

Parallel Programming with OpenMP

Sorting (OddEvenSort, QuickSort)

Intro to OpenMP Tasking

Bibliography

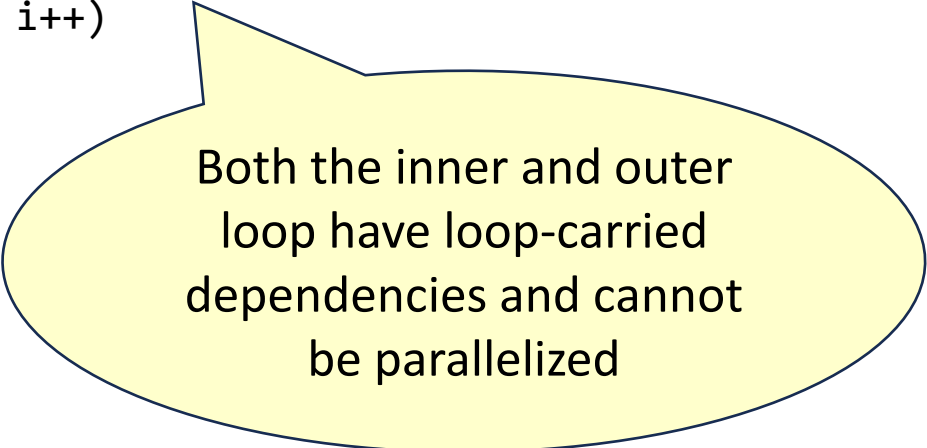
- [Pacheco]: Peter Pacheco, Matthew Malensek, Introduction to Parallel Programming, 2nd Edition, Morgan Kaufmann Publisher, March 2020, Chapter 5.6, Chapter 5.10
- <https://www.openmp.org/wp-content/uploads/sc16-openmp-booth-tasking-ruud.pdf>
- https://hpc-wiki.info/mediawiki/hpc_images/9/91/Hpc.nrw_05_Introduction-Tasking.pdf
- https://www.bsc.es/sites/default/files/public/mare_nostrum/hpc-events/omp_tutorial.pdf

Parallel Sorting

Classic Bubblesort

```
void Bubblesort_serial(int a[], int n)
{
    int list_length, i;
    double tmp;

    for (list_length = n; list_length >= 2; list_length--)
    {
        for (i = 0; i < list_length - 1; i++)
        {
            if (a[i] > a[i + 1])
            {
                tmp = a[i + 1];
                a[i + 1] = a[i];
                a[i] = tmp;
            }
        }
    }
}
```



Both the inner and outer loop have loop-carried dependencies and cannot be parallelized

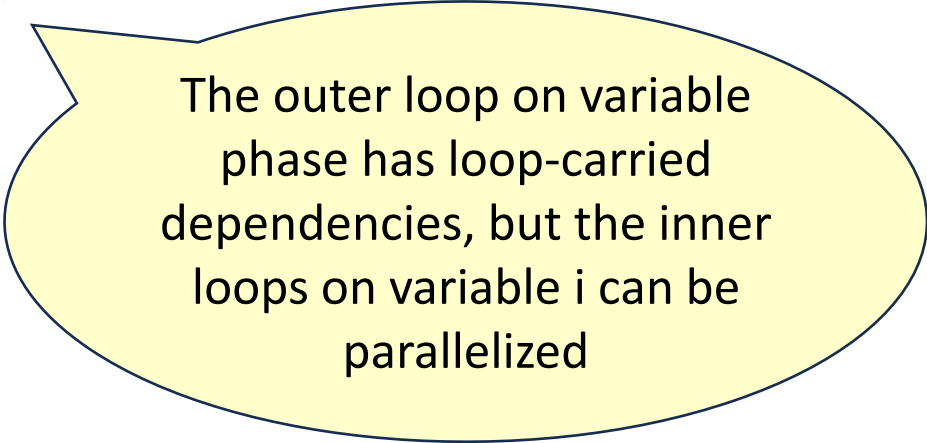
Simple Classic Sorting Algorithms

- The Simple Classic Sorting algorithms are generally based on following idea:
 - Do several runs over the array, at each run swap some pairs of elements that are not in the right order
- This raises issues when parallelizing:
 - Cannot have different threads do the runs in parallel because of the swapping
 - Cannot have different threads share the array in one run
 - Classical algorithms need reformulation for parallelization

Odd-Even Sort

```
void Odd_even_serial(int a[], int n)
{
    int phase, i, tmp;

    for (phase = 0; phase < n; phase++) {
        if (phase % 2 == 0)
            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
            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;
                }
            }
    }
}
```



