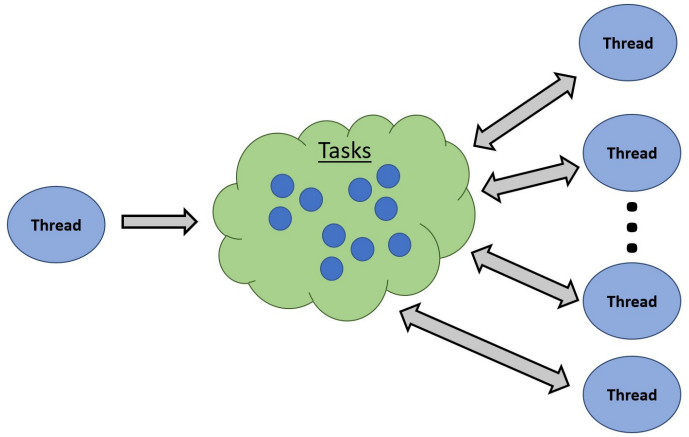
OpenMP Tasks

Cameron Foss

Blue Waters Capstone

Outline

- What are OpenMP tasks?
  - task constructs
  - directives



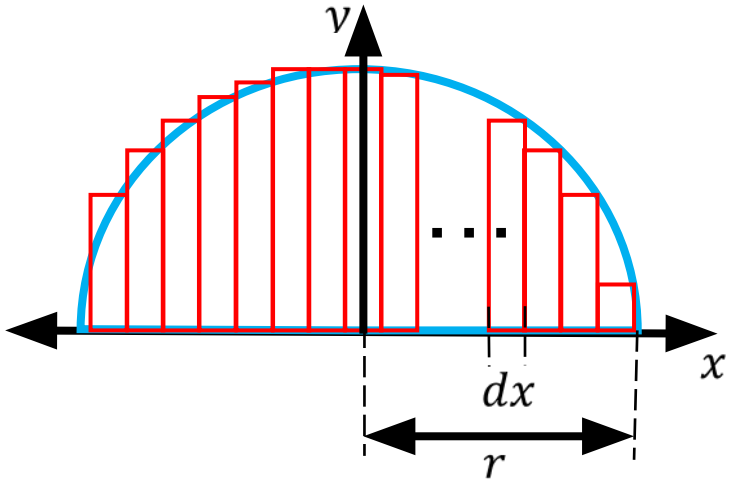

```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

recursive call

- How to implement in Code
  - "A race car"→ a hello-world to OpenMP tasking
  - OpenMP barrier and taskwait
  - Fibonacci recursive example

    1,2,3,5,8,13 …

- Assignment using recursion
  - Compute pi recursion: implement omp task directives



OpenMP Tasking Model

Tasking facilitates the parallelization of applications where units of work are generated dynamically, as in ***recursive*** *structures or* ***while*** *loops.*
