COSC 407
Intro to Parallel Computing

Topic 10 - OpenMP Examples, Models, SIMD

COSC 407: Intro to Parallel Computing

1

Outline (Asynchronous)

Previously:

- Sections
- Scheduling Loops (static, dynamic, guided, auto)
- Ordered Iterations
- Some examples

Today

- Matrix multiplication
- Max reduction
- Some More Examples
- Asides (Producer/Consumer) and Comments on OpenMP

Matrix Multiplication

Write a C program that uses OpenMP to perform parallel matrix multiplication. The code should do the following steps:

1. create three matrices A, B, and C and initializes A and B to some values of your choice
   (e.g., a[i][j]=i+j and b[i][j]=i*j).
2. perform matrix multiplication.
3. print out the resultant matrix C.

Note:

if we have two matrices: $\mathbf{A} = \begin{pmatrix} a & b & c \\ x & y & z \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} \alpha & \rho \\ \beta & \sigma \\ \gamma & \tau \end{pmatrix},$

Then their matrix product is

$$\mathbf{AB} = \begin{pmatrix} a & b & c \\ x & y & z \end{pmatrix} \begin{pmatrix} \alpha & \rho \\ \beta & \sigma \\ \gamma & \tau \end{pmatrix} = \begin{pmatrix} a\alpha + b\beta + c\gamma & a\rho + b\sigma + c\tau \\ x\alpha + y\beta + z\gamma & x\rho + y\sigma + z\tau \end{pmatrix},$$

Serial Version

```
int i, j, k; double a[NRA][NCA], b[NCA][NCB], double c[NRA][NCB];
// Initialize matrices
for (i = 0; i < NRA; i++)
     for (j = 0; j < NCA; j++)
          a[i][j] = i + j;
for (i = 0; i < NCA; i++)
     for (j = 0; j < NCB; j++)
          b[i][j] = i * j;
for (i = 0; i < NRA; i++)
     for (j = 0; j < NCB; j++)
          c[i][j] = 0;
// matrix multiplication
printf("Starting matrix multiplication...\n");
for (i = 0; i < NRA; i++)
     for (j = 0; j < NCB; j++)
          for (k = 0; k < NCA; k++)
               c[i][j] += a[i][k] * b[k][j];

// Print results
printf("Result Matrix:\n");
for (i = 0; i < NRA; i++) {
     for (j = 0; j < NCB; j++)
          printf("%6.2f   ", c[i][j]);
     printf("\n");
}
```

Q1: which loops would you parallelize?
Q2: Is there any loop-carried dependencies?

Finding the Max Value

Following is a function that finds and returns the max integer in a given array

```
int max_serial(int *arr){
     int max = arr[0];
     int i;
     for (i = 0; i < N; i++)
             if (max < arr[i] )
                     max = arr[i];
     return max;
}
```

Assume N is a constant = size of the array

Write an OpenMP code to parallelize the above function.

5

Finding the Max Value

V.1

Will this code produce the required output?

```
int max_parallel(int *arr){
     int max = arr[0];
     int i;
     #pragma omp parallel for
     for (i = 0; i < N; i++)
         if (max < arr[i])
               max = arr[i];
     return max;
}
```


6

Finding the Max Value

v.2

How about this Solution?

```
int max_parallel(int *arr){
     int max = arr[0];
     int i;
     #pragma omp parallel for
     for (i = 0; i < N; i++)
         #pragma omp critical
         if (max < arr[i])
              max = arr[i];
     return max;
}
```


Finding the Max Value

v.3

What do you think now?

```
int max_parallel(int *arr, int thread_count){
   int shared_max = arr[0], i;
   #pragma omp parallel num_threads(thread_count)
   {     int local_max = arr[0];   //each thread gets a copy of local_max

        #pragma omp for
        for (i = 0; i < N; i++)   //each thread finds it local max
           if (local_max < arr[i])
                local_max = arr[i];
        #pragma omp critical      //one thread at a time, i.e. sequential
        if(shared_max < local_max)
           shared_max = local_max;
    }
    return shared_max;
}
```

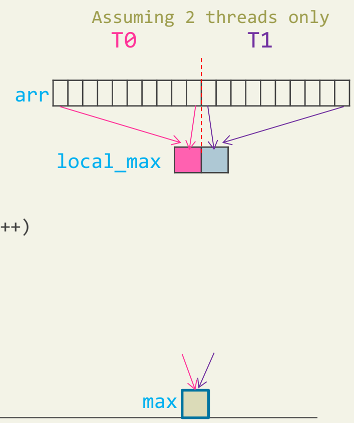
Only a "good" solution – there is a better way using reduction! This will be discussed shortly (not in the next slide though)