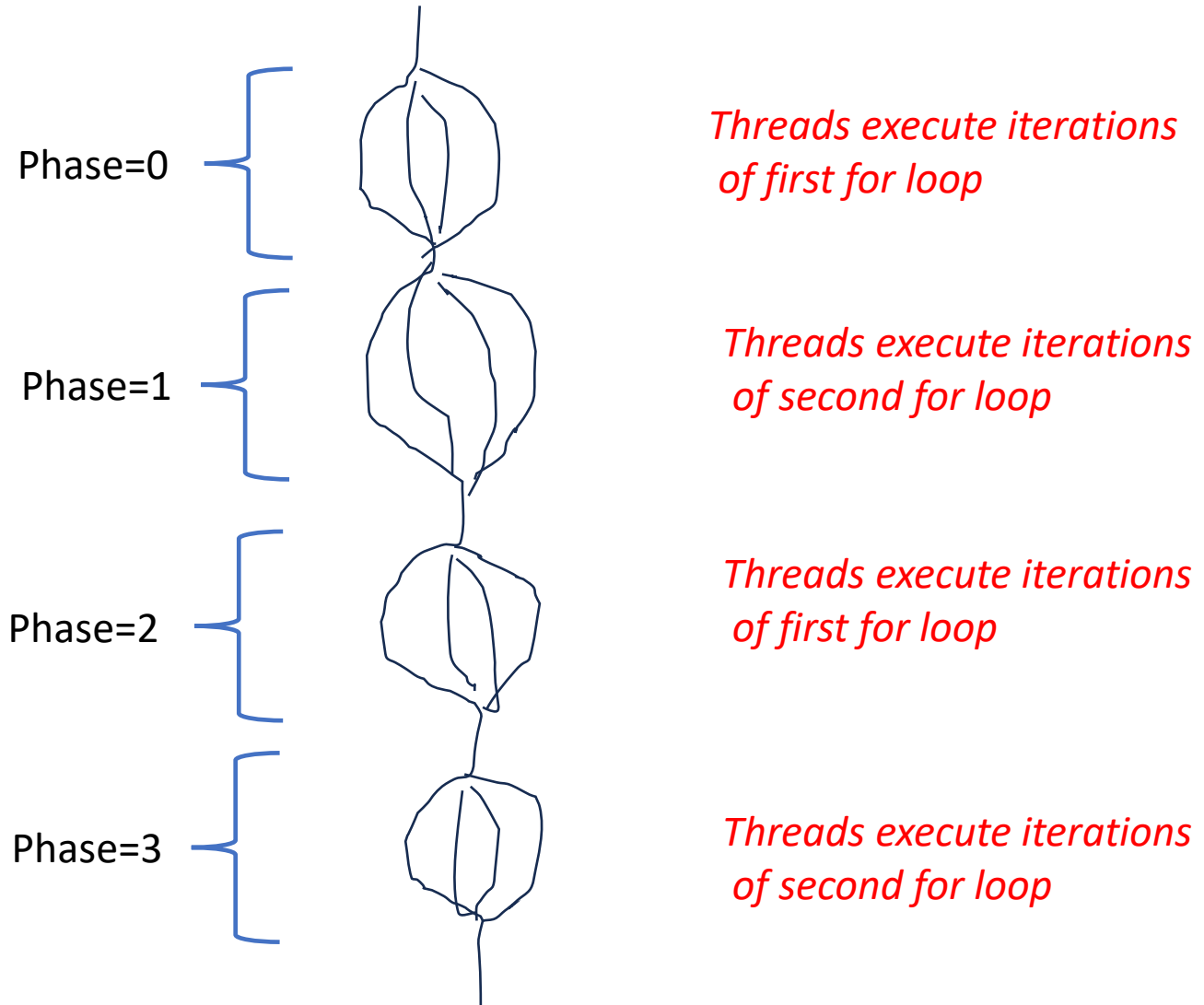
The outer loop on variable phase has loop-carried dependencies, but the inner loops on variable i can be parallelized

```

void Odd_even_v1(int a[], int n)
{
    int phase, i, tmp;
    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;
                }
            }
    }
}

