Compilers for Embedded Systems

Integrated Systems of Hardware and Software

OpenMP Programming

Dr. Vasilios Kelefouras

Email: v.kelefouras@plymouth.ac.uk
Website: https://www.plymouth.ac.uk/staff/vasilios-kelefouras

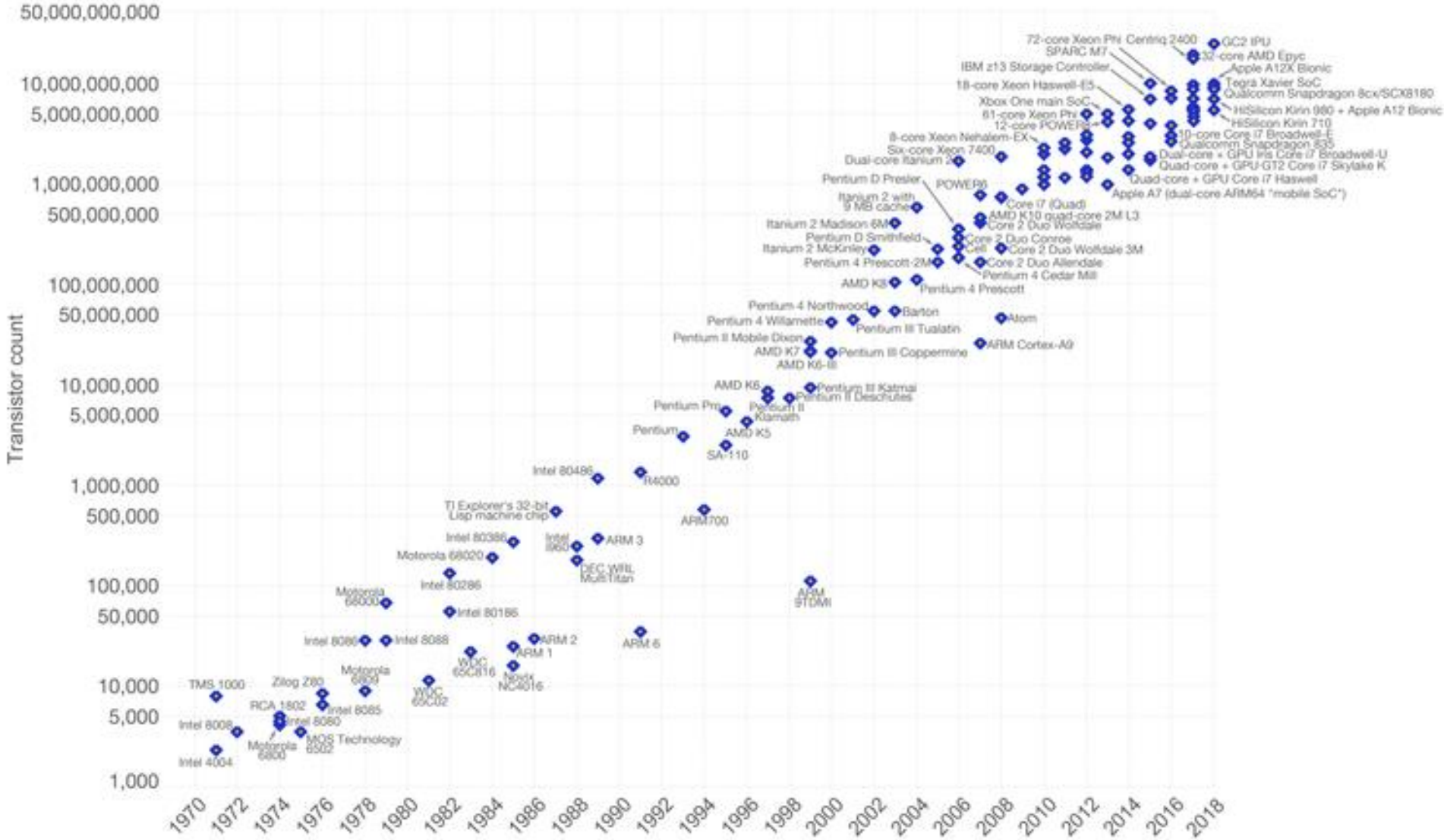
School of Computing
(University of Plymouth)

Outline

- Why Parallel computing?
- Concurrency vs parallelism
- Introduction to OpenMP
- OpenMP examples
  - Array addition
  - Array summation
  - Pi calculation

Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

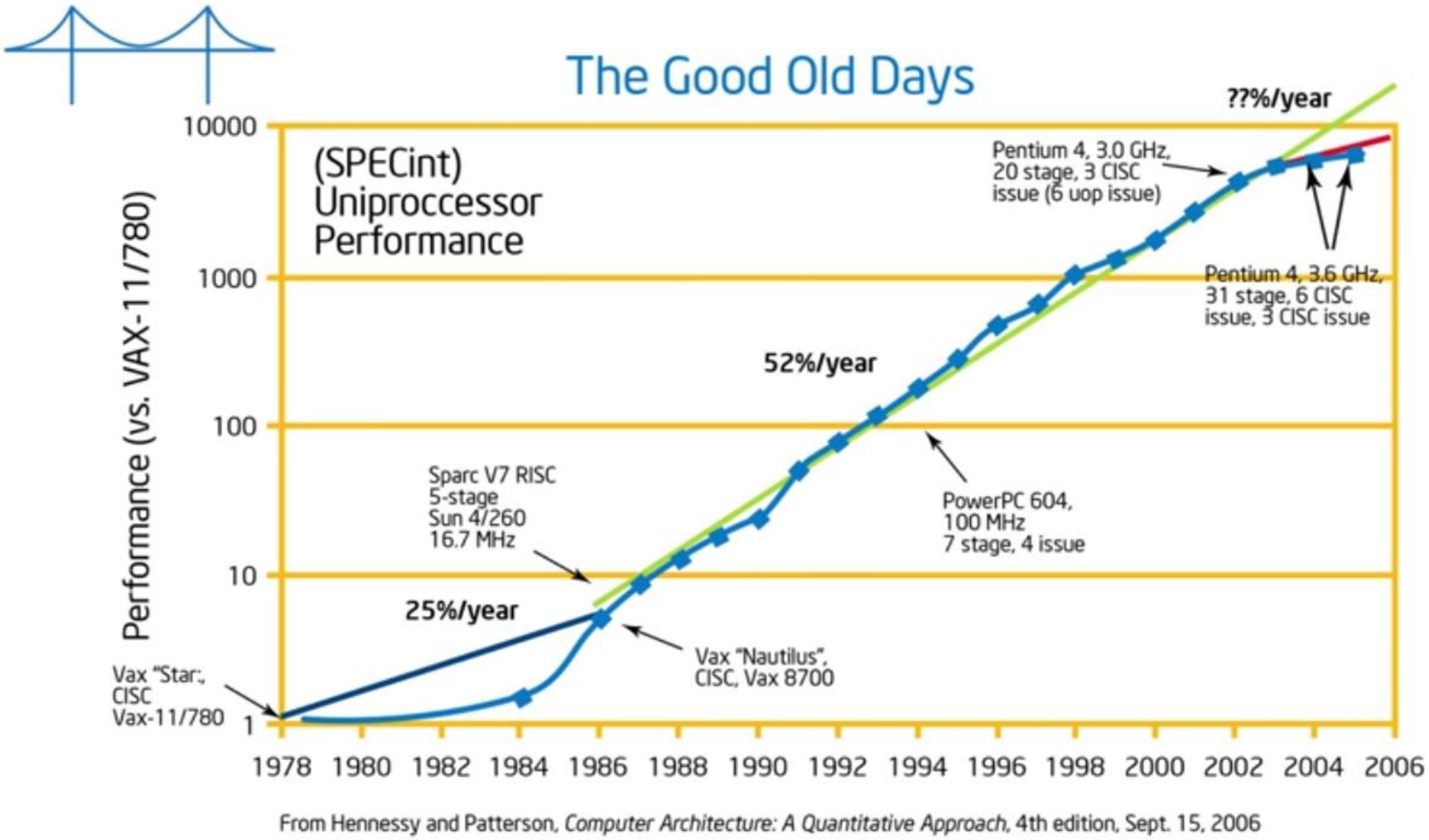
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)