Finding the Max Value

v.4

Will this code produce the required output?

```
int max_parallel(int *arr, int thread_count){
        int local_max[thread_count];
        #pragma omp parallel num_threads(thread_count)
        {  int tid = omp_get_thread_num();
           local_max[tid] = arr[0];
           #pragma omp for
           for (int i = 0; i < N; i++)
              if (local_max[tid]<arr[i])
                   local_max[tid] = arr[i];
        }
        //this part is sequential
        int max = local_max[0];
        for (int i = 0 ; i < thread_count; i++)
           if (local_max[i] > max)
                 max = local_max[i];
        return max;
}
```



Max / Min Reduction

- First introduced in OpenMP v3.1
- Remember:

  Reduction Syntax:

  ```
  reduction (<op> : <variable list>)
  ```

  1. Provides a private copy of the variable for each thread
  2. Combines the private results on exit

Finding the Max Value

v.5

Now, what do you think?

```
int max_parallel(int *arr){
        int i, m = arr[0];
        #pragma omp parallel for reduction(max:m)
        for (i = 0; i < N; i++)
                if (m < arr[i])
                        m = arr[i];
        return m;
}
```


Finding the Min, Max and SUM

A function that returns 3 values: max, min, and sum in an array using OpenMP

```
void stats(int* arr, int* h, int* l, int* s) {
 int i, high = arr[0], low = arr[0], sum = 0;
 #pragma omp parallel for reduction(max:high) reduction(min:low) \
   reduction(+:sum)
 for (i = 0; i < N; i++){
   if(high<arr[i]) high = arr[i];
   if(low>arr[i]) low = arr[i];
   sum += arr[i];
 }

 *h = high; //returning three values
 *l = low;
 *s = sum;
}
```

Nested Loops: `collapse` clause

- Collapse directive turns nested loops into ONE big loop
- The iteration space is then divided according to the **schedule** clause.

Syntax

#pragma omp for **collapse(n)**

Where n is the number of loops to be collapsed

Example: `collapse`

```
#pragma omp parallel for private(j) num_threads(4)
for(i=0; i<2; i++)
   for(j=0; j<4; j++)
      printf("%d,%d ", i, j);
```

o/p: $\underbrace{\texttt{0,0 \quad 0,1 \quad 0,2 \quad 0,3}}_{T0}$ $\underbrace{\texttt{1,1 \quad 1,2 \quad 1,3 \quad 1,4}}_{T1}$

```
#pragma omp parallel for collapse(2) num_threads(4)
for(i=0; i<2; i++)
   for(j=0; j<4; j++)
      printf("T%d,i:%d,j:%d\n",omp_get_thread_num(), i, j);
```

o/p: $\underbrace{\texttt{0,0 \quad 0,1}}_{T0}$ $\underbrace{\texttt{0,2 \quad 0,3}}_{T1}$ $\underbrace{\texttt{1,1 \quad 1,2}}_{T2}$ $\underbrace{\texttt{1,3 \quad 1,4}}_{T3}$

Nested Loops: `collapse` clause

Rules:

- The specified number of loops must be indicated
- The loops must be perfectly nested; i.e., no code between the loops which are collapsed.
- All counters in the collapsed loops are private (no need to add the private clause for them)

Note: collapse doesn't always improve the performance – a smart compiler will automatically collapse nested loops IF needed.

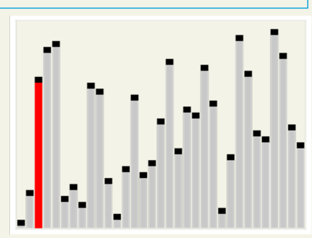
Bubble Sort

- Starting from the first item, compare adjacent items and keep "bubbling" the larger one to the right. Repeat for remaining sublist.

```
for(k = list_length, k >= 2; k--)
   for(i = 0; i < k; i++)
      if(a[i-1] > a[i]) swap(&a[i-1],&a[i]);
```

What is the complexity?

- Bubble sort cannot be easily parallelized!
- **Why?** (i.e. identify the loop carried dependency for both for-loops)



Odd-Even Transposition Sort

- A sequence of phases (`N` is number of elements in list `a`).

```
for(phase = 0, phase < N; phase++)
   if(phase % 2 == 0) //even phases
      for(i = 1; i < N; i += 2)
         if(a[i-1] > a[i]) swap(&a[i-1],&a[i]);
   else                  //odd phases
      for(i = 1; i < N-1; i += 2)
         if(a[i] > a[i+1]) swap(&a[i],&a[i+1]);
```

- Similar to bubble sort, but easier to parallelize
  - Outer-loop **has** loop-carried dependence
  - Inner loop **doesn't have** loop-carried dependence

	Subscript in Array						
Phase	0		1		2		3
0	9	↔	7		8	↔	6
	7		9		6		8
1	7		9	↔	6		8
	7		6		9		8
2	7	↔	6		9	↔	8
	6		7		8		9
3	6		7	↔	8		9
	6		7		8		9

Odd-Even Transposition Sort

```
for(phase = 0, phase < N; phase++)
   if(phase % 2 == 0) //even phases
      for(i = 1; i < N; i += 2)
         if(a[i-1] > a[i]) swap(&a[i-1],&a[i]);
   else                  //odd phases
      for(i = 1; i < N-1; i += 2)
         if(a[i] > a[i+1]) swap(&a[i],&a[i+1]);
```