[API User's Guide]

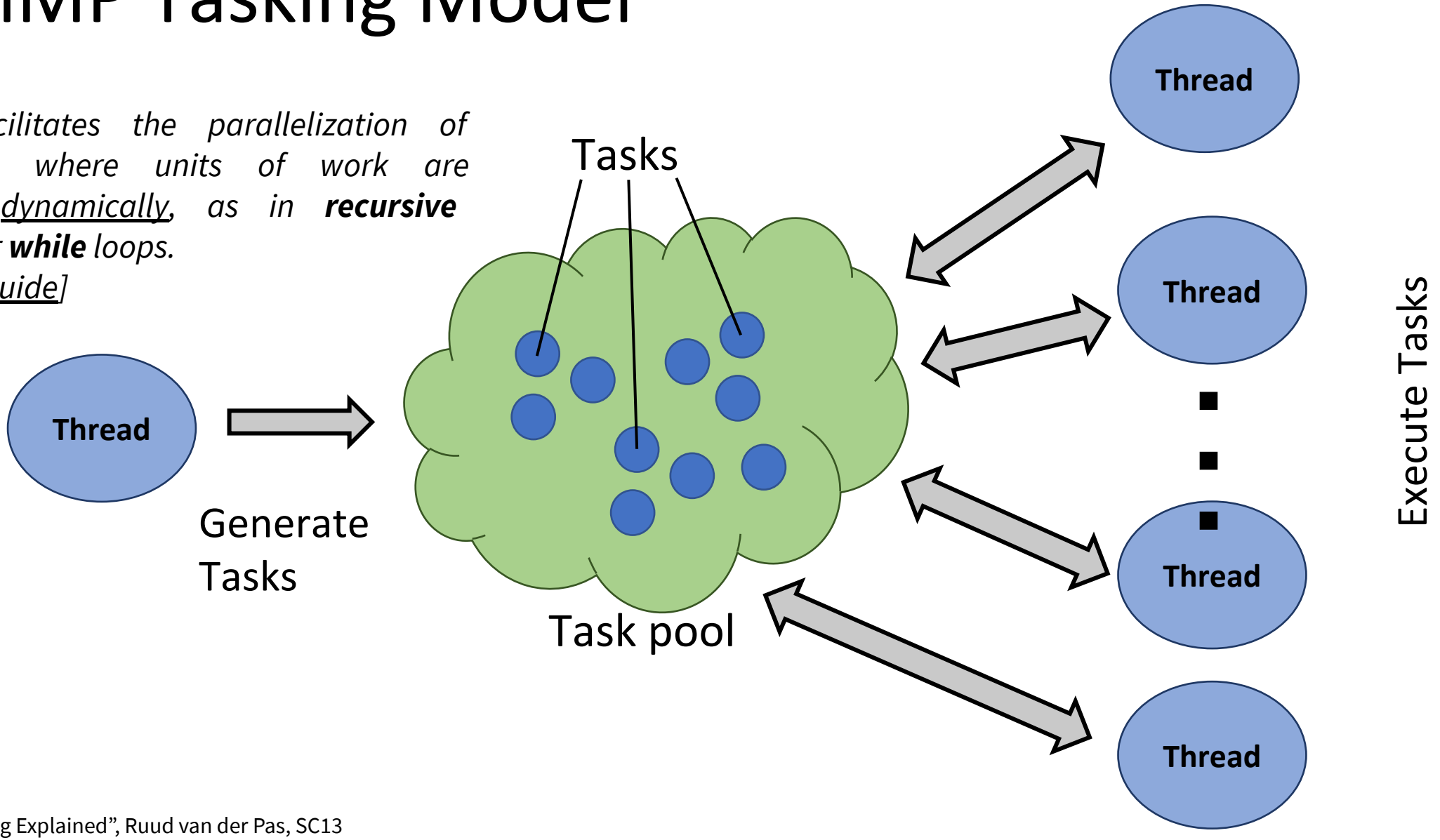


Image source: "OpenMP Tasking Explained", Ruud van der Pas, SC13

…revisit after MPI and Hybrid (MPI+OpenMP)

A more detailed example…

Parallel model implemented for the simulator based on MPI+OpenMP tasking.

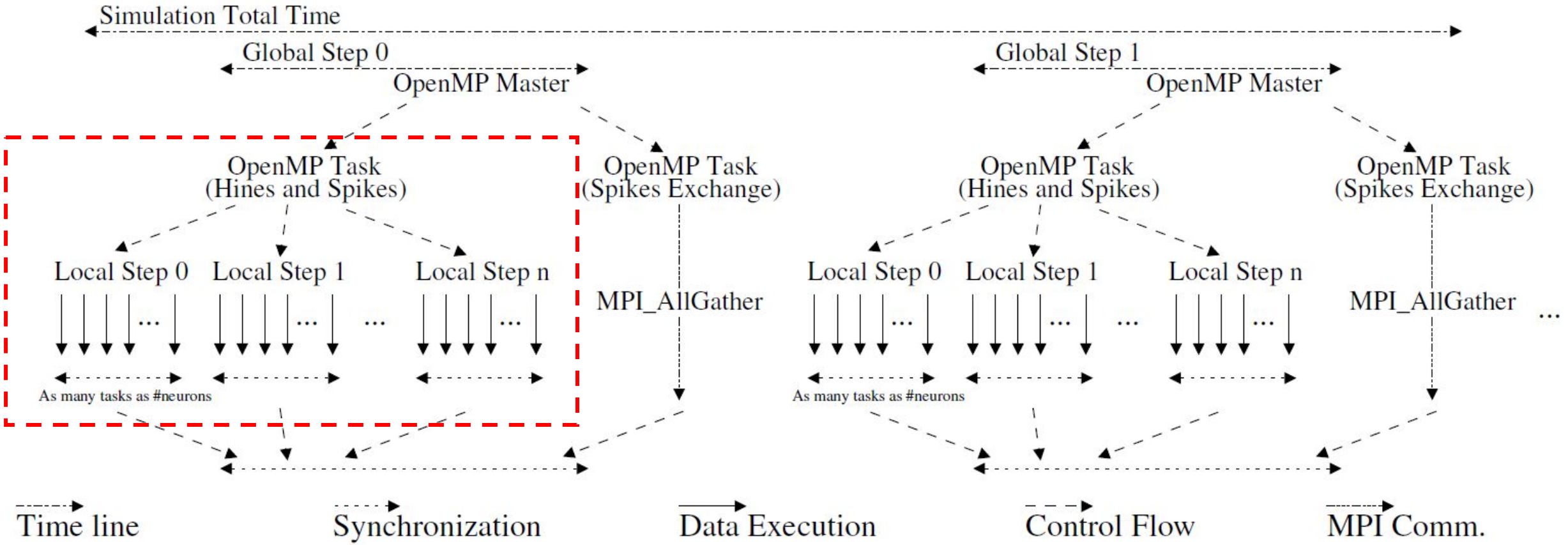


Image source: “MPI+OpenMP tasking scalability for multi-morphology simulations of the human brain”, Parallel Computing, Volume 84, May 2019, Pages 50-61

Why use tasking…?

- Up until OpenMP 3.0, certain types of parallelism was not feasible
  - This required messy workarounds at best
- Threads all run the same code. Tasks allow us to assign different blocks of code that are independent to different threads.
- enable irregular parallelism
- Allowed specific code blocks to be run simultaneously and independently of other task blocks.

Task Execution

- Tasks are generated by specifying the `task` construct.
- Tasks can be generated anywhere in the code.
- A task is composed of:
  - Code to be executed
  - Data environment (inputs to be used and outputs to be generated)
  - A location where the task will be executed (a thread)

http://icl.cs.utk.edu/classes/cosc462/2017/pdf/W43%20-%20OpenMP%20Tasking.pdf

Task Types

- **explicit** tasks are generated using the task directive

