# Finding Primes

Sieve of Eratosthenes

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 |
| 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 |
| 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 |
| 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 |
| 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |
| 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 |

**Prime numbers**

# Timing

How fast is your code?  How long to find the primes between 1 and 10,000,000? How long to find primes between 1 and 1  billion?

Optimization:  the C++ compilers have optimization switches.
g++ filename.cpp -O3  will optimize the code and give significant speed ups with no effort on your part.

```
//timing routines
#include  <time.h>
#include <sys/time.h>
#include <iostream>

using namespace std;
//function for elapsed time
double getTimeElapsed(struct timeval end, struct timeval start)
   {
       return (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1000000.00;
   }
int main()
{
 // timing info variables in main
   struct timeval t0, t1;
   double htime;

    gettimeofday(&t0, NULL);
    //computations to do here
    gettimeofday(&t1, NULL);

    // calculate elapsed time and print
    htime=getTimeElapsed(t1,t0);
       cout<<"time for computation: "<<htime<<endl;
  }
```

# Multi-threaded Applications

**Multi-core computers**

Emphasize multiple full-blown processor cores, implementing the complete instruction set of the CPU

The cores are out-of-order implying that they could be doing different tasks

They may additionally support hyperthreading with two hardware threads

Designed to maximize the execution speed of sequential programs

**Approaches**

Posix threads (pthreads; 1995)

OpenMP (1997)

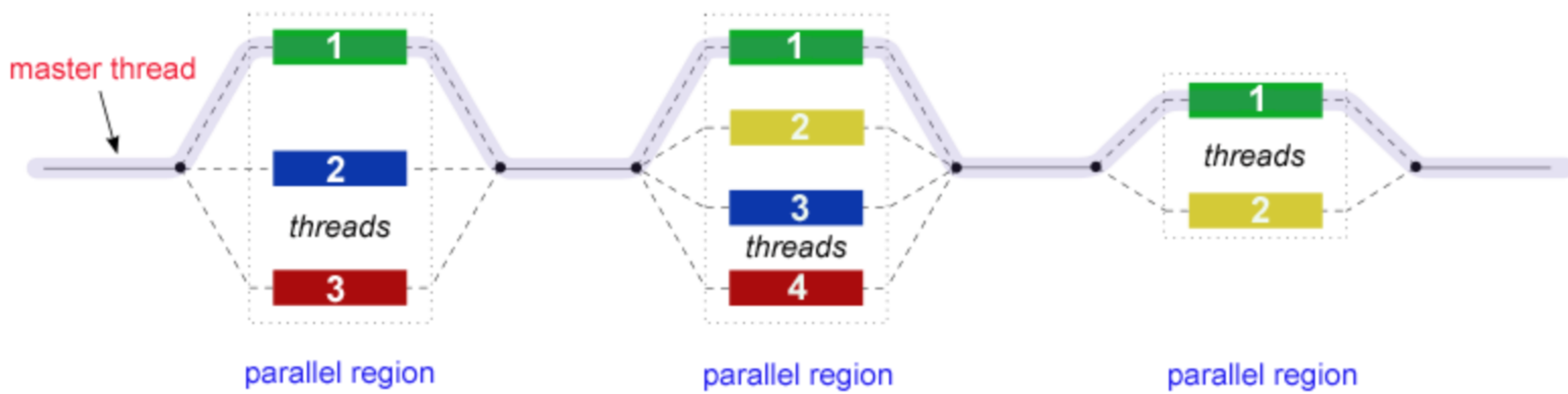
# Multi-threaded/multiprocessor Approach

## Primes

1. One thread or processor per number: half threads gone after one step
2. Thread 0 responsible for 2, 2+p, 2+2p, …
   Thread 1 responsible for 3, 3+p, 3+2p, ..
   etc. where p is the number of processors
3. Divide numbers between processors. Each processor does N/p numbers

A thread is an independent set of instructions that can be scheduled by the processor.