```

Thread create-join in V1




```

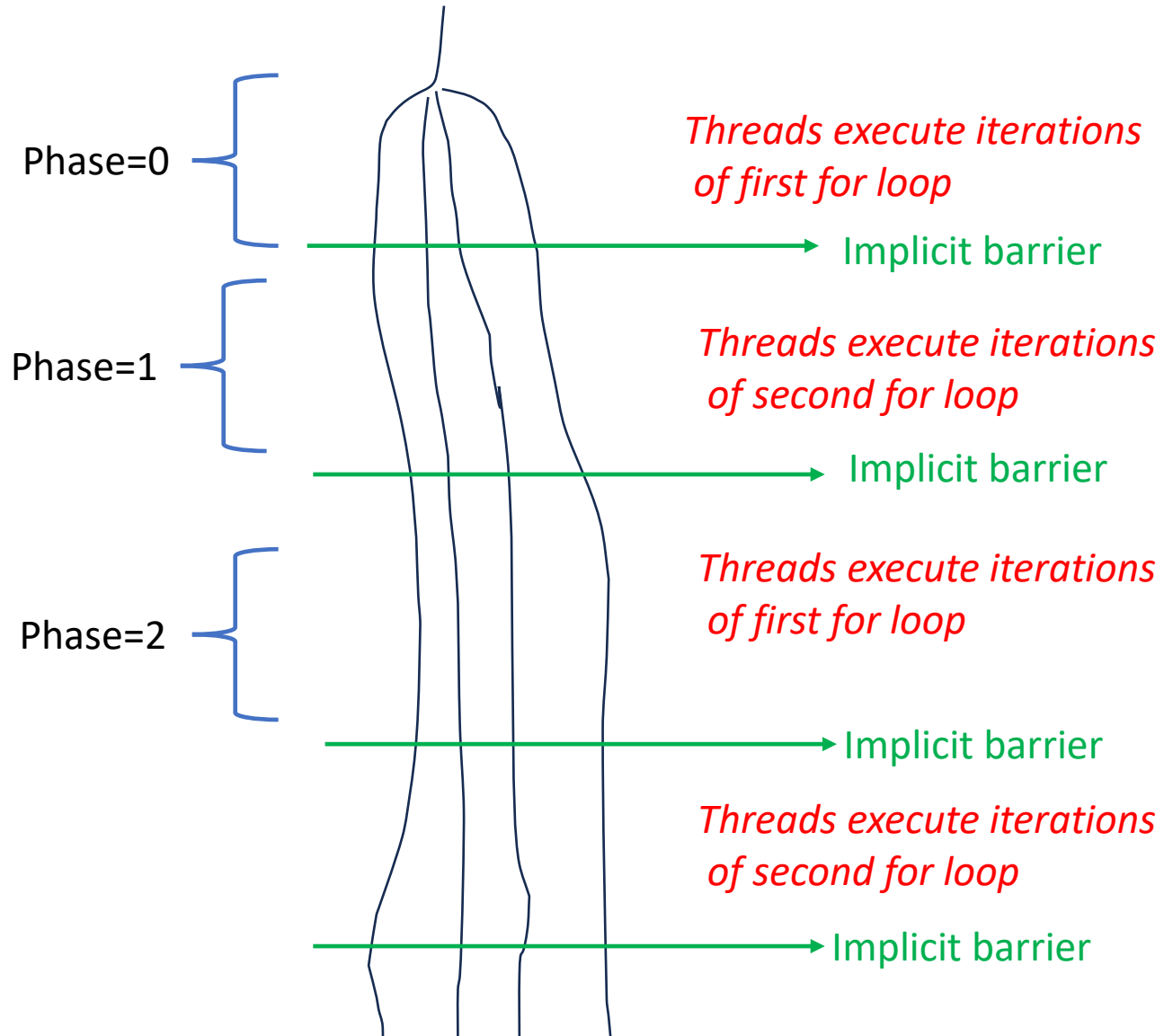
void Odd_even_v2(int a[], int n)
{
    int phase, i, tmp;
    #pragma omp parallel num_threads(thread_count) default(none) shared(a, n)\
                                private(i, tmp, phase)

    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;
                }
            }
    }
}

```

The threads are only created once. The parallel for worksharing constructs use the existing threads

Thread create-join in V2



Odd-Even Sort Performance

	N=10000		N=100000		N=200000	
	Time	S	Time	S	Time	S
Serial	0.136		16.84		72.27	
Parallel V1 Threads=2	0.269	0.50	11.89	1.41	47.34	1.52
Parallel V2 Threads=2	0.149	0.91	10.48	1.60	41.83	1.72
Parallel V1 Threads=4	0.271	0.50	7.74	2.17	30.70	2.35
Parallel V2 Threads=4	0.147	0.92	6.30	2.67	27.36	2.64

*Improvements
due to the fact that
threads are
only created once!*

Good practice:

- If you must parallelize an *inner* loop:
 - if possible **create the team of threads only once** with a simple parallel section before the outer loop
 - the inner loop gets a simple **omp for** directive (**NOT** a **omp parallel for**)
 - Careful if there are other statements in the body of the outer loop!!!
- Parallelizing repeated stencil pattern application (Heat2D example) could benefit as well from such an optimization

QuickSort

- A classical sorting algorithm
- Uses a divide and conquer strategy
- Main steps:
 1. Split the array through a pivot, such that all elements to the left are smaller and all elements to the right are bigger
 2. Recursively repeat for left and right subarrays until done

QuickSort - Serial

```
void quickSort_serial(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);
        quickSort_serial(arr, low, pi - 1);
        quickSort_serial(arr, pi + 1, high);
    }
}

void sort_serial(int arr[], int size)
{
    quickSort_serial(arr, 0, size - 1);
}
```

QuickSort helper functions

```
void swap(int *a, int *b)
```

```
{
```

```
    int t = *a;
```

```
    *a = *b;
```

```
    *b = t;
```

```
}
```

```
int partition(int arr[], int low, int high)
```

```
{
```

```
    int pivot = arr[high];
```

```
    int i = (low - 1);
```

```
    for (int j = low; j <= high - 1; j++)
```

```
    {
```

```
        if (arr[j] <= pivot)
```

```
        {
```

```
            i++;
```

```
            swap(&arr[i], &arr[j]);
```

```
        }
```

```
    }
```

```
    swap(&arr[i + 1], &arr[high]);
```

```
    return (i + 1);
```

```
}
```