```
# pragma omp task clause
{

   block of code ...

}
```

- **implicit** tasks are inherent to parallel constructs and function similarly to explicit tasks

Data environment

```
# pragma omp task clause
{

   block of code ...

}
```

clause can be:

- **depend (*list*)**
- **if (expression)**
- **final (expression)**
- untied
- mergeable
- **default (shared | firstprivate | none)**
- **private (*list*)**
- **firstprivate (*list*)**
- **shared (*list*)**
- priority (value)

defines storage locations where task dependencies exist

Generate a task `if` a condition is met

Stop generating new tasks if a condition is met

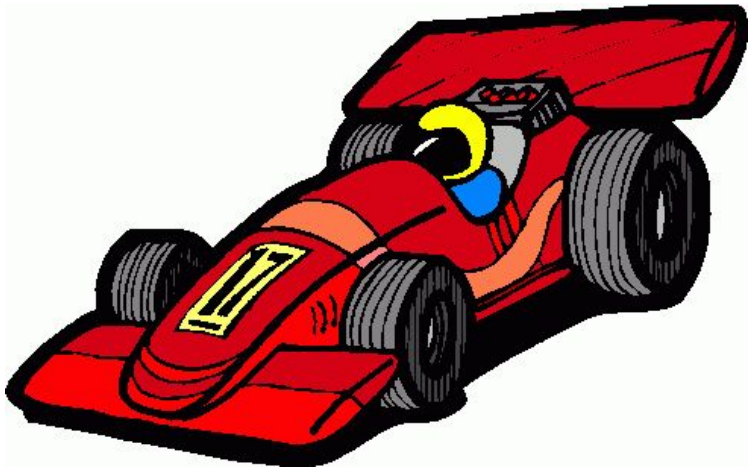
Memory storage requirements on input output variables

- inputs to be used and outputs to be generated
- The `task` directive takes the following data-sharing attribute clauses that define the data environment of the task.

http://icl.cs.utk.edu/classes/cosc462/2017/pdf/W43%20-%20OpenMP%20Tasking.pdf

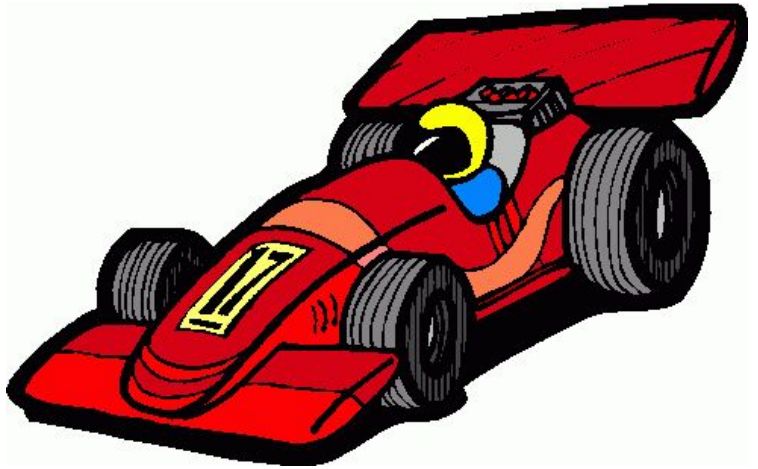
Tasking Explained…

Objective: Write a program that prints either “A race car” or “A car race” and maximize the parallelism.



Source: “OpenMP Tasking Explained”, Ruud van der Pas, SC13

A race car OR A car race…(1)



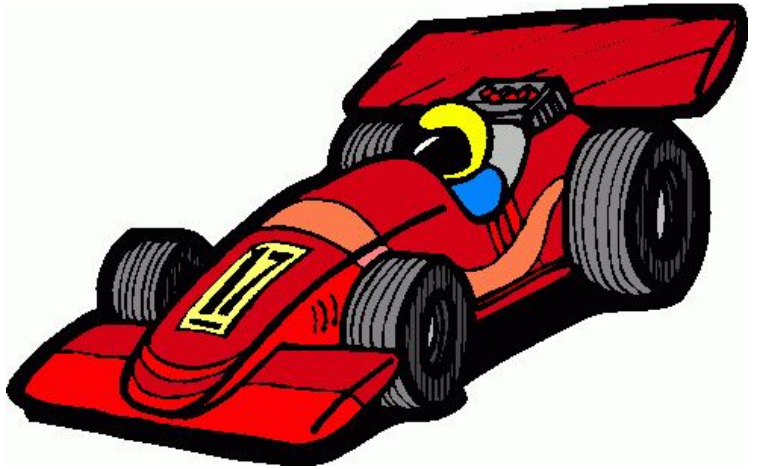
```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf(“A ”);
    printf(“race ”);
    printf(“car ”);

    printf(“.\n”);
    return(0);
}
```

Q: What will this program print?

```
$ gcc racecar.c
$ ./a.out
 A race car .
$
```

Source: “OpenMP Tasking Explained”, Ruud van der Pas, SC13

A race car OR A car race…(2)



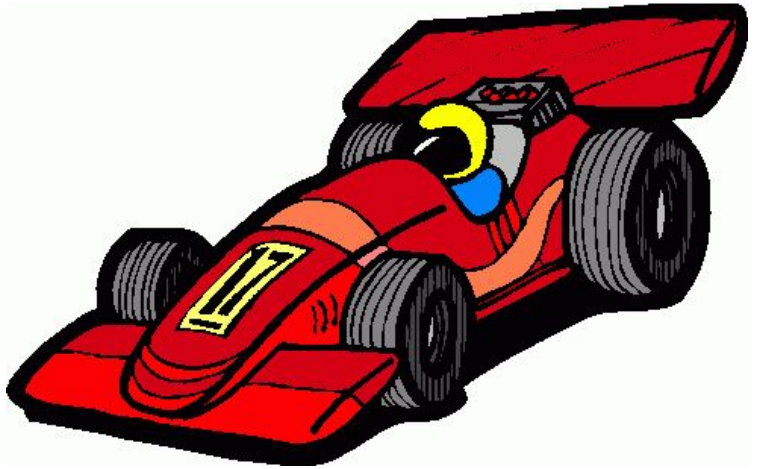
```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        printf("A ");
        printf("race ");
        printf("car ");
    } // End of omp parallel region

    printf(".\n");
    return(0);
}
```

Q: What will this program print when we use 2 threads?