# OpenMP Fork and Join Model



# Try OpenMP: Hello, World

Type, save as hello.cpp, compile, and run*:

```
#include <iostream>
using namespace std;
int main()
{

    int ID = 0;
    printf("Hello, World (%d)\n",ID) ;

}
```

* /usr/local/bin/g++  hello.cpp

# Now with openMP

Add the openMP terms, compile, and run*:

```
#include <omp.h>
#include <iostream>
using namespace std;
int main()
{
 #pragma omp parallel
  {
    int ID = omp_get_thread_num();
    printf("Hello, World (%d)\n",ID) ;
    }
}
```

* /usr/local/bin/g++  hello.cpp -fopenmp

# C++ vectorAdd.cpp

```
#include <iostream>
#define SIZE 1024
using namespace std;

void vectorAdd(int *a, int *b, int *c, int n)
 {
   for (int i = 1; i < n; i++)
     c[i] = a[i] + b[i];
  }

int main()
 {
   int * a;
   int * b;
   int * c;
   a = new int[SIZE];                    //declare dynamic array
   b = new int[SIZE];
   c = new int[SIZE];
   for (int i = 0; i < SIZE; i++)     //initialize vectors
    {
      a[i]=i;
      b[i]=i;
      c[i]=0;
      }
```

# C++ vectorAdd.cpp

```
vectorAdd(a,b,c, SIZE);           //call function

 // print out first 10 numbers c[i]
   for(int i = 0; i<10; i++)
    cout<< "c[" <<i<<"] = "<<c[i]<<endl;

    delete [] a;                          //free array memory
    delete [] b;
    delete [] c;
}
```

**To make parallel:** add OpenMP directives

```
#include <iostream>
#include <omp.h>              //1. include the openmp header
#define SIZE 1024
using namespace std;

void vectorAdd(int *a, int *b, int *c, int n)
 {
    #pragma omp parallel      //2. pragma omp parallel creates team of parallel threads
    {
    #prama omp for            // 3. pragma imp for means each thread handles a
    for (int i = 1; i < n; i++)      different portion of the loop.
       c[i] = a[i] + b[i];
    }
  }

int main()
 {
   int * a;
   int * b;
   int * c;
   a = new int[SIZE];
   b = new int[SIZE];
   c = new int[SIZE];
   for (int i = 0; i < SIZE; i++)    //initialize vectors
    {
```

# C++ vectorAdd.cpp

# Alternative Method: vectorAddInline

```
inline void vectorAdd(int *a, int *b, int *c, int n)
{    for (int i = 1; i < n; i++)
       c[i] = a[i] + b[i];
}
int main()
{
    int * a;
   int * b;
   int * c;
   a = new int[SIZE];
   b = new int[SIZE];
   c = new int[SIZE];

   #pragma omp parallel
   {
     #pragma omp for
     for (int i = 0; i < SIZE; i++)    //initialize vectors in parallel
      {
        a[i]=i;
        b[i]=i;
        c[i]=0;
        }
    vectorAdd(a,b,c, SIZE);
     }   //end parallel section
```

# Parallel Computing

**Embarassingly Parallel** (the best problems)

These are problems that can divided up among workers readily.

Example: find a owner of a random number in a Maryland phone book: xxx-xxxx.
One person: start at page 1 and scan book until the end.
100 people: each scans 1/100 of the book. 100 times faster (avg)

**Data Parallelism** is a program property where many arithmetic operations can be performed on the data simultaneously.

Simulations with many data points and many time steps. Matrix multiplication:
row x column for all rows and columns; each independent.

# OpenMP

A portable threading model to use all processors on a multi-core chip
Laptop: 4 cores; Mac Pro: up to 12 cores; Intel 60 and 80 cores
ClearSpeed 96 cores

Available for many compilers and for FORTRAN, C, C++

Uses **#pragma** commands (pre-processor commands)