QuickSort parallel

- The two recursive calls contained in quicksort operate on distinct subarrays -> they could be done in the same time
- Parallel V1: use parallel sections to create 2 sections, containing the recursive calls, each section executed by a thread

QuickSort parallel V1 (with sections)

```
void quickSort_sections(int arr[], int low, int high)
{
    if (low < high) {
        int pi = partition(arr, low, high);

#pragma omp parallel sections shared(arr) firstprivate( low,high, pi)
        {
#pragma omp section
            quickSort_sections(arr, low, pi - 1);
#pragma omp section
            quickSort_sections(arr, pi + 1, high);
        }
    }
}

void sort_parallel_v1(int arr[], int size)
{
    quickSort_sections(arr, 0, size - 1);
}
```

QuickSort Performance

	N=1M		N=5M		N=10M	
	Time	S	Time	S	Time	S
Serial	0.16		2.06		7.44	
Parallel V1	2.36	0.07	15.02	0.13	34.16	0.21

- Parallel V1 has a disastrous performance
- Each recursive call creates a new parallel region, we have a nested parallelism where the number of created threads explodes
- We must limit the number of threads that are created

QuickSort parallel V2

- with sections and limit on thread creation: if there are already too many threads created, switch over to serial version

```

void quickSort_sections_t1(int arr[], int low, int high, int nthreads)
{
    if (low < high)
    {
        if (nthreads == 1) {
            quickSort_serial(arr, low, high);
            return;
        }
        else {
            int pi = partition(arr, low, high);
#pragma omp parallel sections
            {
#pragma omp section
                quickSort_sections_t1(arr, low, pi - 1, nthreads / 2);
#pragma omp section
                quickSort_sections_t1(arr, pi + 1, high, nthreads / 2);
            }
        }
    }
}

void sort_parallel_v2(int arr[], int size)
{
    quickSort_sections_t1(arr, 0, size - 1, NUMTHREADS);
}

```

QuickSort Performance

NUMTHREADS=8

	N=1M		N=5M		N=10M	
	Time	S	Time	S	Time	S
Serial	0.16		2.06		7.44	
Parallel V1	2.36	0.07	15.02	0.13	34.16	0.21
Parallel V2	0.10	1.6	1.65	1.24	6.26	1.18

- Parallel V2 (parallel sections but with thread number limit) brings a small speedup but not satisfactory

QuickSort parallel V3 (with tasks)

```
void quickSort_tasks(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
#pragma omp task shared(arr)
        {
            quickSort_tasks(arr, low, pi - 1);
        }
#pragma omp task shared(arr)
        {
            quickSort_tasks(arr, pi + 1, high);
        }
    }
}

void sort_parallel_v3(int arr[], int size) {
    omp_set_num_threads(NUMTHREADS);
#pragma omp parallel
    {
#pragma omp single nowait
        quickSort_tasks(arr, 0, size - 1);
    }
}
```

QuickSort Performance

NUMTHREADS=8

	N=1M		N=5M		N=10M	
	Time	S	Time	S	Time	S
Serial	0.16		2.06		7.44	
Parallel V1	2.36	0.07	15.02	0.13	34.16	0.21
Parallel V2	0.10	1.6	1.65	1.24	6.26	1.18
Parallel V3	0.07	2.28	0.56	3.68	1.53	4.86