	Subscript in Array						
Phase	0		1		2		3
0	9	↔ T_n	7		8	↔ T_m	6
	7		9		6		8
1	7		9	↔ T_n	6		8
	7		6		9		8
2	7	↔ T_n	6		9	↔ T_m	8
	6		7		8		9
3	6		7	↔ T_n	8		9
	6		7		8		9

No dependency between thread, thus no race condition

Odd-Even Sort - Attempt 1

Q: Do we need any explicit barriers in this code?

```
for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
          # pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
           for (i = 1; i < n; i += 2){
               if (a[i-1] > a[i]) {
                     tmp = a[i-1];
                     a[i-1] = a[i];
                     a[i] = tmp;
               }
           }
      else
          # pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
           for (i = 1; i < n-1; i += 2)
                if (a[i] > a[i+1]) {
                     tmp = a[i+1];
                     a[i+1] = a[i];
                     a[i] = tmp;
               }
}
```

fork

join (implicit)

fork

join (implicit)

Odd-Even Sort - Attempt 2

```
# pragma omp parallel num_threads(thread_count) \
  default(none) shared(a, n) private(i, tmp, phase)
 for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
          # pragma omp for
           for (i = 1; i < n; i += 2){
               if (a[i-1] > a[i]) {
                     tmp = a[i-1];
                     a[i-1] = a[i];
                     a[i] = tmp;
               }
           }
      else
          # pragma omp for
           for (i = 1; i < n-1; i += 2)
                if (a[i] > a[i+1]) {
                     tmp = a[i+1];
                     a[i+1] = a[i];
                     a[i] = tmp;
               }
}
```

fork

Reuse thread

tells OpenMP to parallelize the for loop with the existing team of threads.

Reuse thread

join (implicit)

Comparison of Performance

- Odd-even sort with two parallel for directives and two for directives.

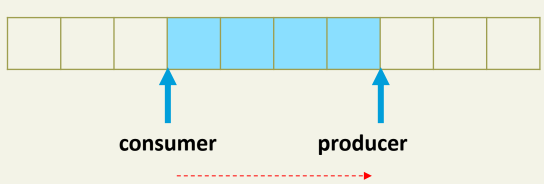
(Times are in seconds, size of array is 20,000)

Thread count	1	2	3	4
Two `parallel for` directives	0.770	0.453	0.358	0.305
Two `for` directives	0.732	0.376	0.294	0.239

Producer - Consumer Model

This model has two (or more) threads/processes work together on a shared resource (a **buffer**).

- Producer writes data to the buffer **when it is not full**
- Consumer reads data from buffer **when it is not empty**



***Circular buffers** are actually linear buffers in which the producer or consumer moves to the beginning whenever it reaches the end of the buffer.*

Condition Synchronization

- ***Condition synchronization*** is a mechanism to ensure that a process is blocked if some condition is not satisfied

Example:

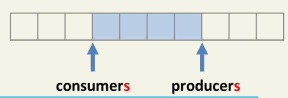
For a consumer process

→ Condition: the buffer is not empty.

For a producer process

→ Condition: the buffer is not full

Pseudo Code




```
#define NUM_ITEMS 30        // number of items produced and consumed
int empty = 1, full = 0;    // at beginning, buffer is empty and not full
int main(){
  #pragma omp parallel
    assign several threads to run produce(), and several to consume()
}
```

PRODUCER code

```
void put(T item){
  //add item to buffer
  //set empty to 0 (buffer is not empty)
  //set full to 1 IF buffer is full
}

void produce(){
  while (i < NUM_ITEMS) {
    #pragma omp critical(one)
    if (!full) {  // if full, don't do
                  // anything
      put(item); // puts data into buffer
      i++;       // incr. only if item
                 // added
    }
  }
}
```

Why?

CONSUMER code

```
T get() {
  // read item from buffer
  // set full to 0 (buffer is not full)
  // set empty to 1 IF buffer has not
  // items
  // return item;
}
void consume() {
  while (j < NUM_ITEMS) {
    #pragma omp critical(two)
    if (!empty) {
      x = get(); // read data from
                 // buffer
      j++;       // incr. only if item
                 // received
    }
  }
}
```