The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Performance **used to** increased according to the number of transistors (1)



From Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 4th edition, Sept. 15, 2006

Performance **used to** increased according to the number of transistors (2)

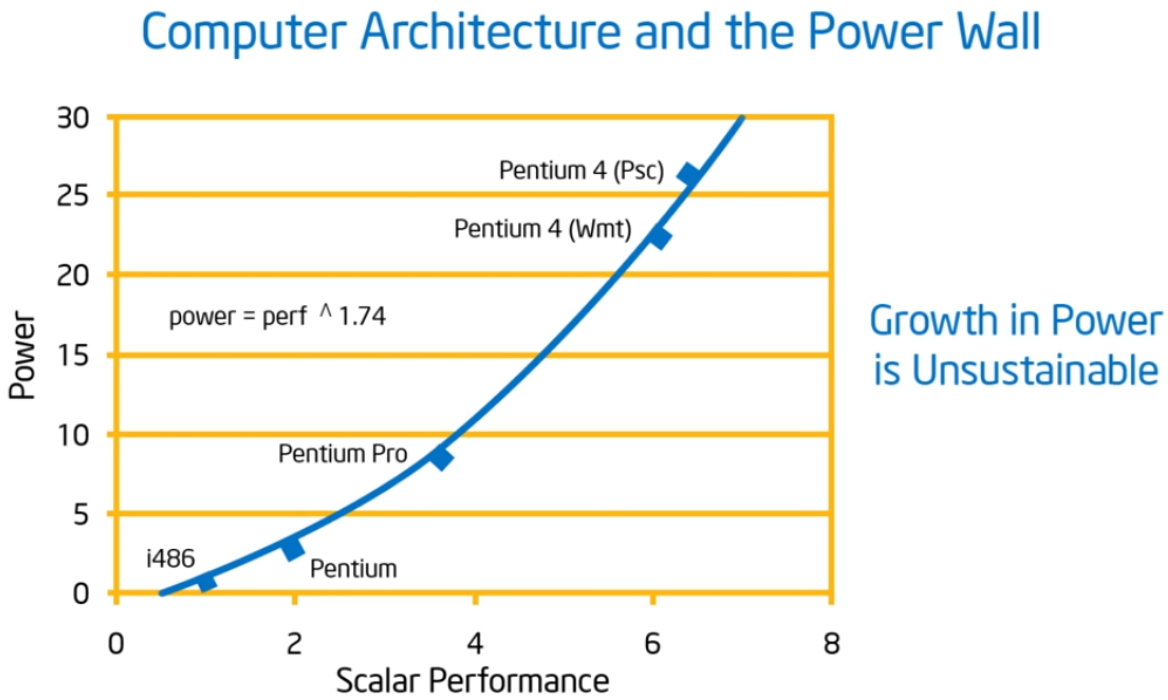
- The fact that performance used to increase by increasing the number of transistors, made people to believe that performance comes from the hardware
- Programmers used to write software without thinking about performance
- They counted on the hardware to do the work
- This model used to work fine…but…back in 2006, something changed…
  - ***The Power Wall problem***

Performance **used to** increased according to the number of transistors (3)

- Power Wall Problem
  - The CPU design goal for the late 1990's and early 2000's was to increase the CPU frequency.
    - This was a way to improve system performance
    - This was done by adding more transistors to a smaller chip.
    - However, this increased the power dissipation of the CPU chip beyond the capacity of inexpensive cooling techniques.
    - The last years the ***CPU frequency has ceased to grow***

Performance **used to** increased according to the number of transistors (4)

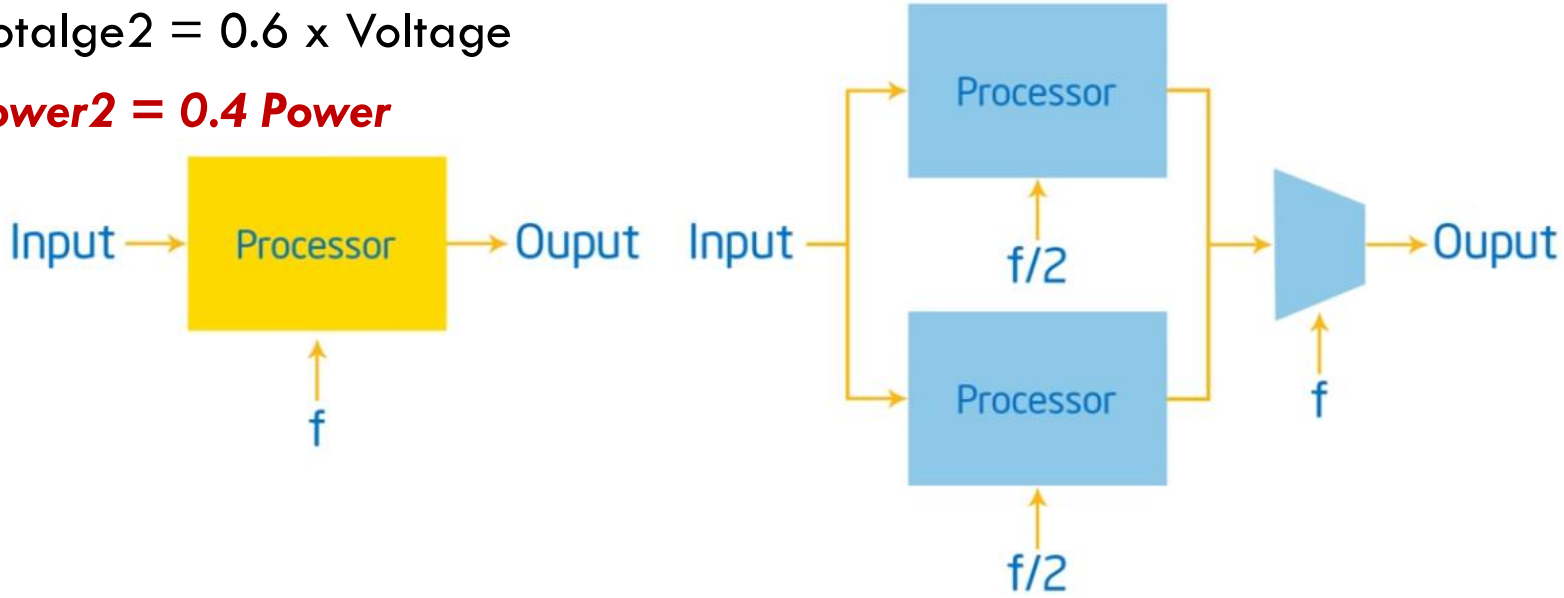
- **By increasing performance, power consumption increases even more** (next slide)
- This is not sustainable
- What to do? ***The solution is Parallel hardware architectures***