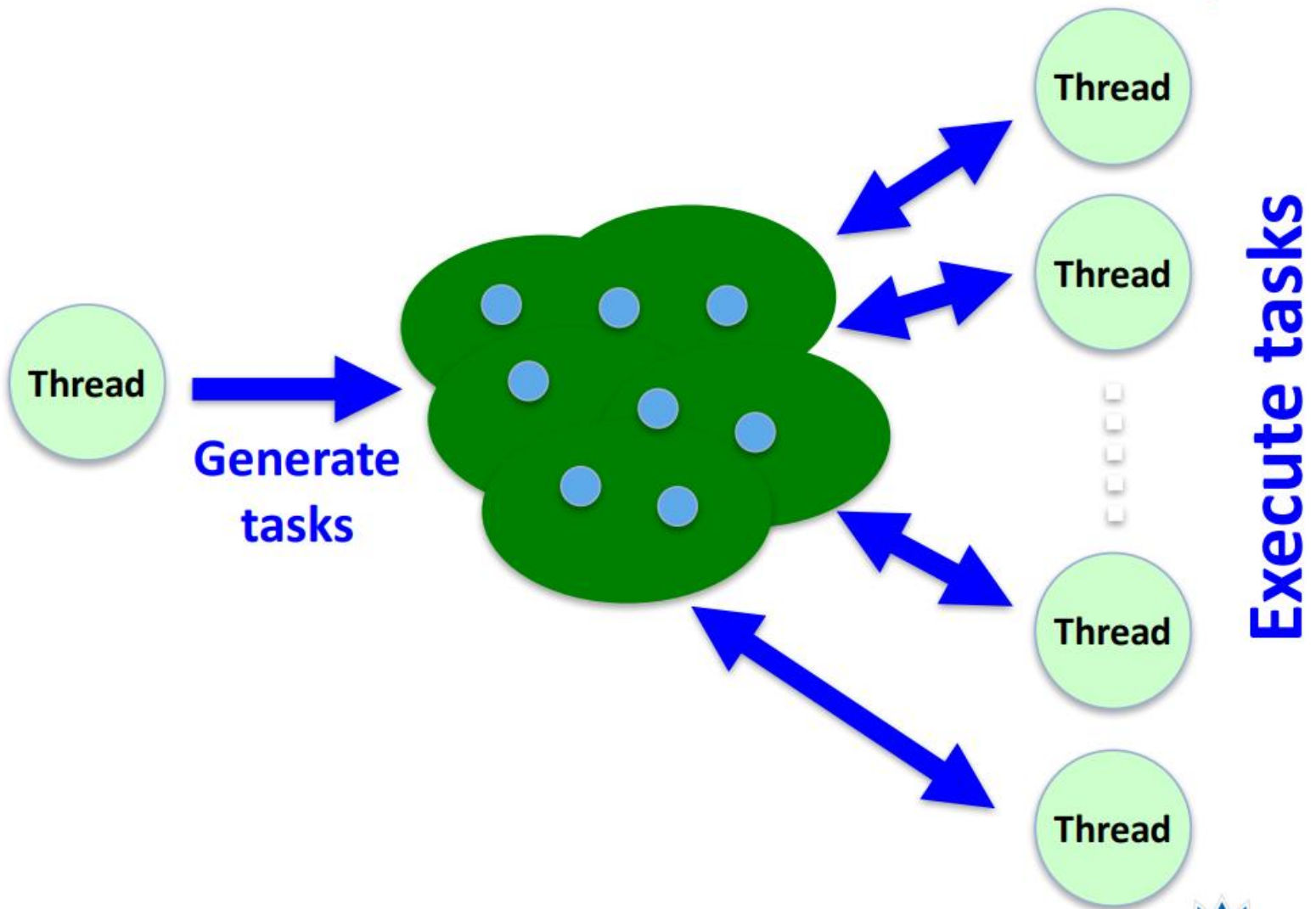
- Parallel V3 (using omp tasks) brings a significant speedup

OpenMP Tasking

OpenMP tasks

- **A task** *is a chunk of independent work*
- When a program declares a task, it guarantees that it contains work that could be executed in parallel
- **#pragma omp task**
- The run time system decides on the scheduling of the tasks:
 - *At which moment* a task is assigned for execution
 - *Which thread* will execute it
- OpenMP tasking works like a **task pool**:
 1. some threads define that they have tasks that can be done in parallel
 2. the task is “put in a pool”
 3. at some later moment the system takes tasks “out of the pool” and assigns them to different threads for execution

The Tasking Concept in OpenMP



Creating tasks - Examples

```
void ex1()  
{  
  #pragma omp parallel num_threads(2)  
  {  
    #pragma omp task  
    {  
      printf("car ");  
    }  
    #pragma omp task  
    {  
      printf("race ");  
    }  
  }  
}
```

car race race car

```
void ex2()  
{  
  #pragma omp parallel num_threads(2)  
  {  
    #pragma omp single  
    {  
      #pragma omp task  
      {  
        printf("car ");  
      }  
      #pragma omp task  
      {  
        printf("race ");  
      }  
    }  
  }  
}
```

car race

OR

race car

Creating tasks

- **tasks must be launched from within a parallel region**
- If they should be **launched only by one of the threads in the team**, the pattern is:

```
# pragma omp parallel
# pragma omp single
{
    . . .
    # pragma omp task
    . . .
}
```

- the parallel directive creates a team of threads and the single directive instructs the runtime to only launch tasks from a single thread.
- If the single directive is omitted, each thread in the team launches a task

Executing tasks

```
void ex3()
{
#pragma omp parallel num_threads(2)
{
#pragma omp single
{
    printf("A ");
#pragma omp task
{
    printf("car ");
}
#pragma omp task
{
    printf("race ");
}
    printf(". ");
}
}
}
```

A . race car

```
void ex4()
{
#pragma omp parallel num_threads(2)
{
#pragma omp single
{
    printf("A ");
#pragma omp task
{
    printf("car ");
}
#pragma omp task
{
    printf("race ");
}
#pragma omp task wait
    printf(". ");
}
}
}
```

A race car .

Executing tasks

- The order in which tasks are executed is not specified
- The moment when a task will be launched or finished is unknown
- If you need a task to be finished before continuing, use the **task wait** directive (similar to a barrier for tasks)
 - The task wait directive performs *shallow* task synchronization: it waits on the completion of child tasks of the current task - only direct children, not all descendant tasks

Data scoping with tasks

- Some rules from Parallel Regions apply:
 - Static and Global variables are shared
 - Automatic Storage (local) variables are private
 - **Task variables are firstprivate** unless shared in the enclosing context
 - Only shared attribute is inherited
 - Exception: Orphaned Task variables are firstprivate by default

Fibonacci Example

```
int fib_serial(int n)
{
    if (n < 2)
        return n;
    int x = fib_serial(n - 1);
    int y = fib_serial(n - 2);
    return x + y;
}
```

Parallelization strategy:
execute each recursive call
as a separate task that can
run in parallel.
Before computing the
sum, we must have a
barrier waiting that the
tasks have been actually
finished.