Produce-Consumer

Producer

```
#define BUFFER_SIZE 10      // Buffer size
#define NUM_ITEMS 30        // total number of producing and consuming

char buffer[BUFFER_SIZE];   //Buffer
int i_in = 0, i_out = 0;    //Two buffer indexes for producer/consumer
int num_items = 0;          //how many items in the buffer
int empty = 1, full = 0;    //buffer is empty and not full at beginning
int i =0, j=0;              //track any producer or consumer action

//producer code
void put(char ch) {
   buffer[i_in] = ch;                      // put character in circular buffer
   i_in = (i_in + 1) % BUFFER_SIZE;        // incr. producer index
   //keep track of number of items in buffer
   num_items++;                            // # of items in buffer
   if(num_items == 1) empty = 0;           // buffer is not empty anymore
   if(num_items==BUFFER_SIZE) full = 1;    // buffer is now full
}
void produce(int tid) {
   while (i < NUM_ITEMS) {
      #pragma omp critical(one)
      if (!full) {
         char ch = 'A' + rand()%26;
         put(ch);
         printf("T%d:Produced %c\t\tnow is %s\n",tid,ch,full?"Full":"Not full");
         i++;                              // only increment if item added
      }
}}
```

This part should be executed atomically for all producers – ok to run in parallel with consumers. e.g. you don't want more than one producer to check `!full` condition and proceed (what if there is only one spot left and two threads proceed because !full is true?)

Condition synchronization: can only *produce* if buffer is not full – if full, skip and try again later in the loop.

26

Produce-Consumer

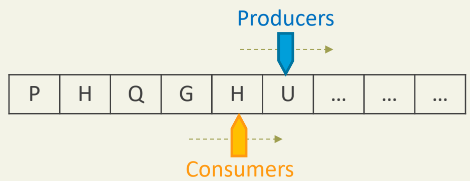
Consumer

```
char get() {
   char ch = buffer[i_out];                  // read item from circular buffer
   i_out = (i_out + 1) % BUFFER_SIZE;     // incr. consumer index
  //keep track of number of items in buffer
   num_items--;                                    // decr. # of items
   if (num_items == BUFFER_SIZE-1) full = 0;// buffer is not full anymore
   if (num_items == 0) empty = 1;               // buffer is now empty
   return ch;
}
void consume(int tid) {
   char ch;
   while (j < NUM_ITEMS) {
      #pragma omp critical(two)
      if (!empty) {
         ch = get();
         printf("T%d:Consumed %c\t\tnow is %s\n",tid,ch,empty?"Empty":"Not empty");
         j++;
      }
}}
int main(){
   #pragma omp parallel firstprivate(i,j) num_threads(8)
   {
      int tid = omp_get_thread_num();
      if(tid< omp_get_num_threads()/2)
         produce(tid);   //1st half of threads producing
      else
         consume(tid);   //2nd half consume
   }
}
```

Only one consumer can get

27

Produce-Consumer




```
T0: Producing P          now is Not full
T1: Consuming P          now is Empty
T2: Producing H          now is Not full
T2: Producing Q          now is Not full
T1: Consuming H          now is Not empty
T0: Producing G          now is Not full
T1: Consuming Q          now is Not empty
T0: Producing W          now is Not full
T1: Consuming G          now is Not empty
T0: Producing U          now is Not full
...
T5: Consuming A          now is Not empty
T7: Consuming P          now is Empty
```


Consumer-Producer Applications

Computer Science Applications:

- When data is set to a printer, a producer writes data to the buffer and the printer's process reads data from buffer
- When we have several "producer" threads and several "consumer" threads
  - Producer threads might "produce" requests for data.
  - Consumer threads might "consume" the request by finding or generating the requested data

Challenge

Producer-Consumer

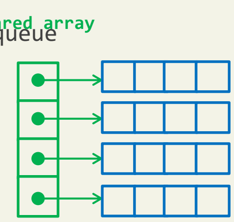
- The code we saw earlier is **not the best implementation** of the producer-consumer model.
  - Threads will be busy always even if they are practically doing nothing.
    - Example: if queue is empty, a consumer will still be busy running the loop and checking the if condition.
- OpenMP *cannot arbitrarily start/stop threads*.
  - Idea: `locks` may be used to implement condition synchronization, but we will not discuss this here.
- Other libraries can do this.
  - Java and POSIX Threads can do this
    - POSIX threads can explicitly create, run, stop, and destroy threads.

OpenMP is best when you want a fixed number of threads which will fully use the CPU, and the threads are more-or-less doing the same thing.

30

Message-Passing: *The Process*

1) Master thread creates a shared array of message queues: one for each thread.
2) Start the threads using a parallel directive. Each thread allocates space for its queue.
3) After ALL the threads are done step 2, each thread will
   - Send all its messages to other threads (putting them in other threads' queues)
   - Receive all message sent to it




```
for(i=0; i<num_msgs; i++){
        send_a_msg();
        try_receive();
}
while(! done())
        try_receive();
```


31

Message-Passing, *cont'd*

Sending Messages

Only one thread can enqueue

```
send_a_msg(){
        msg=... //any msg
        dest=... //thread id
        #pragma omp critical
        enqueue(queue, dest, my_id, msg);
}
```

Cant enqueuer and dequeue in parallel if only one item to dequeue

Receiving messages

```
try_receive(){
        if(queue_size == 0)
                return;
        elseif(queue_size == 1)
                #pragma omp critical
                dequeue(queue, &src_id, &msg);
        else
                dequeue(queue, &src_id, &msg);
        print_msg(src_id, msg);
}
```