Source: E. Grochowski of Intel

The solution to the Power Wall Problem

- **Power=Capacitance x Voltage x Frequency^2**
- By using two processors inside the same chip, with half the frequency each, then:
  - Capacitance2 = 2.2 x Capacitance
  - Frequency2 = F/2
  - Votalge2 = 0.6 x Voltage
  - ***Power2 = 0.4 Power***



Parallel computing gives us the ability to give the same performance with lower power

Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, R.W., "Optimizing power using transformations," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,, vol.14, no.1, pp.12-31, Jan 1995

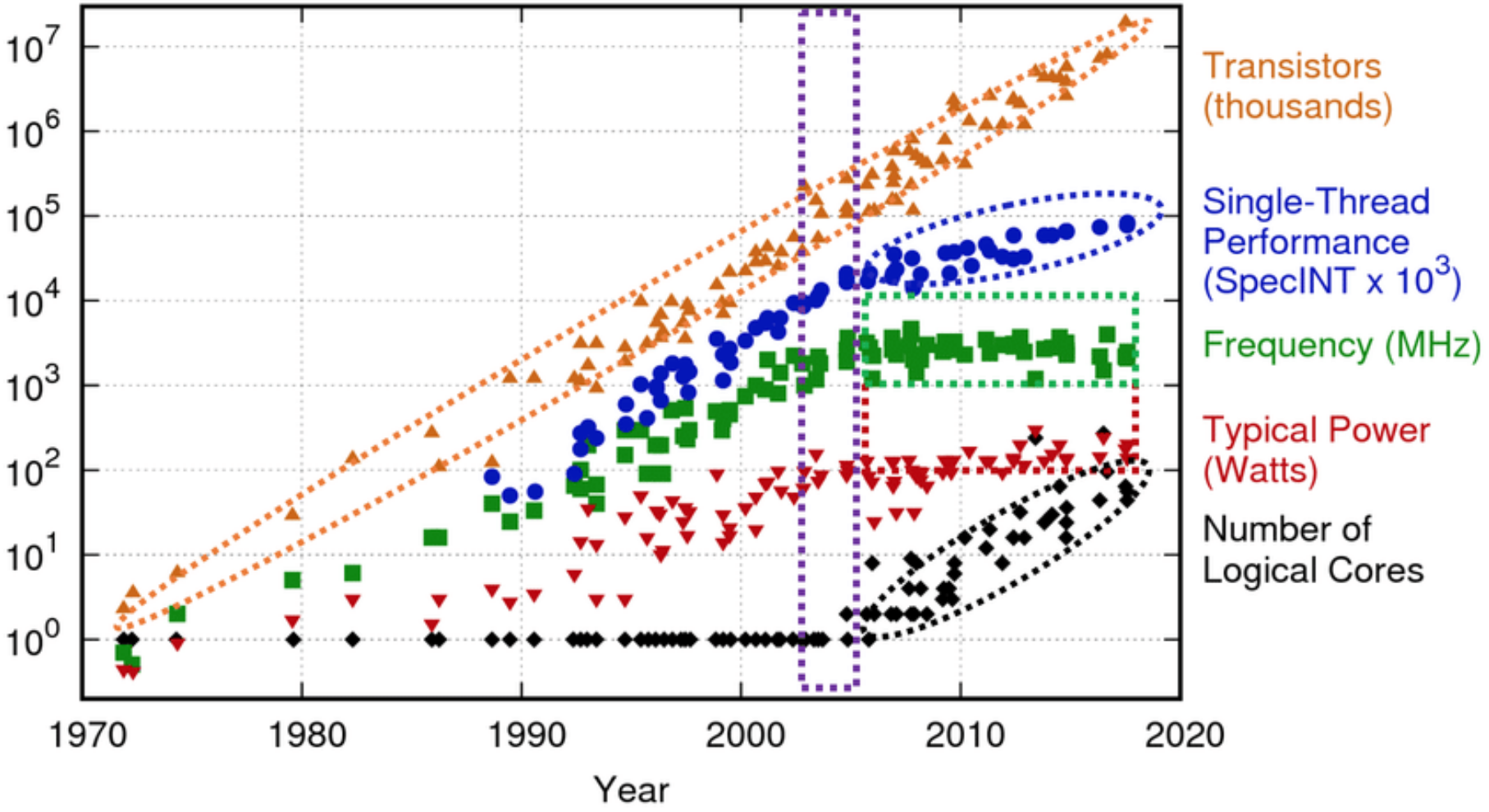
Source: Vishwani Agrawal

The Era of Parallel Computing is here

- Performance comes from the software
- There are no smart-enough tools to efficiently parallelize serial software on the parallel hardware
- Free lunch is over…

