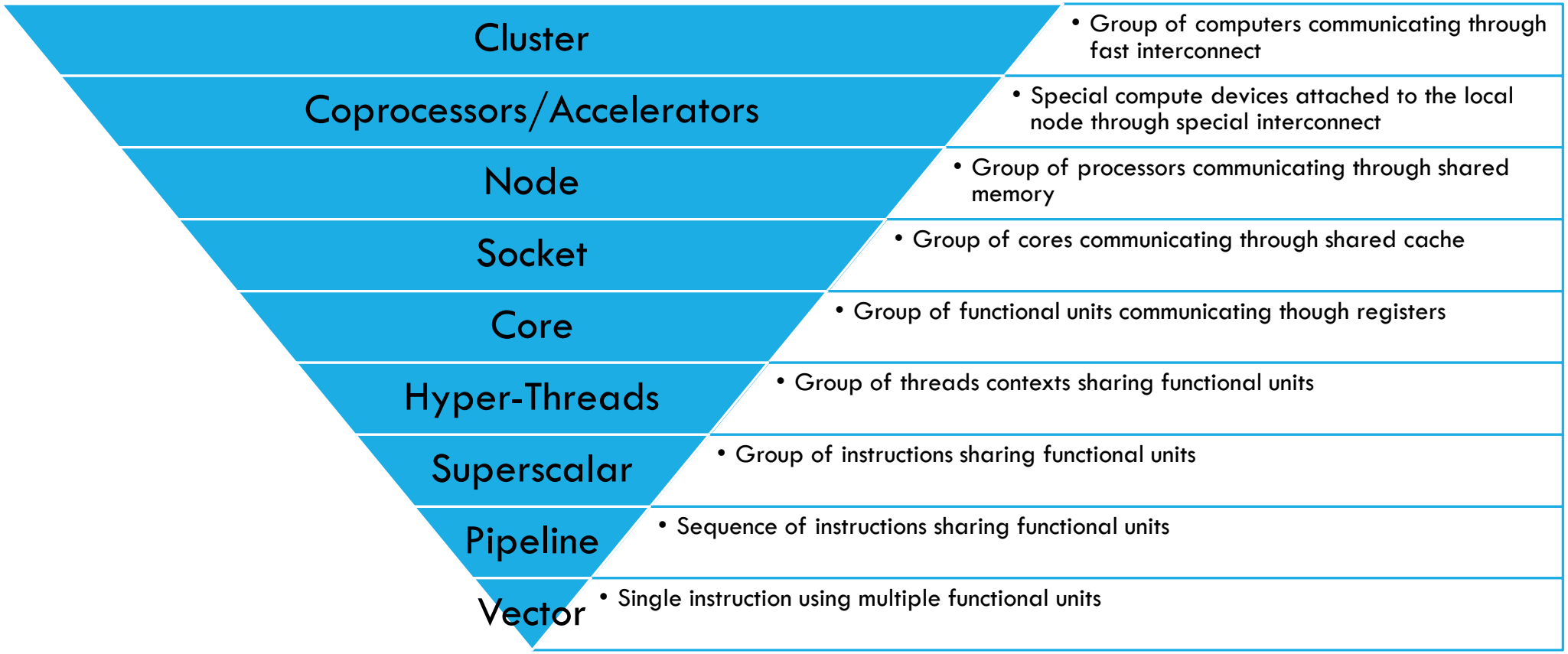
VECTORIZATION AND SIMD EXTENSIONS

Dr. Ayaz ul Hassan Khan
ayazhk@gmail.com

EVOLUTION OF INTEL HARDWARE

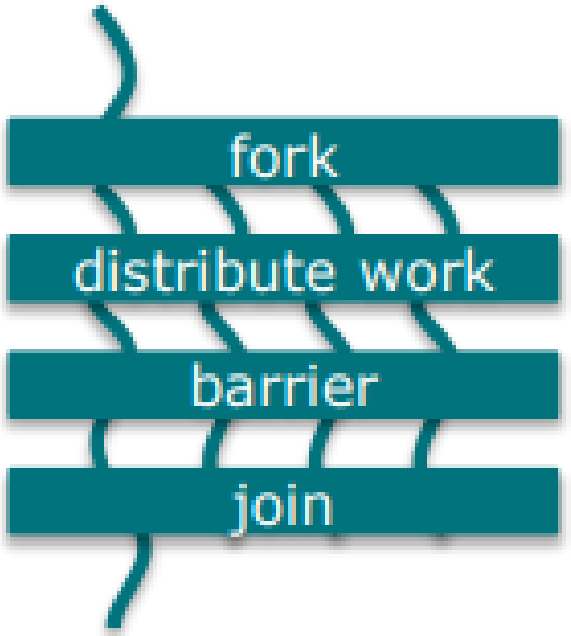
	64-bitIntel® Xeon® processor	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor E5-2600v3 series	Intel® Xeon® Scalable Processor
Frequency	3.6 GHz	3.0 GHz	3.2 GHz	3.3 GHz	2.3 GHz	2.5 GHz
Core(s)	1	2	4	6	18	28
Thread(s)	2	2	8	12	36	56
SIMD Width	128 (2 clocks)	128 (1 clock)	128 (1 clock)	128 (1 clock)	256 (1 clock)	512 (1 clock)