Message-Passing, *cont'd*

Termination Detection

```
done(){
     queue_size = enqueued – dequeued;
     if(queue_size == 0 && done_sending == thread_count)
          return true;
     else
          return false;
}
```

each thread increments this after completing its send for loop

Message-Passing: *Summary*

1. When the program begins execution, master thread allocates an array of message queues: one for each thread.
   - This array needs to be shared among the threads, since any thread can send to any other thread, and hence any thread can enqueue a message in any of the queues.
2. Then, we can start the threads using a parallel directive, and each thread can allocate storage for its individual queue.
   - Note that one or more threads may finish allocating their queues before some other threads. Therefore, we need an ***explicit barrier*** so that when a thread encounters the barrier, it blocks until all the threads in the team have reached the barrier.
   - After all the threads have reached the barrier all the threads in the team can proceed.
3. You have seen that both enqueue and dequeue are protected by critical directive (as they access shared data).
4. After completing its sends, each thread increments `done_sending` before proceeding to its final loop of receives.
   incrementing done sending is a critical section and must be protected by either `critical` or `atomic` directives.

Remember: Locks

A lock consists of a data structure and functions that allow the programmer to explicitly enforce mutual exclusion in a critical section.

```
/* Executed by one thread */
Initialize the lock data structure;
. . .
/* Executed by multiple threads */
Attempt to lock or set the lock data structure;
Critical section;
Unlock or unset the lock data structure;
. . .
/* Executed by one thread */
Destroy the lock data structure;
```

Using Locks in the Message-Passing Program

- You could replace 'critical' with the locks

```
#   pragma omp critical
    /* q_p = msg_queues[dest] */
    Enqueue(q_p, my_rank, mesg);
```

```
/* q_p = msg_queues[dest] */
omp_set_lock(&q_p->lock);
Enqueue(q_p, my_rank, mesg);
omp_unset_lock(&q_p->lock);
```


36

Using Locks in the Message-Passing Program

- You could replace 'critical' with the locks

```
#   pragma omp critical
    /* q_p = msg_queues[my_rank] */
    Dequeue(q_p, &src, &mesg);
```

```
/* q_p = msg_queues[my_rank] */
omp_set_lock(&q_p->lock);
Dequeue(q_p, &src, &mesg);
omp_unset_lock(&q_p->lock);
```


37

Options for Nested Loops

Option 1: parallelize one loop only (e.g. the outer loop)

As we have done before

Option 2: use the **collapse** clause

When you have nested loops, you can use the collapse clause to apply the threading to multiple nested iterations

Example:

```
#pragma omp parallel for collapse(2)
for(int i=0; i<20; i++)
  for(int j=0; j<10; j++) {
     foo(i,j);
}
```

Option 3:

Use Nested Parallelism

Nested Parallelism

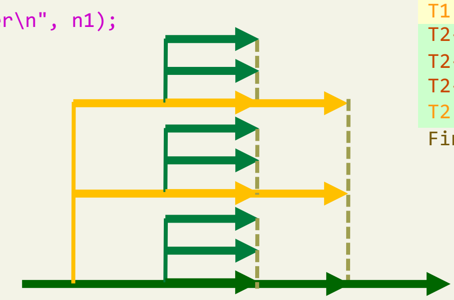
- Introduced in OpenMP 2.5
- Allows the programmer to create a parallel region within another parallel region.
- When a thread in the outer parallel region enters an inner parallel region, it **creates its own team** of threads and **becomes the master** of that team.
- The function omp_set_nested can be used to enable or disable nested parallelism
  - omp_set_nested(0) to disable
  - omp_set_nested(1) to enable
- **Many** openMP implementations **turn off this feature by default**

Enabling Nested Parallelism

```
omp_set_nested(1);      //enable nested parallelism
#pragma omp parallel num_threads(3)
{
  int n1 = omp_get_thread_num();
  printf("T%d: before\n", n1);

  #pragma omp parallel num_threads(3)
  { int n2 = omp_get_thread_num();
    printf("T%d-%d: inner\n", n1, n2);
  }
  printf("T%d: after\n", n1);
}
printf("Finished");
```

```
T0: before
T0-1: inner
T0-0: inner
T0-2: inner
T0: after
T1: before
T2: before
T1-0: inner
T1-1: inner
T1-2: inner
T1: after
T2-0: inner
T2-1: inner
T2-2: inner
T2: after
Finished
```