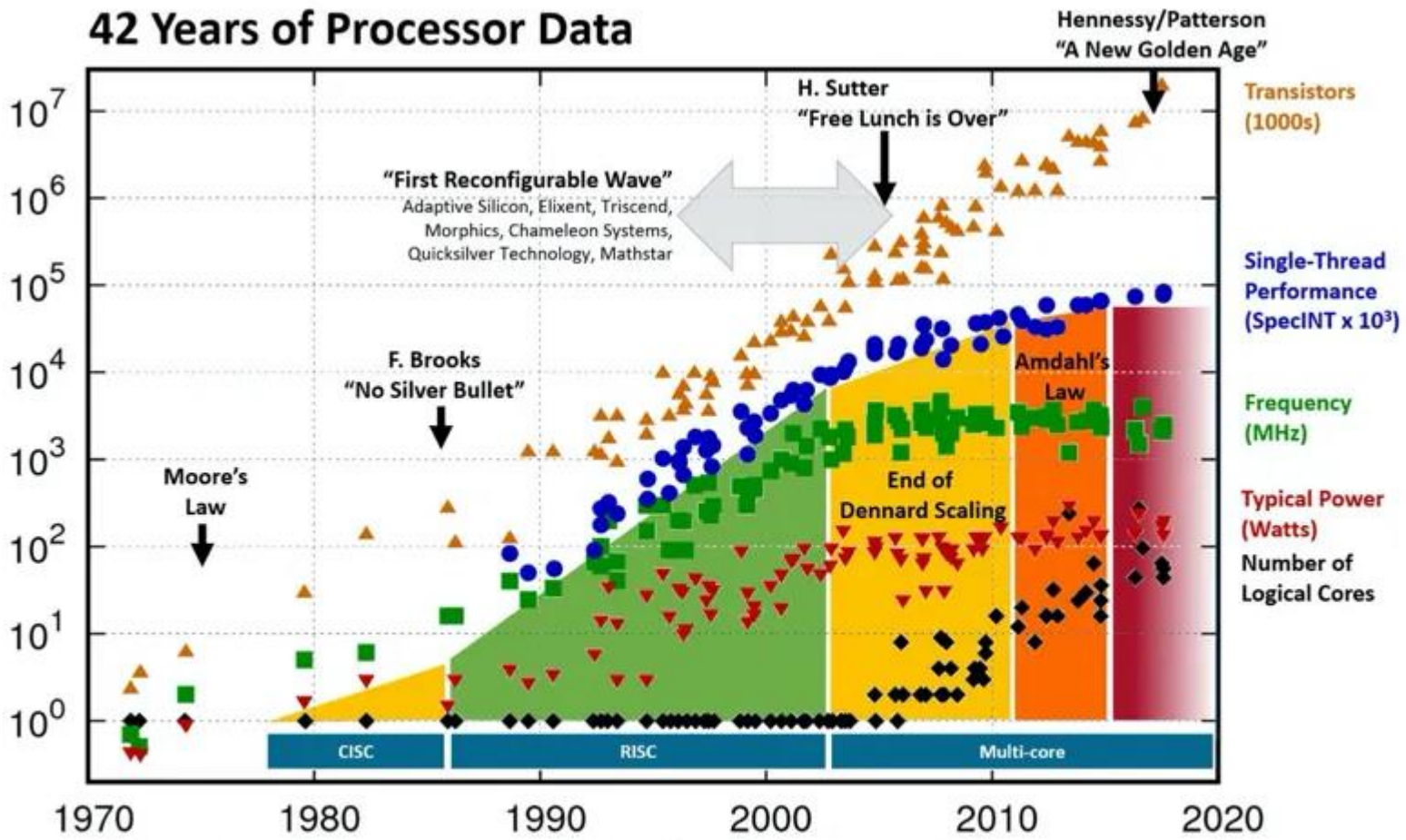
- We must learn how to write parallel applications…

Hardware Architecture Trends (1)



Taken from https://www.researchgate.net/publication/336577121_BACKUS_Comprehensive_High-Performance_Research_Software_Engineering_Approach_for_Simulations_in_Supercomputing_Systems

Hardware Architecture Trends (2)

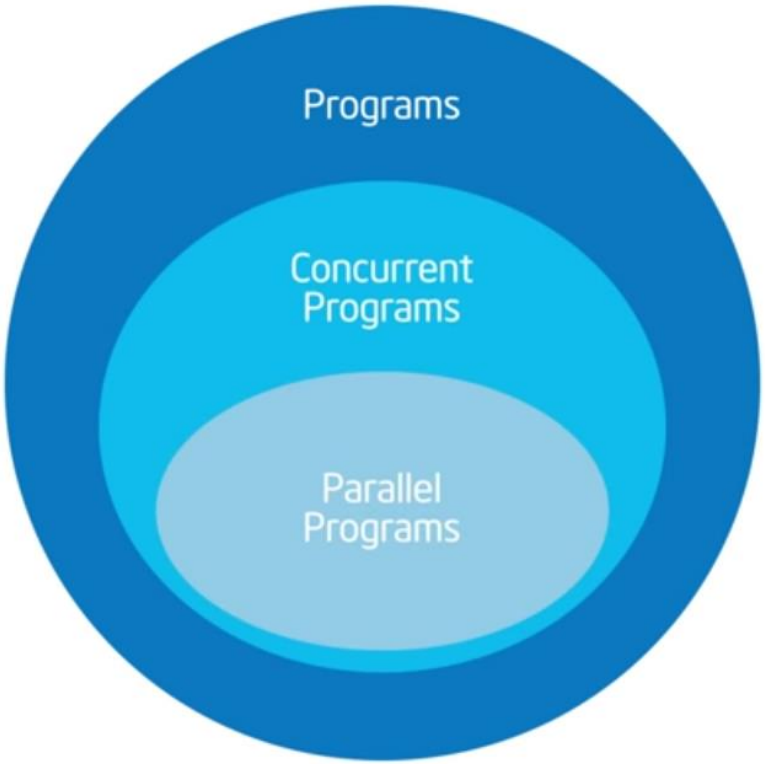


Hennessy and Patterson, Turing Lecture 2018, overlaid over "42 Years of Processors Data"
https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/;"First Wave" added by Les Wilson, Frank Schirrmeister
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Concurrency vs Parallelism (1)

- ***Concurrency***: A condition of a system in which multiple tasks are **logically active** at one time
  - Multiple things can happen at the same time
  - They may not be happening but they can
- ***Parallelism***: A condition of a system in which multiple tasks are **actually active** at one time
  - A subset of concurrency
- ***Concurrent application***: An application for which computations logically execute simultaneously
- ***Parallel application***: An application for which computations actually execute simultaneously in order to compute a problem faster

Concurrency vs Parallelism (2)



Taken from https://www.youtube.com/watch?v=6jFkNjhJ-Z4&list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG&index=3

Concurrency vs Parallelism (2)

Steps:

1. We need to find the concurrency in our application
   - This process cannot be automated
2. Then we decide which parallel programming language (OpenMP, MPI, etc) to use
   - This is the easy part
   - The hard part is to find the concurrency and understand the algorithmic strategy

- We must expose the concurrency of a problem and map it on the parallel hardware so it can be executed in parallel

What is OpenMP? (1)

- ***OpenMP (open multi-processing)*** : An API for writing multithreaded apps
- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multithreaded programs in Fortran, C/C++

What is OpenMP? (2)