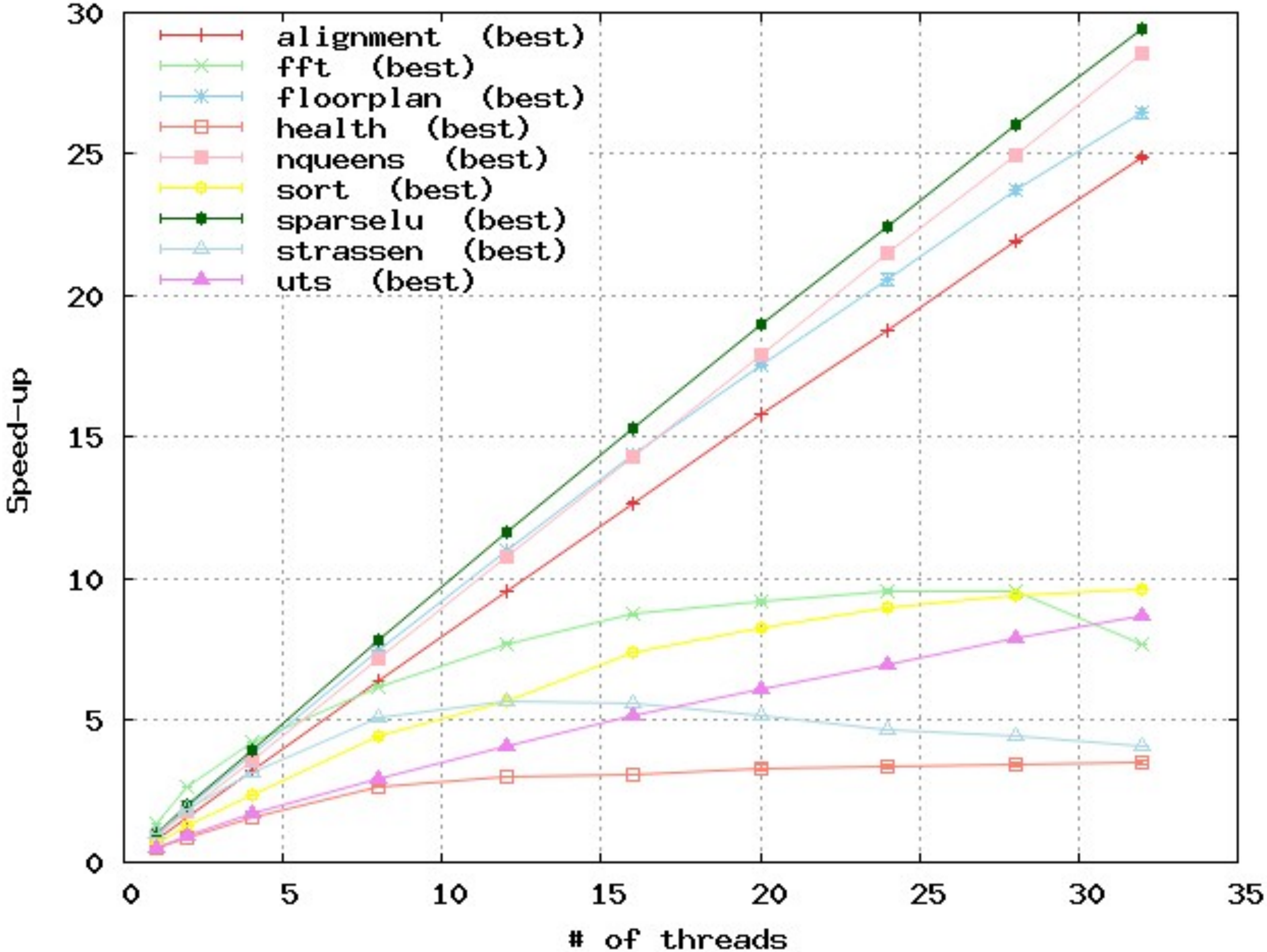
**Easy to use:** can make a loop parallel with one command

/usr/local/bin/g++ file.cc -fopenmp -O3 -o file

Reference for OpenMP 4.0:
http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

# A Test Suite for OpenMP

Barcelona OpenMP Test Suite Project




https://pm.bsc.es/projects/bots

# Tests

| *Name* | *Origin* | *Domain* | *Summary* |
|---|---|---|---|
| Alignment | AKM | Dynamic programming | Aligns sequences of proteins |
| FFT | Cilk | Spectral method | Computes a Fast Fourier Transformation |
| Floorplan | AKM | Optimization | Computes the optimal placement of cells in a floorplan |
| Health | Olden | Simulation | Simulates a country health system |
| NQueens | Cilk | Search | Finds solutions of the N Queens problem |
| Sort | Cilk | Integer sorting | Uses a mixture of sorting algorithms to sort a vector |
| SparseLU | - | Sparse linear algebra | Computes the LU factorization of a sparse matrix |
| Strassen | Cilk | Dense linear algebra | Computes a matrix multiply with Strassen's method |
| UTS | UNC/OSU/UMD* | Search | Computes the number of nodes in an Unbalanced Tree |