40

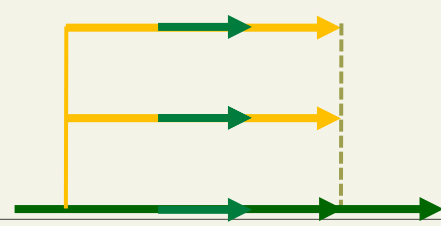
Disabling Nested Parallelism

```
omp_set_nested(0);      //disable nested parallelism
#pragma omp parallel num_threads(3)
{
  int n1 = omp_get_thread_num();
  printf("T%d: before\n", n1);

  #pragma omp parallel num_threads(3) //not spawning threads
  { int n2 = omp_get_thread_num();
    printf("T%d-%d: inner\n", n1, n2);
  }
  printf("T%d: after\n", n1);
}
printf("Finished");
```

```
T0: before
T0-0: inner
T0: after
T1: before
T2: before
T1-0: inner
T2-0: inner
T2: after
T1: after
Finished
```

Inner region is executed by only one thread

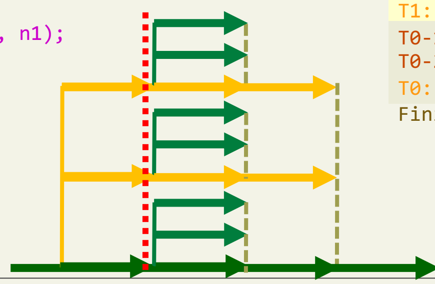


41

Nested Parallelism with Synchronization

```
omp_set_nested(1);        //enable nested parallelism
#pragma omp parallel num_threads(3)
{
  int n1 = omp_get_thread_num();
  printf("T%d: before\n", n1);
  #pragma omp barrier     //required only if needed
  #pragma omp parallel num_threads(3)
  { int n2 = omp_get_thread_num();
    printf("T%d-%d: inner\n", n1, n2);
  }
  printf("T%d: after\n", n1);
}
printf("Finished");
```

```
T0: before
T1: before
T2: before
T2-0: inner
T2-1: inner
T2-2: inner
T2: after
T0-0: inner
T1-0: inner
T1-1: inner
T1-2: inner
T1: after
T0-1: inner
T0-2: inner
T0: after
Finished
```

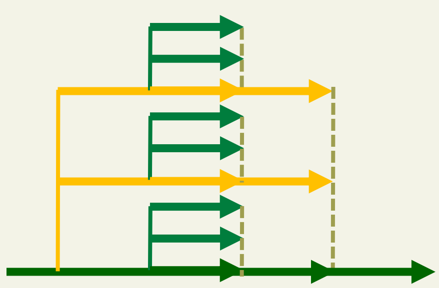


42

Nested Parallel For

```
omp_set_nested(1);        //enable nested parallelism
int i, j;
#pragma omp parallel for num_threads(3)   //spawn 3 threads here
 for(i=0; i<3; i++){
    int ni = omp_get_thread_num();
    #pragma omp parallel for num_threads(3)      //each thread spawns 3 more
     for(j=0; j<3; j++){
         int nj = omp_get_thread_num();
         printf("T%d-%d: %d x %d = %d\n",ni, nj, i, j, i*j);
     }
 }
```

```
T1-0: 1 x 0 = 0
T1-2: 1 x 2 = 2
T1-1: 1 x 1 = 1
T2-0: 2 x 0 = 0
T2-1: 2 x 1 = 2
T2-2: 2 x 2 = 4
T0-0: 0 x 0 = 0
T0-1: 0 x 1 = 0
T0-2: 0 x 2 = 0
```



43

Do We Really Need It?

Not necessarily

- You can create a large team of threads to do all the work in **one** parallel region.
  - You can also ***divide the work*** and ***reuse threads*** (in inner omp constructs such as #pragma omp for) without having to spawn new teams of threads.
- Speedup is not guaranteed with creating new teams of threads as there is an **overhead** of forking and joining threads
  - Refer to even-odd sort example from the previous lecture
- Having too many threads >> # of cores might reduce the performance (due to overhead, e.g. context switching)

When may I need it?

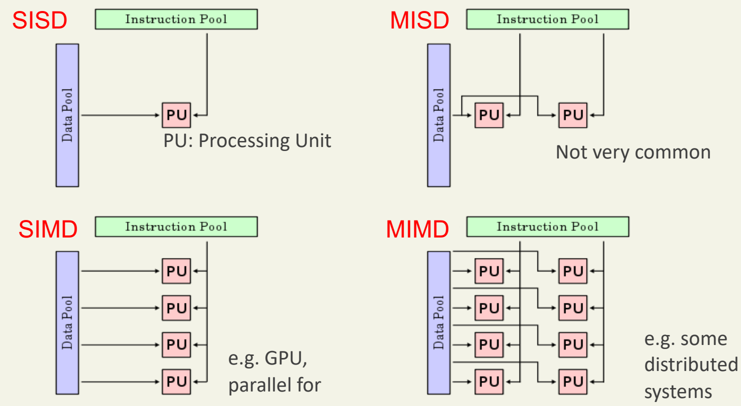
- It may be useful when parallelism isn't all exposed at once.
  - E.g. we want to run two functions in parallel, each of which needs 4 threads and you have 8 cores.

Remember

- **Core:**
  - an independent CPU on a chip.
    - Multicore: several independent CPUs on the same chip
  - Each CPU has it is own ALU, Control Unit, load/store units etc.
- **Thread:**
  - a sequence of instructions that are processed by a **single core**.