- See how **easy** it is to parallelize applications using OpenMP

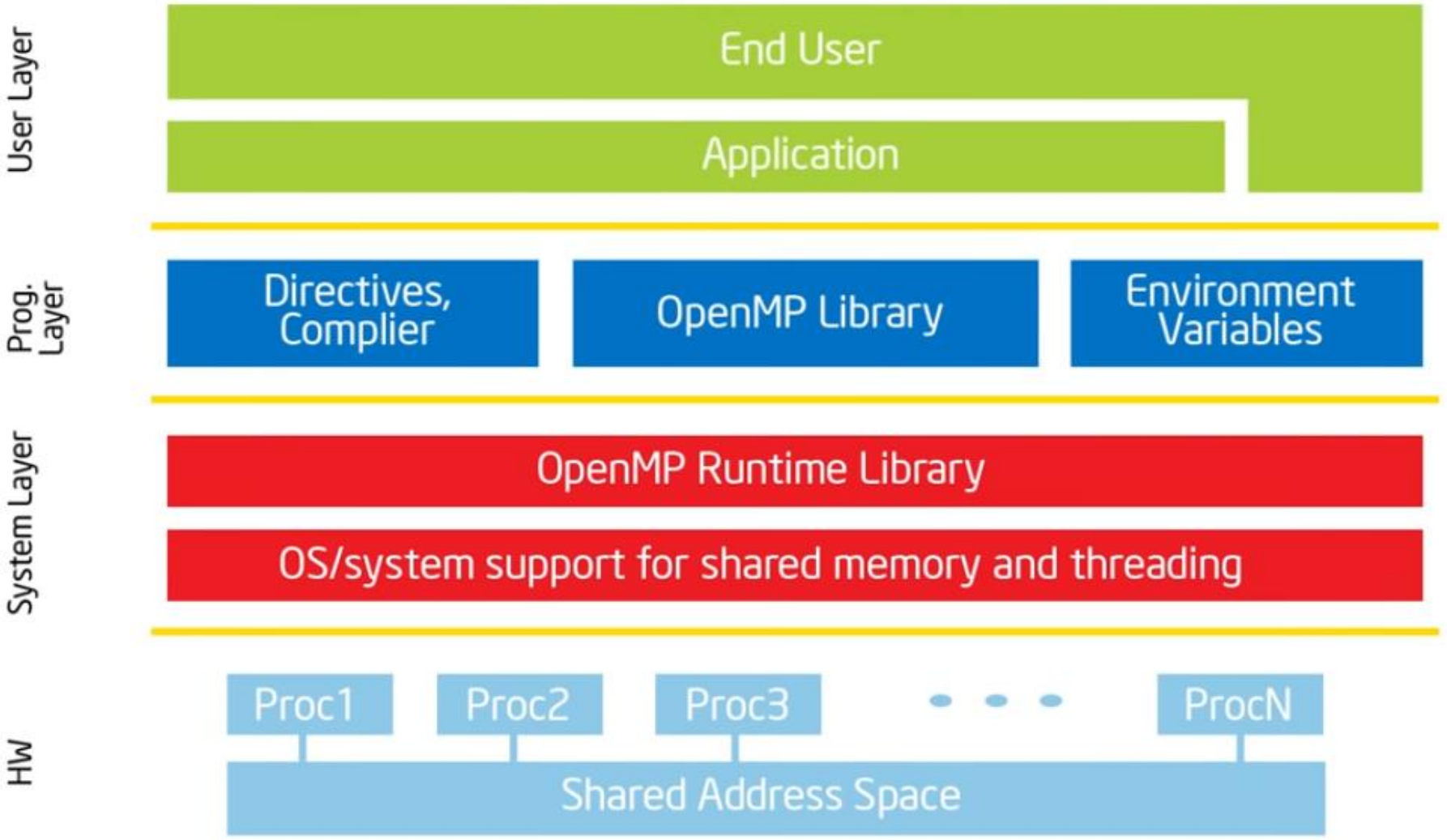
```
#pragma omp parallel for private(i, j, k, tmp)
 for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
   tmp = 0.0;
  for (k = 0; k < N; k++) {
    tmp += A[N * i + k] * B[N * k + j];
    }
 test[N * i + j] = tmp;
 }
}
```

What is OpenMP? (3)

- See how easy it is to parallelize and vectorize applications using OpenMP

```
#pragma omp parallel
{
 #pragma omp for private(i, j, k, tmp)
 for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
   tmp = 0.0;
#pragma omp simd reduction(+:tmp) aligned(C,A,B:64)
   for (k = 0; k < N; k++) {
    tmp += A[N * i + k] * B[N * k + j];
    }
 C[N * i + j] = tmp;
 }
}
}
```

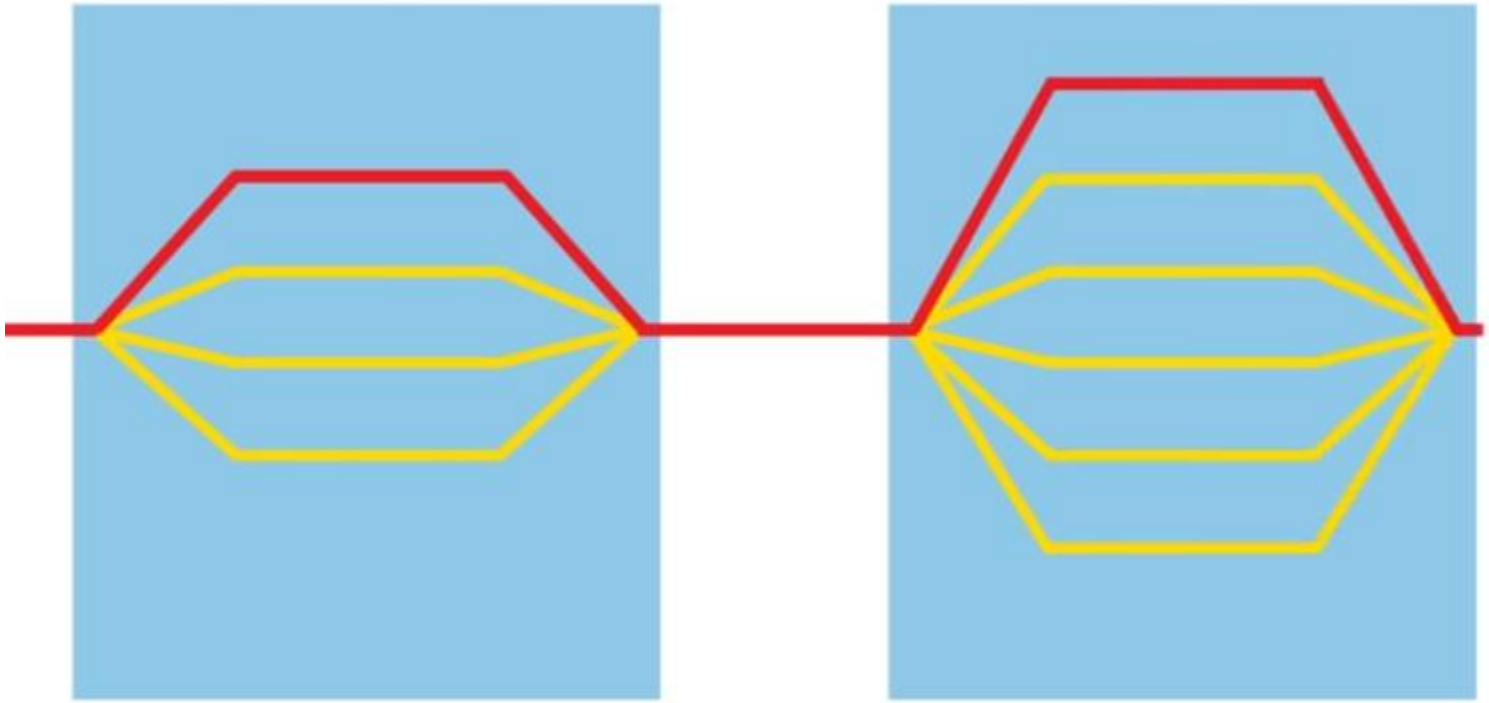
What is OpenMP? (4)