- University of North Carolina, the Ohio State University and the University of Maryland

# Speed-up

If there are N threads or processors and the problem was 100% parallelizable, then each processor does 1/N of the problem and the total computational time is proportional to 1/N. Calculate speed-up, S:

$$S = \frac{\text{Time for 1 processor, } T(1)}{\text{Time for N processors, } T(N)} = N$$

**This is linear speed-up**

If the program consists of parts that can only be done by serial execution and parts in parallel, then we can divide the execution time into $T = T_s$ and $T_p$. Therefore the above relationship for this type of program is

$$S = \frac{T_s + T_p}{T_s + T_p/N}$$

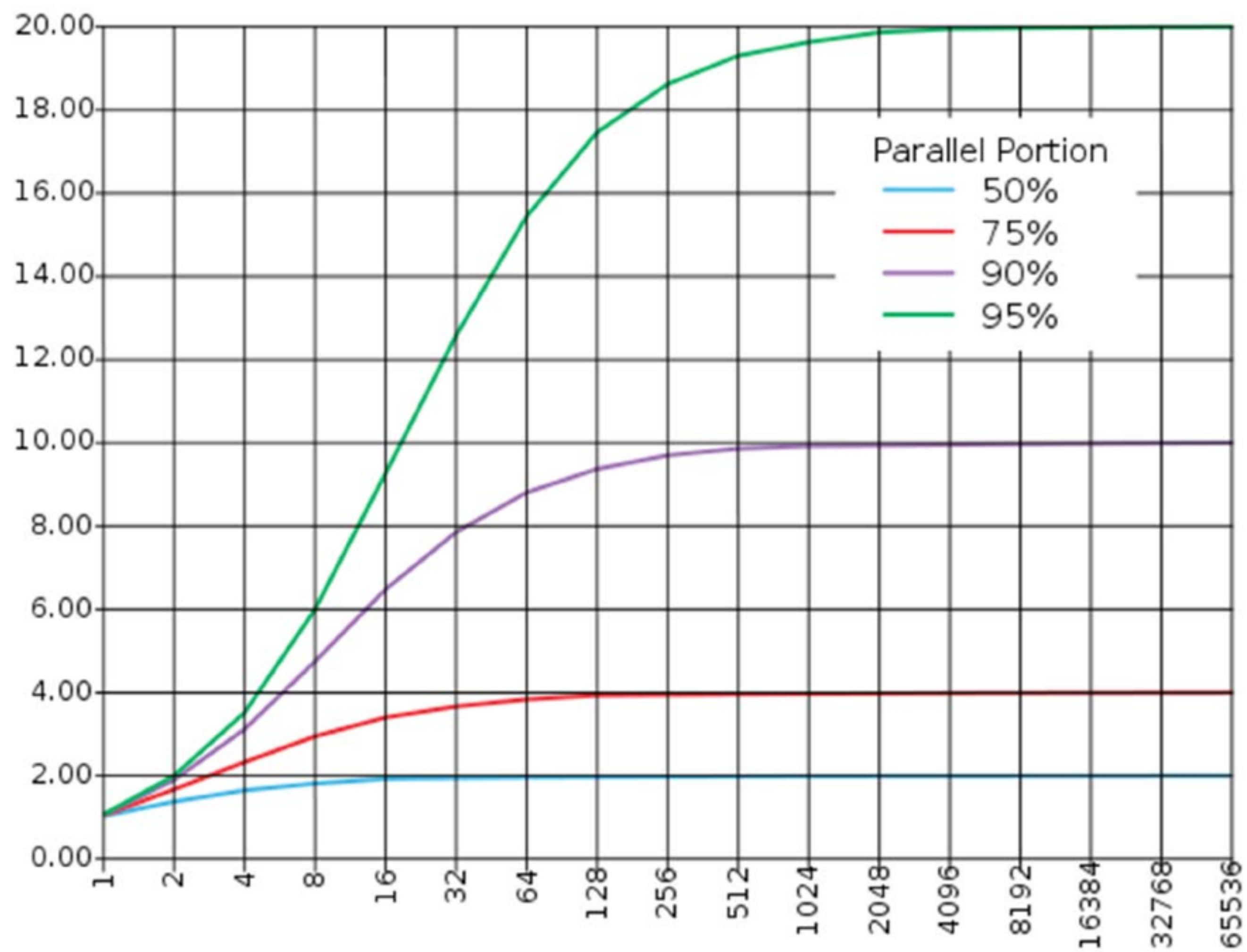
Amdahl's Law

# Amdahl’s Law

If a given fraction, F, of a program can be run parallel, then the non-parallel part is 1-F.