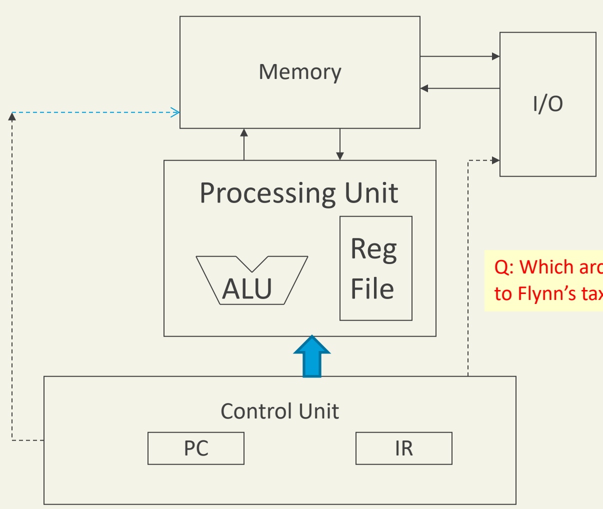
Flynn's Taxonomy

- Taxonomy of architectures based on the number of concurrent instruction streams and data streams available in the architecture.



46

The Von-Neumann Model



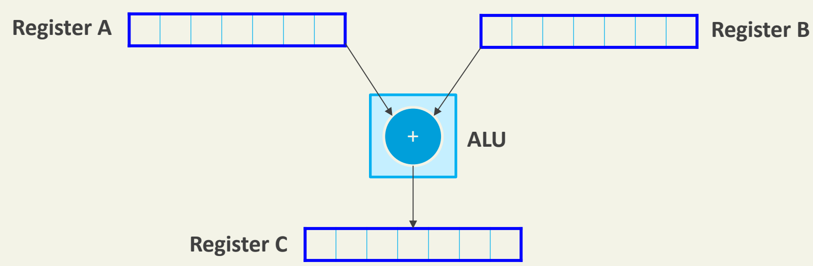
Q: Which architecture according to Flynn's taxonomy?

48

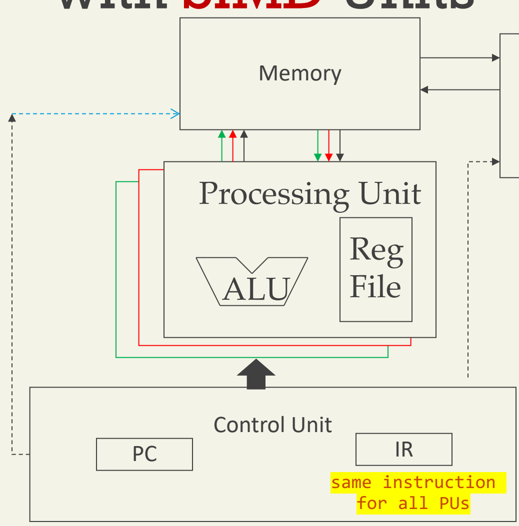
Scalar Architecture

Scalar processors includes

- **Scalar registers:**
  - Each register is m **bit** long
- **functional units**
  - One operation at a time
- **Example:**
  - `Add C, A, B     # C = A+B`

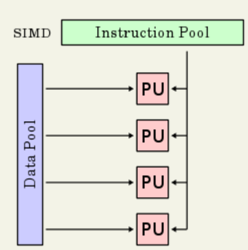


The Von-Neumann Model with SIMD Units



SIMD ALUs:
- multiple PUs
- not multiple cores.

Single fetch/execute, but on vector elements.



Vector Architecture

Vector processors includes

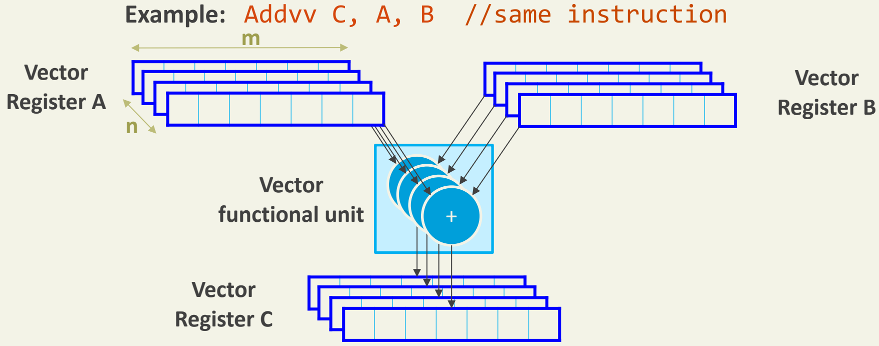
Vector registers:

Each register holds n-elements (where n is the length of the vector), each element with m bits