LEVELS OF PARALLELISM

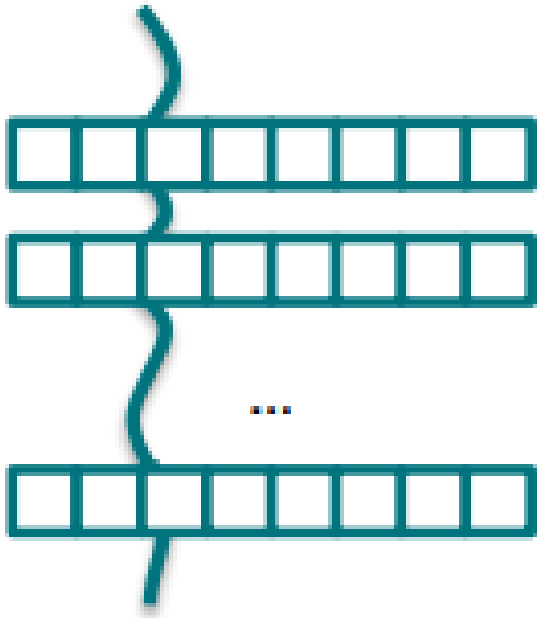


WORKSHARING VS SIMD

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




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



SIMD LOOP CONSTRUCT

Vectorize a loop nest

- Cut loop into chunks that fit a SIMD vector register
- No parallelization of the loop body

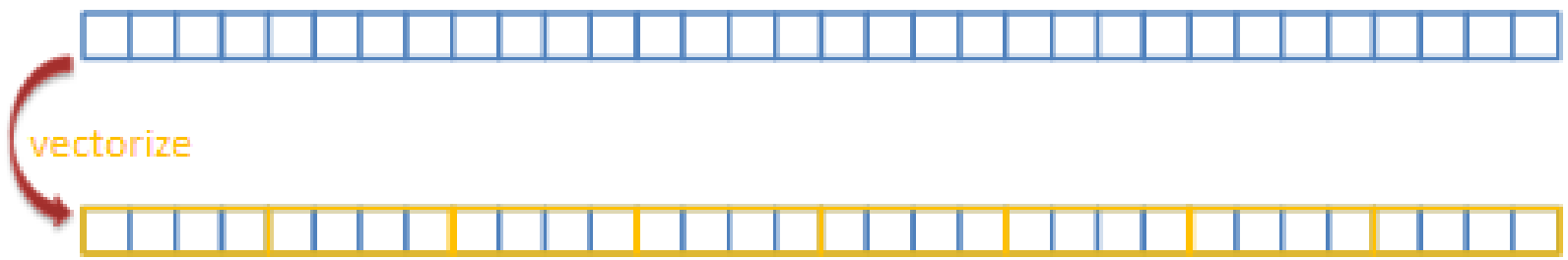
Syntax

#pragma omp simd[clause[[,] clause],...]