Taken from https://www.youtube.com/watch?v=6jFkNjhJ-Z4&list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG&index=3

What is OpenMP? (5)

- Fork-Join Parallelism



OpenMP Core Syntax

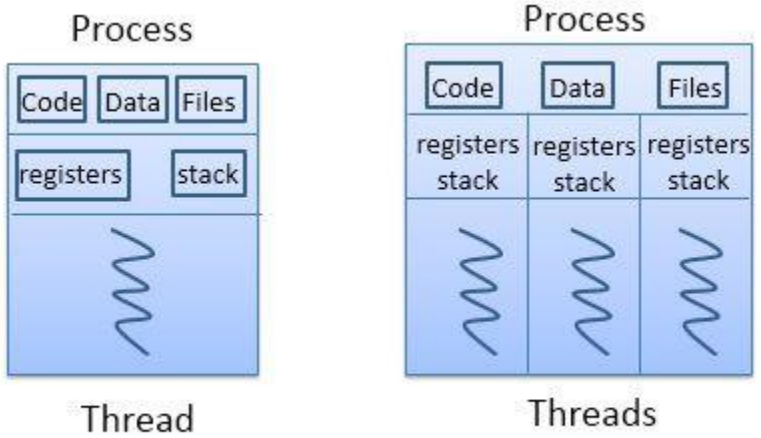
- The syntax in C follows:
  - ***#pragma omp construct [clause]***
  - e.g., ***#pragma omp parallel for***
- A '***#pragma***' is a compiler directive and is going to tell the compiler to do something special beyond the scope of C language.
- The pragmas are designed so that even if the compiler does not support them, the program will still yield correct behavior, but without any parallelism.
- By default, the Visual Studio does **not** support the '*#pragma omp*' clause.
  - No error or warning will be shown.
  - The compiler will just ignore the '*#pragma*' and the program will run serially.
  - ***We must enable OpenMP in Visual Studio***

First Example


```
int main(){

double A[1000];
omp_set_num_threads(4); //requests 4 threads.
#pragma omp parallel {    //fork a number of threads – we asked 4
  int ID = omp_get_thread_num(); //get the ID for each thread
  function1 (ID, A ) //each thread will run this function
} //end of multi-threading region

printf("all done\n"); //just the main thread runs this command

return 0;
}
```

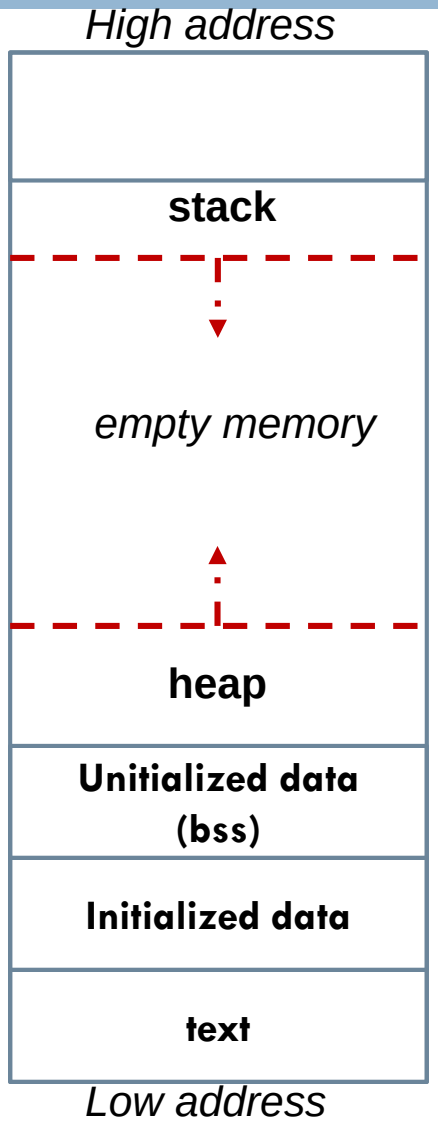
What is difference between thread, process and program?

- **Program:**
  - Program is an executable file containing the set of instructions written to perform a specific job on your computer
  - For example, *skype.exe* is an executable file containing the set of instructions which help us to run *skype*
- **Process:**
  - Process is an executing instance of a program
  - For example, when you double click on the *skype.exe* on your computer, a process is started that will run the skype program
- **Thread:**
  - Thread is the smallest executable unit of a process
  - For example, when you run skype program, OS creates a process and starts the execution of the main thread of that process
  - A process can have multiple threads
  - All threads of the same process share memory of that process



Memory Layout of a Process

- **Text** : contains the compiled code (binary) – read only
- **Initialized data** : contains the global and static variables that are initialized by the programmer
- **Uninitialized data** : contains the global and static variables that are initialized to zero or they do not have explicit initialization
- **Stack**: it is a last in first out (LIFO) structure that stores temporary variables created by each function (including main).
  - Every time a function is called, ***its local data, function arguments and return values*** are pushed into the stack
  - Every time a function exits, all its local data are freed (popped from the stack).
  - Limited size – normally its default value is 1Mbyte.
- **Heap** : contains all the data allocated dynamically



Shared and Private Data

- If we define data outside the '*#pragma omp parallel{ }*' region, they are allocated to heap memory (visual to any thread, shared data).
- If they are defined inside the '*omp parallel{ }*' region, they are allocated to the threads individual stack (private to the thread, local).

```
int main(){

double A[1000];
omp_set_num_threads(4); //requests 4 threads.
#pragma omp parallel {    //fork a number of threads – we asked 4
  int ID = omp_get_thread_num(); //get the ID for each thread
  function1 (ID, A ) //each thread will run this function
} //end of multi-threading region

printf("all done\n"); //just the main thread runs this command

return 0;
}
```

Commands to get environment information

- **nthreads = omp_get_num_threads();** //returns the number of threads used inside #pragma omp parallel { }
- **procs = omp_get_num_procs();** //returns the number of physical CPU cores of this machine
- **maxt = omp_get_max_threads();** //returns the maximum number of threads available. By default this number will be set to the maximum number of available cores
- **inpar = omp_in_parallel();** //This function returns true if currently running in parallel, false otherwise.