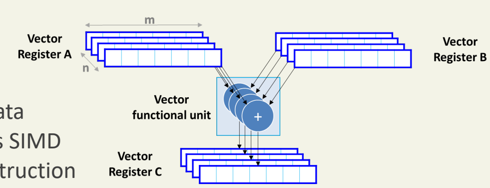
Vector functional units

Implements SIMD

Example: `Addvv C, A, B  //same instruction`



SIMD Parallelism



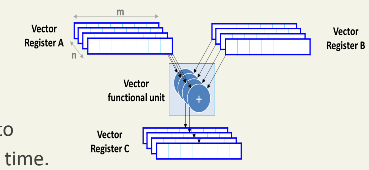
SIMD parallelism

- Same instruction, multiple data
- Requires a CPU that supports SIMD
- **One thread** for one SIMD instruction

SIMD and Thread Level Parallelism

- Both may be automatically **invoked by the compiler IF** the compiler is smart enough to know it is safe to use them.
  - In many situations, the compiler fails to make this decision.
- Both may be **explicitly used by the programmer** using OpenMP (SIMD is ***only*** supported in OpenMP 4.0+).
  - The programmer is then asserting that it is safe to parallelise this part
  - In other words, we are helping the compiler to figure out how to vectorize or parallelize.

OpenMP 4.0: New Features



SIMD construct

- Explicit vectorization of both serial and parallelized loops.
  - Vectorization: single instruction applied to several elements of a vector at the same time.
  - Usually, there are 2, 4, or 8 SIMD lanes.
- The new SIMD directives control how the compiler generates data parallel instructions.

OpenMP for devices

- Now OpenMP can pass some of the workload to ***co-processors***
- Limitations in gcc: not for GPU, needs processor support

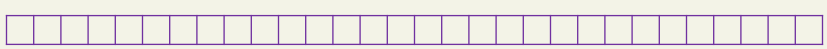
User-defined reductions

- Define with `omp declare reduction` then use it as a reduction operator

`simd` Construct

WITHOUT vectorization: each element is processed by one thread in one iteration:

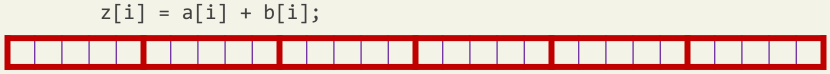
```
for (i = 0; i < len; ++i)
        z[i] = a[i] + b[i];
```

Thread 0 

WITH vectorization: each *chuck* of elements is processed by ***one*** **thread at once** by one instruction.

i.e. the for loop is unrolled so that it can take use SIMD instructions to perform the same operation on multiple elements in a single instruction.

```
#pragma omp simd          //can use private, shared, reduction, etc.
for (i = 0; i < len; ++i)
   z[i] = a[i] + b[i];
```

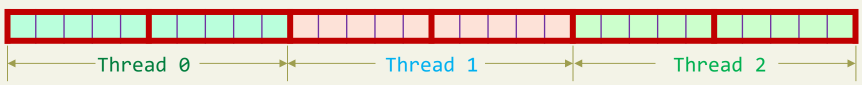
Thread 0 

for simd Construct

With vectorisation and thread parallelism:

- Iterations are divided among threads (loop is parallelized), and then each thread's group is "vectorized".

```
#pragma omp for simd      //can use private, shared, reduction, etc.
for (i = 0; i < len; ++i)
  z[i] = a[i] * b[i];
```

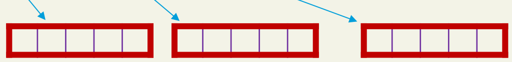


Declaring simd Function

If you are calling a function in a vectorized loop, you need to declare it simd.

- The compiler coverts (i) the arguments and (ii) the function's parameters and return values **to vectors**.

```
#pragma omp declare simd  //must declare simd if used in vectorized loop
double add(double x, double y){
  return x + y;
}
```



```
void foo(double* a, double *b, double *z, int len){
  long i;
  #pragma omp for simd reduction(+:sum)
  for (i = 0; i < len; ++i)
    z[i] = add(a[i], b[i]);
}
```

Concluding Remarks

OpenMP

- OpenMP
  - Standard for programming shared-memory systems.
  - Uses special functions and preprocessor directives called pragmas.
  - Start multiple threads rather than multiple processes.
  - Many OpenMP directives can be modified by clauses.
- OpenMP supports thread synchronization
- A major problem with **shared memory** programs is the possibility of **race conditions**
- OpenMP provides several mechanisms for insuring **mutual exclusion** in critical sections.
  - Critical directives
  - Named critical directives
  - Atomic directives
  - Simple locks

Concluding Remarks, cont'd

OpenMP

- Use a block-partitioning of the iterations in a **parallelized for loop**
- In OpenMP the **scope** of a variable is the collection of threads to which the variable is accessible
- A **reduction** is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result

Conclusion/Up Next

- What we covered today (review key concepts):
  - Some More Examples
    - Matrix multiplication
    - Max reduction
  - Asides and Comments on OpenMP
- Next:
  - Speed and Efficiency