- for-loops

EXAMPLE: DOT PRODUCT

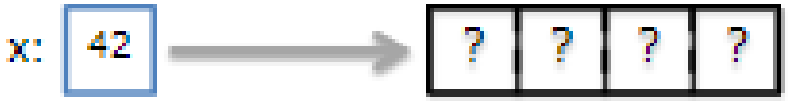
```
float sprod(float *a, float *b, int n) {
        float sum = 0.0f;
        #pragma omp simd reduction(+:sum)
        for (int k=0; k<n; k++)
                sum += a[k] * b[k];
        return sum;
}
```



DATA SHARING CLAUSES

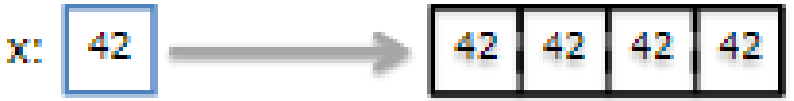
private(var-list):

- Uninitialized vectors for variables in var-list



firstprivate(var-list):

- Initialized vectors for variables in var-list



reduction(op:var-list):

- Create private variables for var-list and apply reduction operator op at the end of the construct

SIMD WORKSHARING CONSTRUCT

Parallelize and vectorize a loop nest

- Distribute a loop's iteration space across a thread team
- Subdivide loop chunks to fit a SIMD vector register

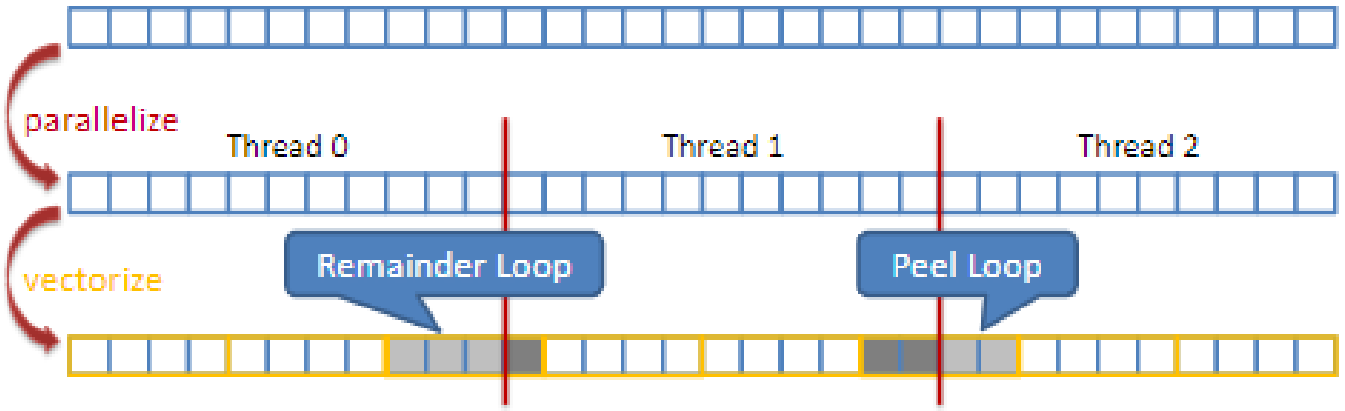
Syntax

#pragma omp for simd[clause[[,] clause],...]

- for-loops

EXAMPLE: DOT PRODUCT

```
float sprod(float *a, float *b, int n) {
        float sum = 0.0f;
        #pragma omp for simd reduction(+:sum)
        for (int k=0; k<n; k++)
                sum += a[k] * b[k];
        return sum;
}
```



SIMD THREAD SCHEDULING

```
float sprod(float *a, float *b, intn) {
        float sum = 0.0f;
        #pragma omp for simd reduction(+:sum) schedule(static, 5)
        for (int k=0; k<n; k++) sum += a[k] * b[k];
        return sum;
}
```

You should choose chunk sizes that are multiples of the SIMD length

- Remainder loops are not triggered
- Likely better performance

SIMD CHUNKS: OPENMP 4.5 OR ABOVE

```
float sprod(float *a, float *b, int n) {
        float sum = 0.0f;
        #pragma omp for simd reduction(+:sum) schedule(simd: static, 5)
        for (int k=0; k<n; k++)
                sum += a[k] * b[k];
        return sum;
}
```