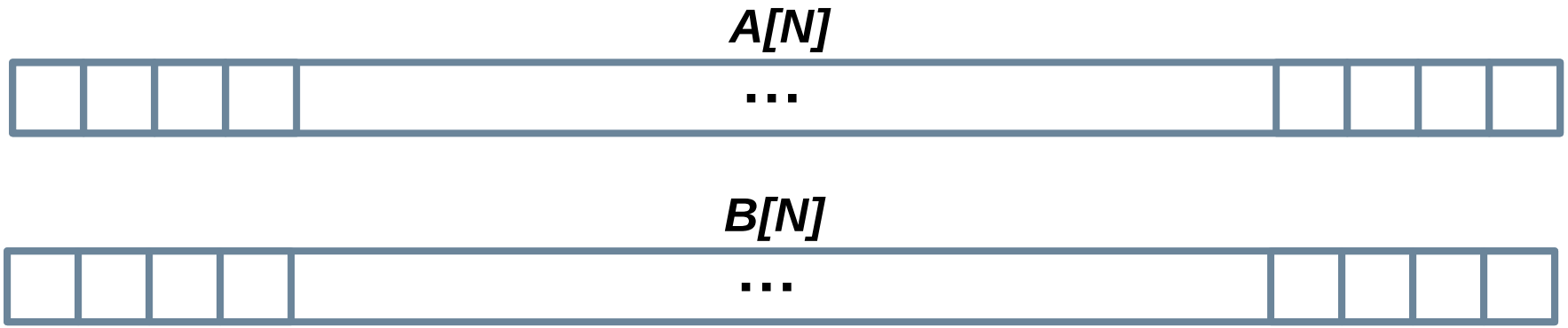
What does this program print?


```
/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel
  {
  int nthreads, tid; //each thread has its own copies
  tid = omp_get_thread_num(); //get the number of each thread
  printf("Hello World from thread = %d\n", tid); //THE ORDER OF THE PRINTF() DIFFERS FROM
RUN TO RUN

  if (tid == 0)  // Only master thread does this – Equivalent to 'If (master thread)'
    {
    nthreads = omp_get_num_threads(); //returns the number of threads used inside #pragma omp
parallel { }
    procs = omp_get_num_procs(); //returns the number of physical CPU cores
    maxt = omp_get_max_threads(); //returns the maximum number of threads available.
    printf("Number of threads = %d\n", nthreads);
    printf("Number of processors = %d\n", procs);
    printf("Max threads = %d\n", maxt);
   }
}  // All threads join master thread and disband
```

Array Addition Example (1)

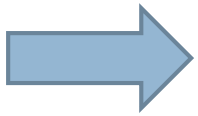
- We will study three different implementations of the following program
  - There are no dependencies – all the loop iterations can be executed in parallel

```
for (i=0; i<N; i++)
  A[i]=A[i] + B[i];
```



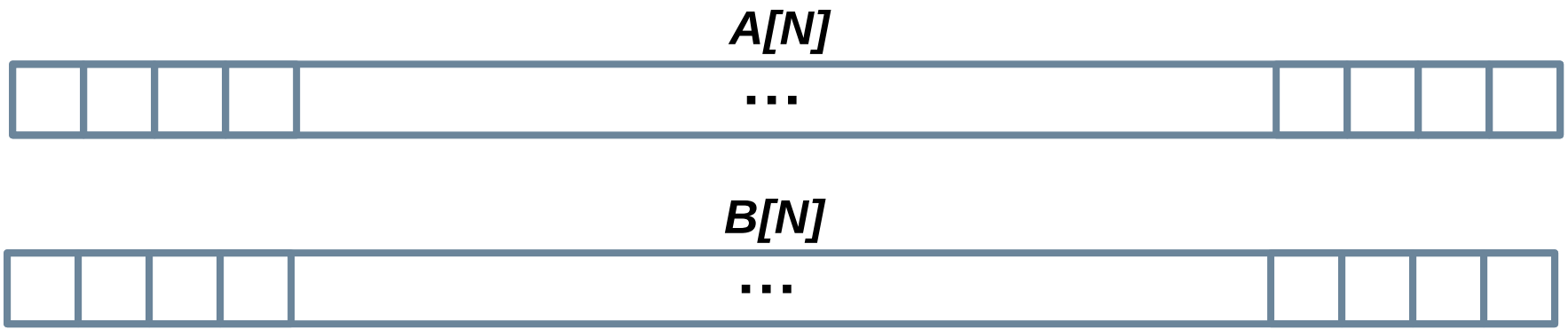
Array Addition Example Implementation #1 (1)


```
for (i=0; i<N; i++)
  A[i]=A[i] + B[i];
```



```
#pragma omp parallel for
for (i=0; i<N; i++)
  A[i]=A[i] + B[i];
```

- **OpenMP makes 'i' private to each thread, automatically**



Array Addition Example Implementation #1 (2)

- **But what OpenMP actually does?**
  - Considering there are four threads, the i loop will be split into four parts and each core will execute its part