Fibonacci Parallel V0 ?

```
int fib_p_v0(int n)
{
    if (n < 2) return n;
    int x=0, y=0;
    #pragma omp task
    { x = fib_p_v0(n - 1); }
    #pragma omp task
    { y = fib_p_v0(n - 2); }
    #pragma omp task wait
    return x + y;
}
```

Recursive function,
recursively generates tasks

```
int fib_parallel_v0(int n)
{
    int result;
    #pragma omp parallel num_threads(NUMTHREADS) shared(result)
    {
        #pragma omp single
        { result = fib_p_v0(n); }
    }
    return result;
}
```

Only one thread starts the
function that recursively
generates tasks

Fibonacci Parallel V0 = WRONG

```
int fib_p_v0(int n)
{
    if (n < 2) return n;
    int x=0, y=0;
    #pragma omp task
    { x = fib_p_v0(n - 1); }
    #pragma omp task
    { y = fib_p_v0(n - 2); }
    #pragma omp task wait
    return x + y;
}

int fib_parallel_v0(int n)
{
    int result;
    #pragma omp parallel num_threads(NUMTHREADS) shared(result)
    {
        #pragma omp single
        { result = fib_p_v0(n); }
    }
    return result;
}
```

The result of this function
is always zero!

**Variables x and y are private as per
the default scoping rules in tasks!**

The memory locations that are
assigned the results of fib(n-1) and
fib(n-2) are not the same as the
memory locations
declared at the beginning of the
function!

Fibonacci Parallel V1

```
int fib_p_v1(int n)
{
    if (n < 2) return n;
    int x, y;
    #pragma omp task shared(x)
    { x = fib_p_v1(n - 1); }
    #pragma omp task shared(y)
    { y = fib_p_v1(n - 2); }
    #pragma omp task wait
    return x + y;
}
```

```
int fib_parallel_v1(int n)
{
    int result;
    #pragma omp parallel num_threads(NUMTHREADS) shared(result)
    {
        #pragma omp single
        { result = fib_p_v1(n); }
    }
    return result;
}
```

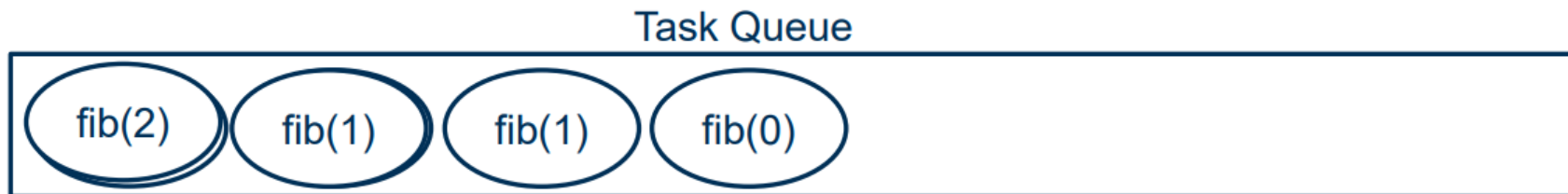
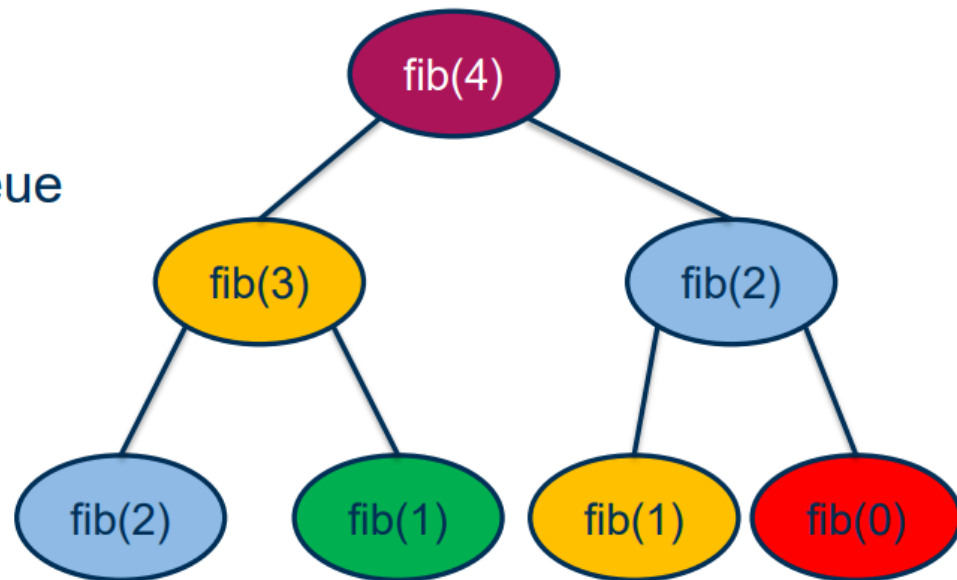
Which thread is doing which task?

```
int fib_p_v1(int n)
{
    int tid = omp_get_thread_num();
    printf("Thread %d is doing fib(%d) \n", tid, n);

    if (n < 2)
        return n;
    int x, y;
    #pragma omp task shared(x) private(tid)
    {
        x = fib_p_v1(n - 1);
    }
    #pragma omp task shared(y) private(tid)
    {
        y = fib_p_v1(n - 2);
    }
    #pragma omp task wait
    return x + y;
}
```