Chooses chunk sizes that are multiples of the SIMD length

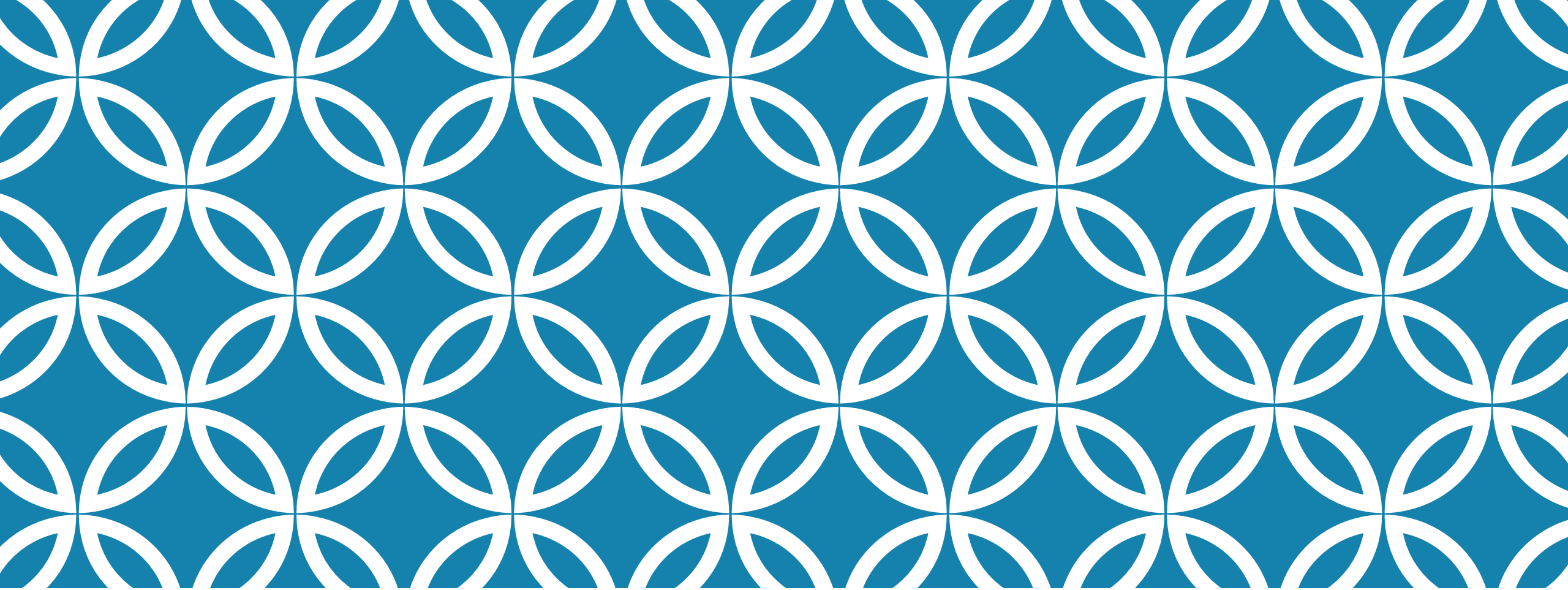
- First and last chunk may be slightly different to fix alignment and to handle loops that are not exact multiples of SIMD width
- Remainder loops are not triggered
- Likely better performance

SIMD FUNCTION VECTORIZATION

```
float min(float a, float b) {
        return a < b ? a : b;
}

float distsq(float x, float y) {
        return (x -y) * (x - y);
}

void example() {
        #pragma omp  parallel for simd
        for (i=0; i<N; i++) {
                d[i] = min(distsq(a[i], b[i]), c[i]
        }
}
```

```
#pragma omp declare simd
float min(float a, float b) {
        return a < b ? a : b;
}

#pragma omp declare simd
float distsq(float x, float y) {
        return (x -y) * (x - y);
}

void example() {
        #pragma omp  parallel for simd
        for (i=0; i<N; i++) {
                d[i] = min(distsq(a[i], b[i]), c[i]);
        }
}
```

EXERCISE: MANDELBROT COMPLETE APPLICATION



USING OPENMP WITH C/C++: PROFILING

Dr. Ayaz ul Hassan Khan
ayazhk@gmail.com

ONLINE TUTORIALS

https://www.intel.com/content/www/us/en/develop/documentation/vtune-tutorial-common-bottlenecks-linux/top.html

VTune Tutorial on Intel DevCloud (to be done later)