```
$ gcc –fopenmp racecar_omp.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car A race car .
$
```

Source: "OpenMP Tasking Explained", Ruud van der Pas, SC13

A race car OR A car race…(3)



```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        #pragma omp single
        {
         printf("A ");
         printf("race ");
         printf("car ");
        }
    } // End of omp parallel region

    printf(".\n");
    return(0);
}
```

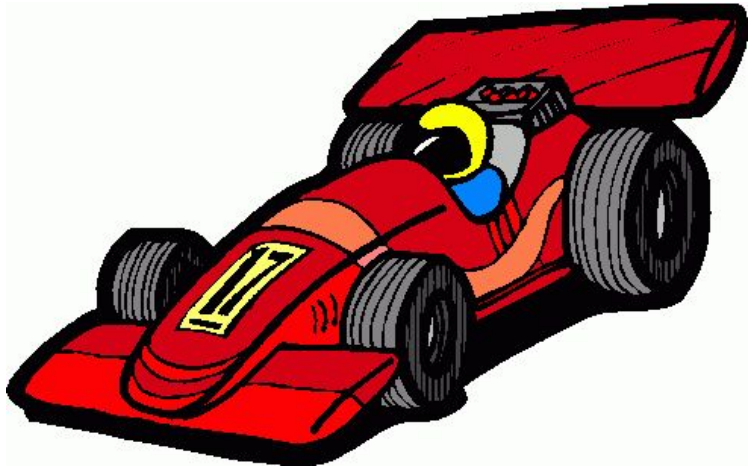
Q: What will this program print when we use 2 threads?

```
$ gcc –fopenmp racecar_omp.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car .
$
```

It only prints it once now because the omp single block is run by only one (*single*) thread.

Source: "OpenMP Tasking Explained", Ruud van der Pas, SC13

A race car OR A car race…(4)



```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
       #pragma omp single
       {
        printf("A ");
        #pragma omp task
        {  printf("race "); }
        #pragma omp task
        {  printf("car ");  }
       }
    } // End of omp parallel region

    printf(".\n");
    return(0);
}
```

Q: What will this program print when we use 2 threads?

```
$ gcc –fopenmp racecar_omp.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car .
$ ./a.out
A car race .
```

Tasks are executed in arbitrary order!

Note: It may take many attempts before both "A race car" and "A car race" appear.

Source: "OpenMP Tasking Explained", Ruud van der Pas, SC13

A race car OR A car race…(5)

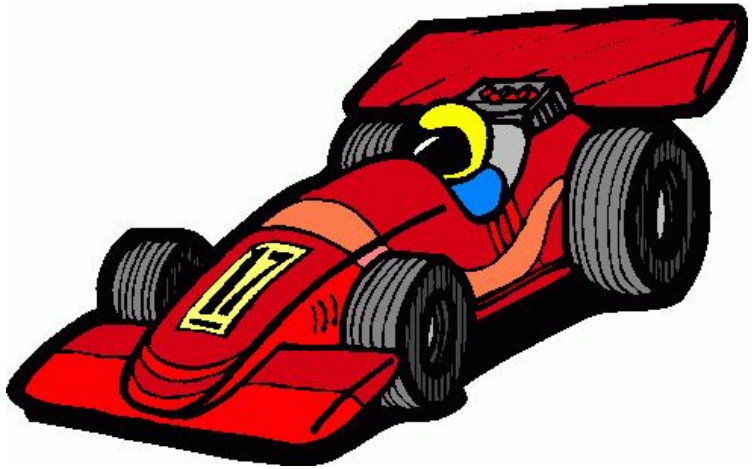
Objective: Write a program that prints either “A race car” or “A car race” and maximize the parallelism.

COMPLETED

Objective 2: Add “is fun to watch.” to the end of the sentence…

Source: “OpenMP Tasking Explained”, Ruud van der Pas, SC13

A race car OR A car race…(6)



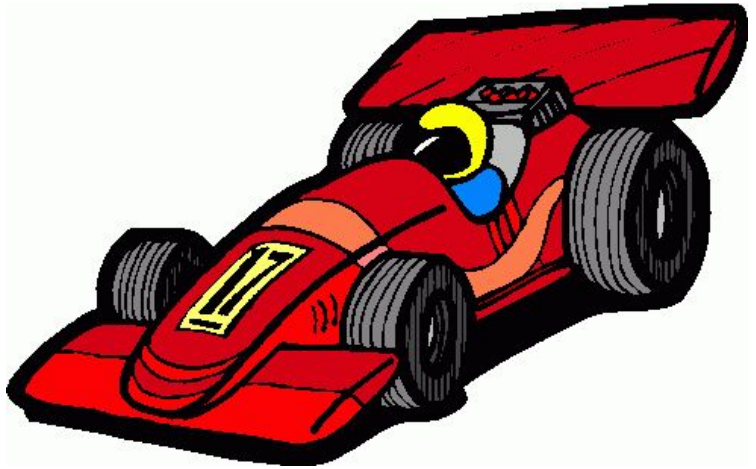
```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
     #pragma omp parallel
     {
        #pragma omp single
        {
          printf("A ");
          #pragma omp task
          {  printf("race "); }
          #pragma omp task
          {  printf("car ");  }
          printf("is fun to watch ");
        }
     } // End of omp parallel region