$$S = \frac{(1-F)P + FP}{(1-F)P + \frac{F}{N}P} = \frac{1}{1-(1-\frac{1}{N})F}$$

Speed-up



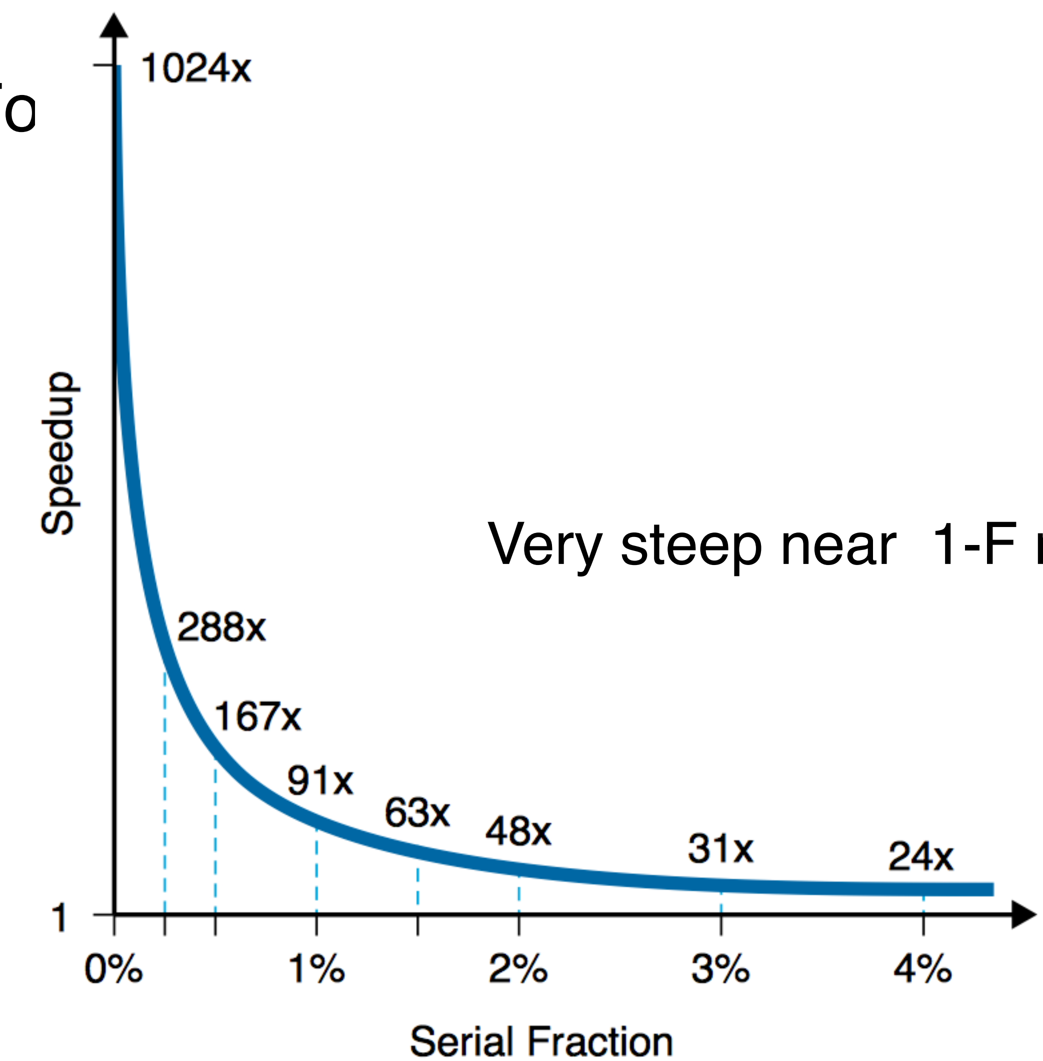

Processors

# Hard to get speed ups

$$S = \frac{1}{1-(1-\frac{1}{N})F}$$

If 20 processors, but only 20% of program is in parallel form, then Amdahl’s law says S=1.23.

Fo … parallel code




Very steep near  1-F near zero.

# Gustafson/Barsis Law

Amdahl Law for a fixed problem, but as problems grow in size, then efficiency grows. That is, problems we do grow with the equipment that we have. Also the serial part of the code does not grow that much.

If s and p represent the amount of time computing on a parallel machine, then the serial machine will take s+N P to do the task.