Fibonacci tasks illustration

- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks



Parallel Fibonacci Performance

	N=25		N=35		N=45	
	Time	S	Time	S	Time	S
Serial	0.001		0.048		5.78	
Parallel V1 Threads=8	0.192	0.005	23.13	0.002		

- Parallel V1 has a mega-disastrous performance
- “Speedup” values show that the parallel version is approx. 500 times slower than the serial one
- Each recursive call creates a new task, we have parallel tasks even for small values of n. **We must limit the number of tasks that are created !**

The if clause

- The if clause of a task construct:
 - allows to optimize task creation/execution
 - reduces parallelism but also reduces the pressure in the runtime's task pool
 - for “very” fine grain tasks you may need to do your own (manual) if
- If the expression of the “if” clause evaluates to false
 - the encountering task is suspended
 - the new task is executed immediately
 - the parent task resumes when the task finishes
- This is known as undeferred task

Fibonacci Parallel V2

```
int fib_p_v2(int n)
{
    if (n < 2) return n;
    int x, y;
    #pragma omp task shared(x) if (n > 30)
    { x = fib_p_v2(n - 1); }
    #pragma omp task shared(y) if (n > 30)
    { y = fib_p_v2(n - 2); }
    #pragma omp task wait
    return x + y;
}

int fib_parallel_v2(int n)
{
    int result;
    #pragma omp parallel num_threads(NUMTHREADS) shared(result)
    {
        #pragma omp single
        { result = fib_p_v2(n); }
    }
    return result;
}
```


Parallel Fibonacci Performance

	N=25		N=35		N=45	
	Time	S	Time	S	Time	S
Serial	0.001		0.048		5.78	
Parallel V1 Threads=8	0.192	0.005	23.13	0.002		
Parallel V2 Threads=8	0.013	0.07	0.39	0.12	46.94	0.12

- Parallel V2 still has very bad performance

Fibonacci Parallel V3

```
int fib_p_v3(int n)
{
    if (n < 2)    return n;
    if (n <= 30) return fib_serial(n);
    int x, y;
#pragma omp task shared(x)
    {    x = fib_p_v3(n - 1); }
#pragma omp task shared(y)
    {    y = fib_p_v3(n - 2); }
#pragma omp task wait
    return x + y;
}

int fib_parallel_v3(int n)
{
    int result;
#pragma omp parallel Num threads(NUMTHREADS) shared(result)
    {
#pragma omp single
        {    result = fib_p_v3(n);    }
    }
    return result;
}
```

Parallel Fibonacci Performance

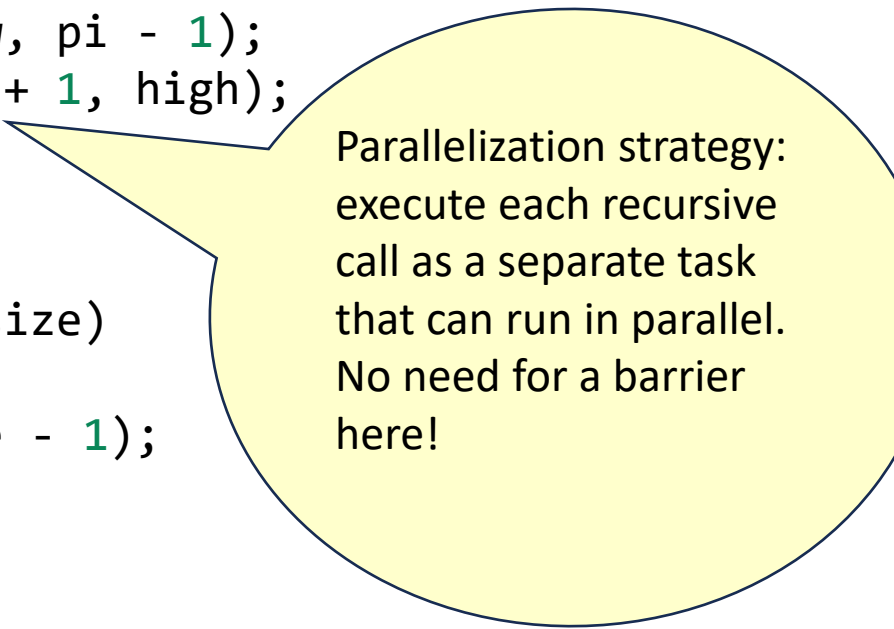
	N=25		N=35		N=45	
	Time	S	Time	S	Time	S
Serial	0.001		0.048		5.78	
Parallel V1 Threads=8	0.192	0.005	23.13	0.002		
Parallel V2 Threads=8	0.013	0.07	0.39	0.12	46.94	0.12
Parallel V3 Threads=8	0.000		0.013	3.69	1.59	3.73

- Parallel V3 has a quite good performance !