     printf(".\n");
     return(0);
}
```

Q: What will this program print when we use 2 threads?

```
$ gcc –fopenmp racecar_omp.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A is fun to watch race car .
$ ./a.out
A race is fun to watch car .
 …
```

Source: "OpenMP Tasking Explained", Ruud van der Pas, SC13

A race car OR A car race…(7)



```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
       #pragma omp single
       {
         printf("A ");
         #pragma omp task
         {  printf("race "); }
         #pragma omp task
         {  printf("car ");  }
         #pragma omp taskwait
         printf("is fun to watch ");
       }
    } // End of omp parallel region
    printf(".\n");
    return(0);
}
```

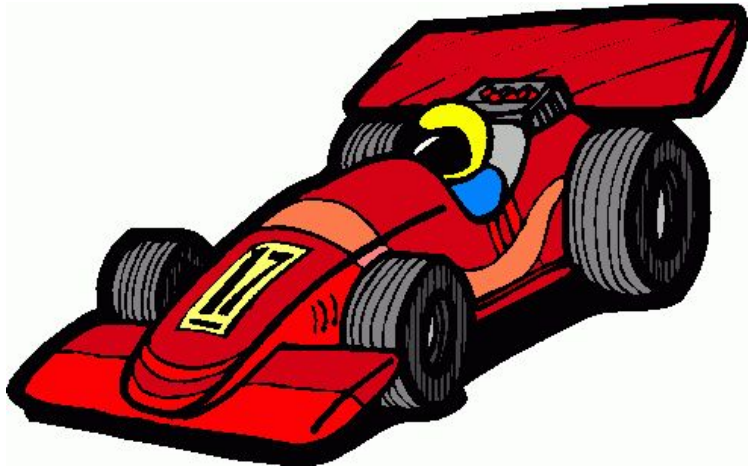
Task 1 is generated

Task 2 is generated

Synchronization point:
"Wait here until all previous tasks have finished…"

Source: "OpenMP Tasking Explained", Ruud van der Pas, SC13

A race car OR A car race…(8)



```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
     #pragma omp parallel
     {
        #pragma omp single
        {
          printf("A ");
          #pragma omp task
          {  printf("race "); }
          #pragma omp task
          {  printf("car ");  }
          #pragma omp taskwait
          printf("is fun to watch ");
        }
     } // End of omp parallel region
     printf(".\n");
     return(0);
}
```

Q: What will this program print when we use 2 threads?