$$S = \frac{s+Np}{s+p} = N + (1-N)s, \text{ given s+p scaled to 1.}$$

Amdahl: scaled on time; fixed problem size; G/B scaled on N of processors

S= s + N p

# Exploitable Concurrency

Most programs have parts of the code that can be decomposed into subproblems that can be computed independently.

You have to identify these parts and recognize any dependencies that might lead to race conditions.

Procedure:

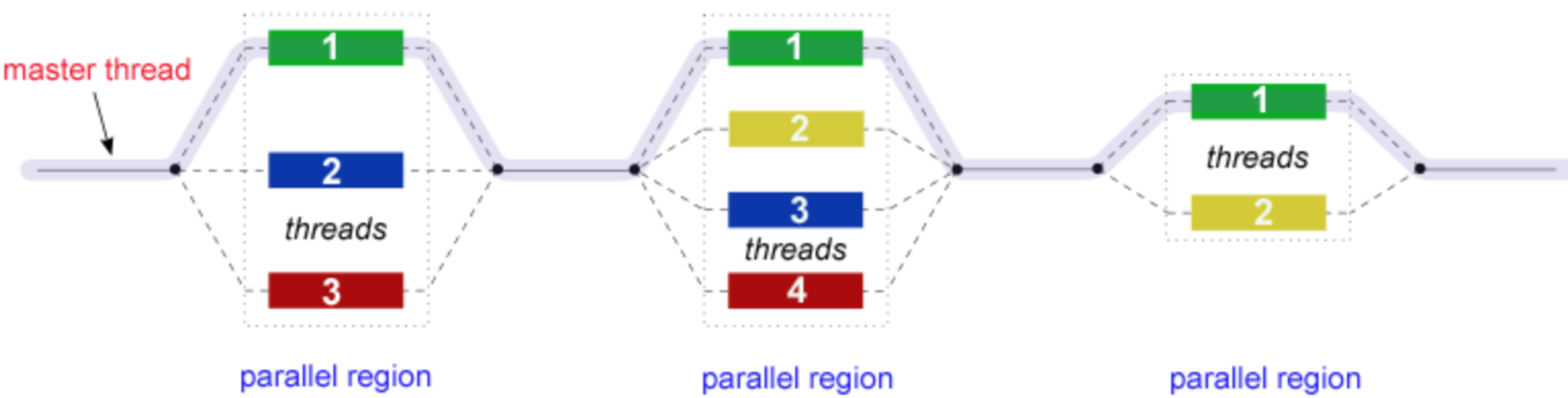
- Find Concurrency
- Design parallel algorithms to exploit concurrency

Mattson, T et al. Patterns for Parallel Programming

# OpenMP: Open Multi-Processing

Multi-threaded shared memory parallelism
C, C++, FORTRAN, since 1997.

Schematic of code: **Fork and Join Model**




No jumping out or into parallel sections.