QuickSort Example

```
void quickSort_serial(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);
        quickSort_serial(arr, low, pi - 1);
        quickSort_serial(arr, pi + 1, high);
    }
}

void sort_serial(int arr[], int size)
{
    quickSort_serial(arr, 0, size - 1);
}
```



Parallelization strategy:
execute each recursive
call as a separate task
that can run in parallel.
No need for a barrier
here!

QuickSort parallel V3 (with tasks)

```
void quickSort_tasks(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
#pragma omp task shared(arr) firstprivate( low, pi)
        {
            quickSort_tasks(arr, low, pi - 1);
        }
#pragma omp task shared(arr) firstprivate( high, pi)
        {
            quickSort_tasks(arr, pi + 1, high);
        }
    }
}

void sort_parallel_v3(int arr[], int size) {
    omp_set_num_threads(NUMTHREADS);
#pragma omp parallel
    {
#pragma omp single nowait
        quickSort_tasks(arr, 0, size - 1);
    }
}
```

QuickSort Performance

NUMTHREADS=8

	N=1M		N=5M		N=10M	
	Time	S	Time	S	Time	S
Serial	0.16		2.06		7.44	
Parallel V1	2.36	0.07	15.02	0.13	34.16	0.21
Parallel V2	0.10	1.6	1.65	1.24	6.26	1.18
Parallel V3	0.07	2.28	0.56	3.68	1.53	4.86

- Parallel V3 (using omp tasks) brings a significant speedup

QuickSort parallel V4

- We can apply the optimization strategy of limiting the creation of tasks when the array size is too small
- The value of SIZELIMIT must be found by experimental finetuning

QuickSort parallel V4

```
void quickSort_tasks_bis(int arr[], int low, int high) {  
    if (low < high) {  
        if (high - low < SIZELIMIT) {  
            quickSort_serial(arr, low, high);  
            return; }  
        int pi = partition(arr, low, high);  
        #pragma omp task shared(arr) firstprivate( low, pi)  
        { quickSort_tasks_bis(arr, low, pi - 1); }  
  
        #pragma omp task shared(arr) firstprivate( high, pi)  
        { quickSort_tasks_bis(arr, pi + 1, high); }  
    }  
}  
  
void sort_parallel_v2_2(int arr[], int size)  
{  
    omp_set_num_threads(NUMTHREADS);  
    #pragma omp parallel  
    {  
        #pragma omp single nowait  
        quickSort_tasks_bis(arr, 0, size - 1);  
    }  
}
```


QuickSort Performance

NUMTHREADS=8, SIZELIMIT=10000

	N=1M		N=5M		N=10M	
	Time	S	Time	S	Time	S
Serial	0.16		2.06		7.44	
Parallel V1	2.36	0.07	15.02	0.13	34.16	0.21
Parallel V2	0.10	1.6	1.65	1.24	6.26	1.18
Parallel V3	0.07	2.28	0.56	3.68	1.53	4.86
Parallel V4	0.03	5.33	0.33	6.24	1.14	6.53

- Very good speedup with V4!

Using tasks to parallelize irregular structures


- List traversal: for each value in a list, test if it is a prime number

```
node_t * current = head;
while (current != NULL) {
    test_prime(current->val);
    current = current->next;
}
```

Using tasks to parallelize irregular structures

- List traversal:

```
#pragma omp parallel num_thread(8)
{
#pragma omp single
{
    node_t *current = head;
    while (current != NULL)
    {
#pragma omp task
        {
            test_prime(current->val);
        }
        current = current->next;
    }
}
```



ONE thread traverses the list and creates tasks
ALL threads take tasks and execute them

Tasking overhead

- Typical overheads in task-based programs are:
 - Task creation: populate task data structure, add task to task queue
 - Task execution: retrieve a task from the queue (may including *work stealing*)
- If tasks become too fine-grained, overhead becomes noticeable
 - Execution spends a higher relative amount of time in the runtime
 - Task execution contributing to runtime becomes significantly smaller

Tasks vs Threads

- Task models make complex algorithms easier to parallelize
 - *Programmers can think in concurrent pieces of work*
 - *Mapping of concurrent execution to threads handled elsewhere*
 - *Task creation can be irregular* (e.g., recursion, list traversal, graph traversal)
- Task models are inherently composable
 - A pool of threads executes all created tasks
 - Tasks from different modules can freely mix
- Some scenarios are more suitable for traditional threads:
 - Granularity too coarse for tasking
 - *Static allocation* of parallel work is typically easier with threads