```
$ gcc –fopenmp racecar_omp.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car is fun to watch .
$ ./a.out
A car race is fun to watch .
 …
```

Source: "OpenMP Tasking Explained", Ruud van der Pas, SC13

A race car OR A car race…(9)

Objective: Write a program that prints either “A race car” or “A car race” and maximize the parallelism.

COMPLETED

Objective 2: Add “is fun to watch.” to the end of the sentence… **COMPLETED**

Source: “OpenMP Tasking Explained”, Ruud van der Pas, SC13

Recap so far…

- When a thread encounters a `#pragma omp task` it generates a task
  - That is, the number of threads that encounter a `#pragma omp task` will generate that many number of tasks.
  - Can control with a directive `#pragma omp single` or more dynamically with the function `omp_set_num_threads`

- Tasks are executed in parallel and in **arbitrary order.**
  - Implies a race occurs between tasks that are initiated by the same thread.

- `#pragma omp taskwait` can be used to execute code by the main thread strictly after all tasks are completed.

Recursion

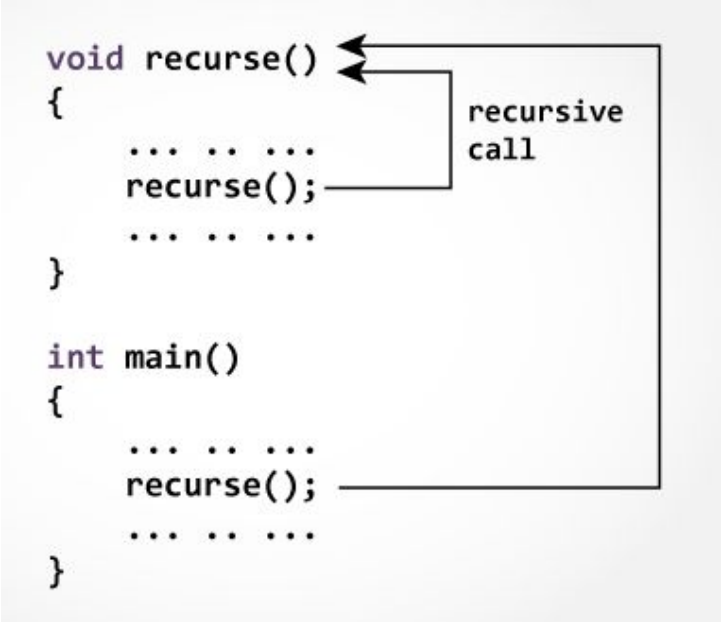
Recursion is defined as "relating to or involving a program or routine of which a part requires the application of the whole, so that its explicit interpretation requires in general many successive executions."

- Tasking is often used in codes that use recursion
- **Recursion**, in simple terms, means defining a problem in terms of itself.
- A **recursive function** is a function that calls itself iteratively until some condition is met.

Calculating the n^{th} element in a Fibonacci sequence

$$\text{fib}(n = 6) = [1, 2, 3, 5, 8, \mathbf{13}] = \mathbf{13}$$

```
int fib(int n) {
  int i, j;
  if (n<2) {   return n; }
  else  {
        i=fib(n-1);
        j=fib(n-2);
        return i+j;
    }
}
```



https://www.cs.utah.edu/~germain/PPS/Topics/recursion.html

Recursion – Compute Pi