Uses environmental variables for max no. of threads
    setenv OMP_NUM_THREADS 8
or  export OMP_NUM_THREADS 8
    OMP_WAIT_POLICY ACTIVEIPASSIVE

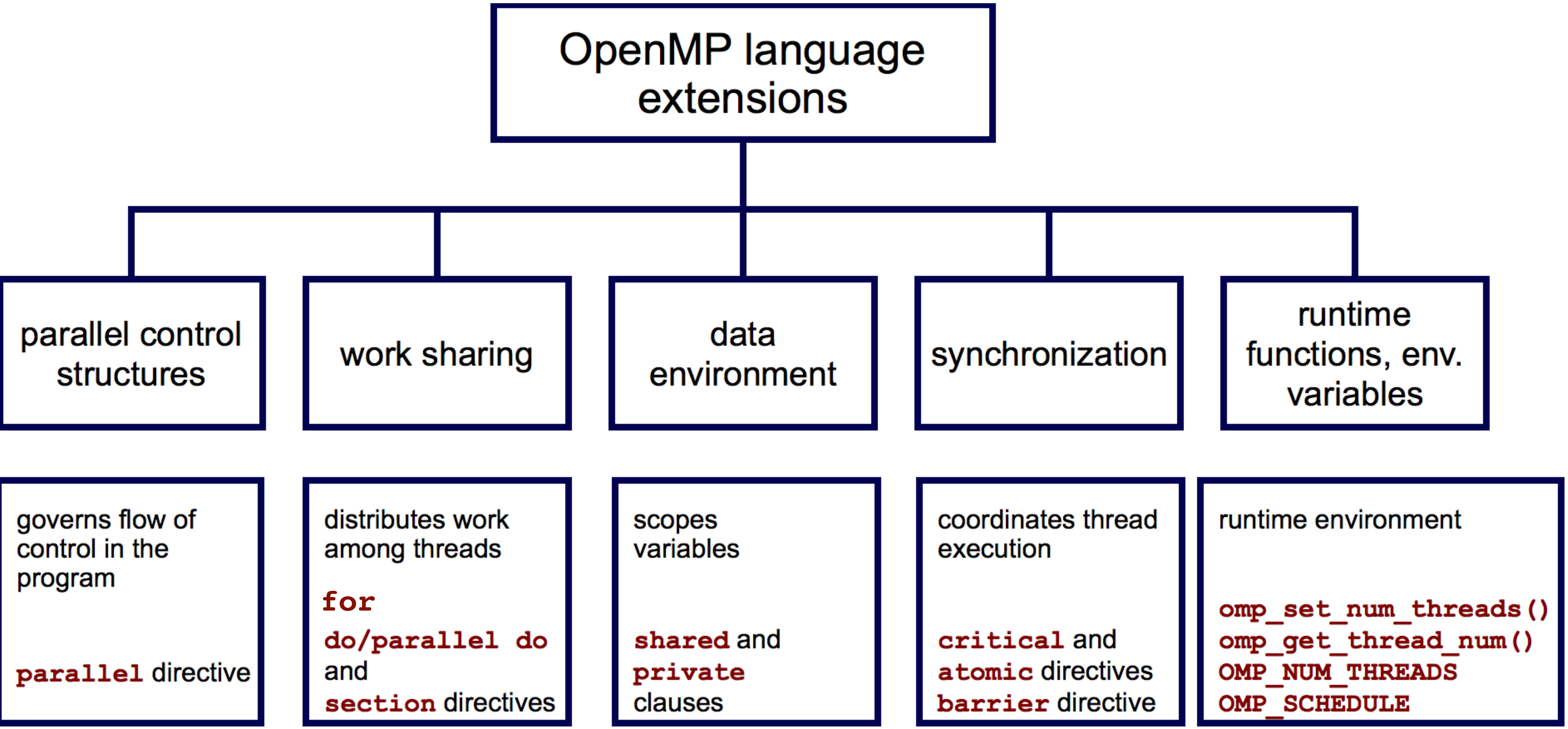
# Procedure

When a thread reaches a **parallel** directive, it creates a team of threads and becomes the master thread, with thread number 0.

The code in the parallel region is duplicated and all threads execute that code

There is an implied barrier at the end of the parallel region, only master thread continues.  All other threads terminate.