```
double pi_r (double h, unsigned depth, unsigned maxdepth, unsigned long long begin, unsigned long long niters)
{
    if (depth < maxdepth)
    {
        double area1, area2;
        // Process first half
        area1 = pi_r (h, depth+1, maxdepth, begin, niters/2-1);
        // Process second half
        area2 = pi_r (h, depth+1, maxdepth, begin+niters/2, niters/2);
        return area1+area2;
    }
    else
    {
        unsigned long long i;
        double area = 0.0;
        for (i = begin; i <= begin+niters; i++)
        {
            double x = h * (i - 0.5);
            area += (4.0 / (1.0 + x*x));
        }
        return area;
    }
}
```

Recursion – Compute Pi

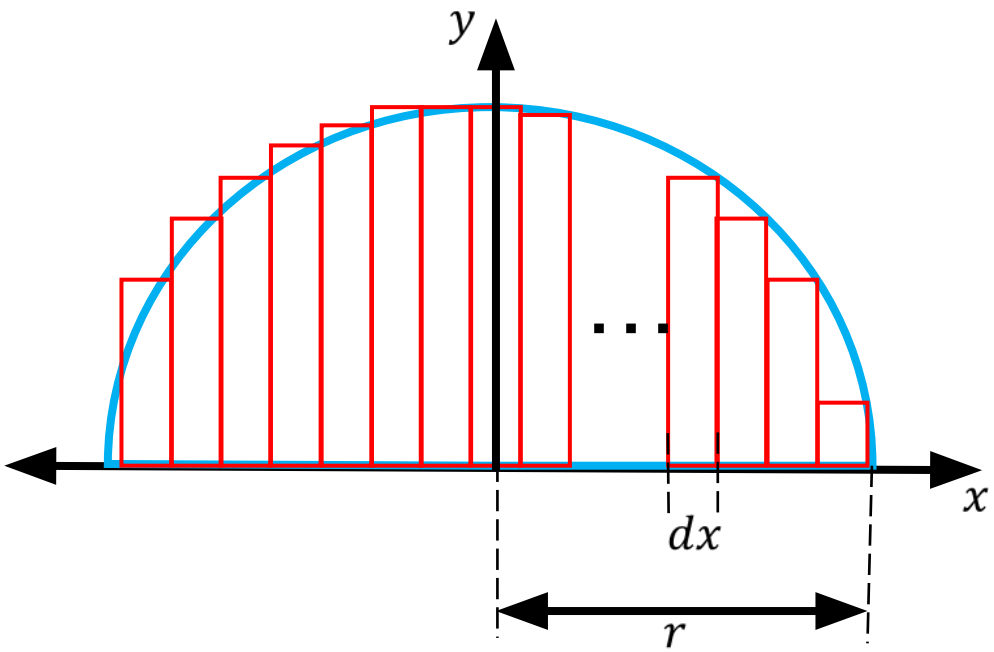
```
double pi (unsigned long long niters)
{
    double res;
    double h = 1.0 / (double) niters;
#define MAX_PARALLEL_RECURSIVE_LEVEL 4
    res = pi_r (h, 0, MAX_PARALLEL_RECURSIVE_LEVEL, 1, niters);
    return res * h;
}

int main (int argc, char *argv[])
{
    double tstart = omp_get_wtime();
    int NITERS = atoi(argv[1]);
    printf ("Number of rectangles: %d\n",NITERS);

    double pi_approx = pi(NITERS);

    printf ("PI (w/%d iters) is %15.13f\n", NITERS, pi_approx);
    printf ("Accuracy for %d \titers is :: %10.7f % \n", NITERS, pi_approx/M_PI*100);
    printf ("runtime is: %lf\n", omp_get_wtime()-tstart);
    printf ("\n");
    return 0;
}
```

Assignment: Compute pi with OpenMP tasks

1. Parallelize the compute pi code with openMP
2. Use openMP tasks on the recursive function pi_r
3. Once the code has been parallelized, perform the following:
   a) Compute pi with [100,500,1000,5000,10000,1000000,100000000] numbers of rectangles for both the serial and parallelized code with 2 and 4 threads. Note the runtime and accuracy of the approximation in each.
   b) Comment on trends you see with the runtime when increasing the number of rectangles and the number of threads.

$$\frac{\lim_{n\to\infty} \sum_n h[n] \times dx}{r^2} \approx \pi$$

$$h[n] = (r^2 - x[n]^2)^{1/2}$$



References and Additional resources:

OpenMP Tasking :

- https://openmp.org/wp-content/uploads/sc13.tasking.ruud.pdf
- http://icl.cs.utk.edu/classes/cosc462/2017/pdf/W43%20-%20OpenMP%20Tasking.pdf
- https://en.wikibooks.org/wiki/OpenMP/Tasks

OpenMP tasking model (advanced):

- https://www.slideshare.net/InformaticaUCM/openmp-tasking-model-from-the-standard-to-the-classroom?from_action=save

A nice book on OpenMP in general, many many examples:

- https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.Examples.pdf

OpenMP API User's Guide:

- https://docs.oracle.com/cd/E77782_01/html/E77801/gljyr.html#scrolltoc (2017 documentation)
- https://docs.oracle.com/cd/E19205-01/820-7883/auto15/index.html (2010 documentation)

OpenMP directives:

- https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-directives?view=vs-2019

Basics of recursion

- https://www.programiz.com/c-programming/c-recursion
- https://www.cs.utah.edu/~germain/PPS/Topics/recursion.html

Runtime Determinacy Race Detection for OpenMP Tasks:

- https://link.springer.com/chapter/10.1007/978-3-319-96983-1_3