Number of threads dictated by env variable: OMP_NUM_THREADS
or by omp_set_num_threads()


OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
governs flow of control in the program
parallel directive
distributes work among threads
for
do/parallel do and
section directives
scopes variables
shared and private clauses
coordinates thread execution
critical and atomic directives
barrier directive
runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

# Generic OpenMP Structure

```
#include <omp.h>
main ()  {
int var1, var2, var3;
Serial code here
      .
      .
Beginning of parallel section. Fork a team of threads while
specifing variable scoping

#pragma omp parallel private(var1, var2) shared(var3)
  {
    Creates a team of threads and each thread executes the same code
    Other OpenMP directives may be here as well as
    run-time library calls
    omp_get_thread_num();
    All variables here are local to the block
    private (var 1, var2) are declared earlier, but have no initial value/leave with none
    shared (var3) means var3 value known to all threads
    At end, all threads join master thread and disband
  }

Resume serial code
}
```

# OpenMP pragmas

**#pragma omp parallel**

{

stuff done with threads

}

Inside parallel block, additional commands

**#pragma omp single**

{

done by one thread

}

**#pragma omp for** distribute the for loop among the existing team of threads

```c
int main(int argc, char *argv[])
{
    const int N = 100000;
    int i, a[N];

    #pragma omp parallel for
    for (i = 0; i < N; i++)
        a[i] = 2 * i;

    return 0;
}
```

# Parallel Programming Pitfalls

Threads can change variables and lead to **race conditions**

A race condition occurs when two threads access a shared variable at the same time. The first thread reads the variable, and the second thread reads the same value from the variable. Then the first thread and second thread perform their operations on the value, and they race to see which thread can write the value last to the shared variable. The value of the thread that writes its value last is preserved, because that thread is writing over the value that the previous thread wrote.

Solve by controlling access to shared variables. Compare two possible outcomes for the case where two threads want to increment a global variable x by 1. Both can happen, depending on the processor:

Start: x=0
1. T1 reads x=0
2. T1 calculates x=0+1=1
3. T1 writes x=1
4. T2 reads x=1
5. T2 calculates x=1+1=2
6. T2 writes x=2

Result: x=2

Start: x=0
1. T1 reads x=0
2. T2 reads x=0
3. T1 calculates x=0+1=1
4. T2 calculates x=0+1=1
5. T1 writes x=1
6. T2 writes x=1

Result: x=1

# OpenMP and Functions

**Loops <u>including function calls</u> are not usually parallelized unless the compiler can guarantee there are no dependencies between iterations.**

A work-around is to **inline** the function call within the loop, which works if no dependencies.

```
#include <iostream>
using namespace std;

inline void hello()
{
  cout<<"hello";
}
int main()
{
  hello(); //Call it like a normal function...
  cin.get();
}
```

The compiler writes the function code directly within the main loop.

# Barrier

```
#include <stdio.h>
#include <omp.h>
int main(){
  int x;
x = 2;
#pragma omp parallel num_threads(4) shared(x)
{
  if (omp_get_thread_num() == 0) {
     x = 5;
  } else {
  /* Print 1: the following read of x has a race */
    printf("1: Thread# %d: x = %d\n", omp_get_thread_num(),x );
  }
  #pragma omp barrier
  if (omp_get_thread_num() == 0) {
  /* Print 2 */
    printf("2: Thread# %d: x = %d\n", omp_get_thread_num(),x );
  } else {
  /* Print 3 */
    printf("3: Thread# %d: x = %d\n", omp_get_thread_num(),x );
} }
return 0; }
```

# Single

```
#include <stdio.h>
#include <omp.h>
int main(){
  int x;
x = 2;
#pragma omp parallel num_threads(4) shared(x)
{
  #pragma omp single
  {
     x = 5;
  }

    printf("1: Thread# %d: x = %d\n", omp_get_thread_num(),x );

  if (omp_get_thread_num() == 0) {
  /* Print 2 */
    printf("2: Thread# %d: x = %d\n", omp_get_thread_num(),x );
  } else {
  /* Print 3 */
    printf("3: Thread# %d: x = %d\n", omp_get_thread_num(),x );
} }
return 0;
}
```