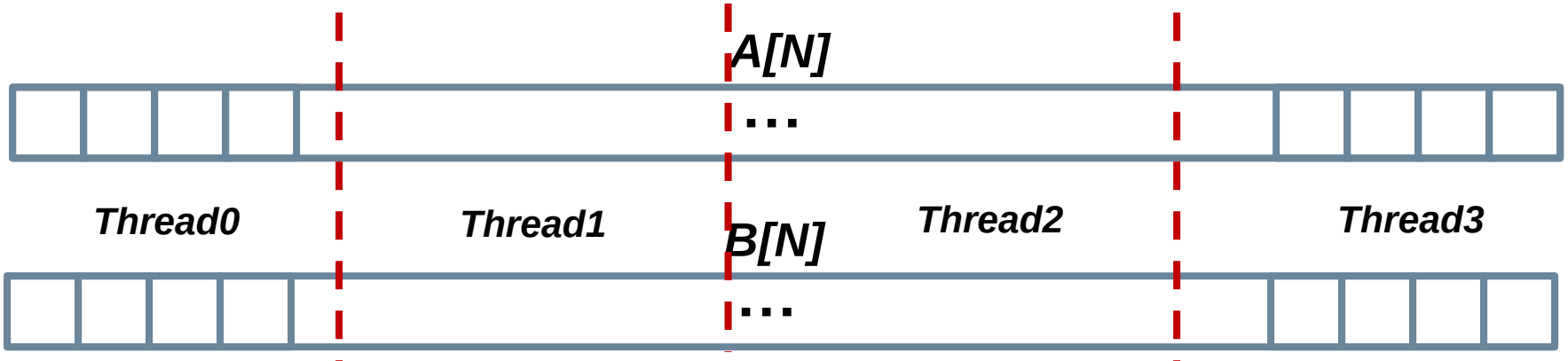
```
#pragma omp parallel for
for (i=0; i<N; i++)
  A[i]=A[i] + B[i];
```

```
for (i=0; i<250; i++)
  A[i]=A[i] + B[i];
```

```
for (i=250; i<500; i++)
  A[i]=A[i] + B[i];
```

```
for (i=500; i<750; i++)
  A[i]=A[i] + B[i];
```

```
for (i=750; i<1000; i++)
  A[i]=A[i] + B[i];
```



Array Addition Example Implementation #2

- This syntax is equivalent

```
#pragma omp parallel
{ //fork a team of threads
#pragma omp for
 for (i=0; i<N; i++)
  A[i]=A[i]+B[i];

} //all threads join master
```

Array Addition Example Implementation #3

- This is not how we write OpenMP programs but it is important to understand how it works
- Each thread executes the code in the parallel region
- Each thread has its own 'start' and 'end' values

```
#pragma omp parallel
{
int id,i,Nthrds, start, end;

id=omp_get_thread_num();
Nthrds=omp_get_num_threads();

start=id   * N / Nthrds;
end=(id+1) * N / Nthrds;

if (id==Nthrds-1)
 end=N;

for (i=start; i<end; i++)
 A[i]=A[i]+B[i];

}
```

2nd Example – Reduction

- In this example, data dependencies exist (race condition)
- If we parallelize this program as before then
  - Different threads will be writing concurrently to 'ave'.
  - Some threads might clashing and trying to update the memory at the same time.
  - In this case, you never know what the value of 'ave' will be.

```
for (i=0; i<N; i++){
 ave+=A[i];
}
```

2nd Example – Reduction Implementation #1

- Each thread has its own private/local 'ave'
- When they are finished, all the 'ave' values must be added together
- ***omp critical***: Restricts execution of the associated structured block to a single thread at a time.
  - threads wait at the beginning of the critical section until no other thread in the team is executing it.

```
int average=0;

#pragma omp parallel
{

int ave=0; //each thread has its own 'ave'

#pragma omp for
for (i=0; i<N; i++){
 ave+=A[i];
}

#pragma omp critical //only one thread at a time can enter
average+=ave;

}
```

2nd Example – Reduction Implementation #2

- Each thread has its own private/local 'ave'
- When they are finished, all the 'ave' values must be added together
- ***omp atomic***: Ensures that a specific storage location is accessed atomically

```
int average=0;

#pragma omp parallel
{

int ave=0; //each thread has its own 'ave'

#pragma omp for
for (i=0; i<N; i++){
 ave+=A[i];
}

#pragma omp atomic //only one thread at a
time can access average
average+=ave;

}
```

2nd Example – Reduction
Implementation #3 – **Recommended Solution**

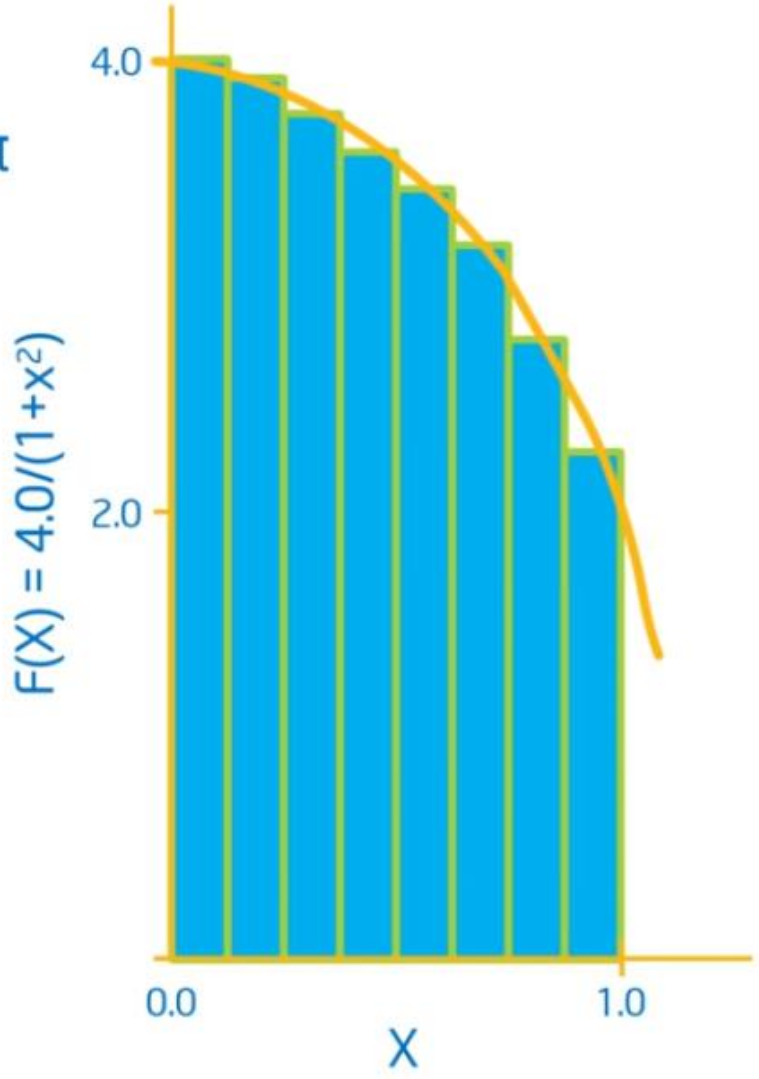
- **reduction (op : list) –** A local copy of each 'list' variable is made and initialized depending on the 'op', e.g., 0 for '+'.
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- Each thread has its own copy of 'ave', each thread does its own summation and when they are done, they are combined with the global copy of 'ave'

```
#pragma omp parallel for reduction(+:ave)
for (i=0; i<N; i++){
 ave+=Acopy1[i];
}
```

An easy but complete example
Calculate the value of Pi (1)

$$\int_0^1 \frac{4.0}{(1+X^2)} \, dx = \pi$$

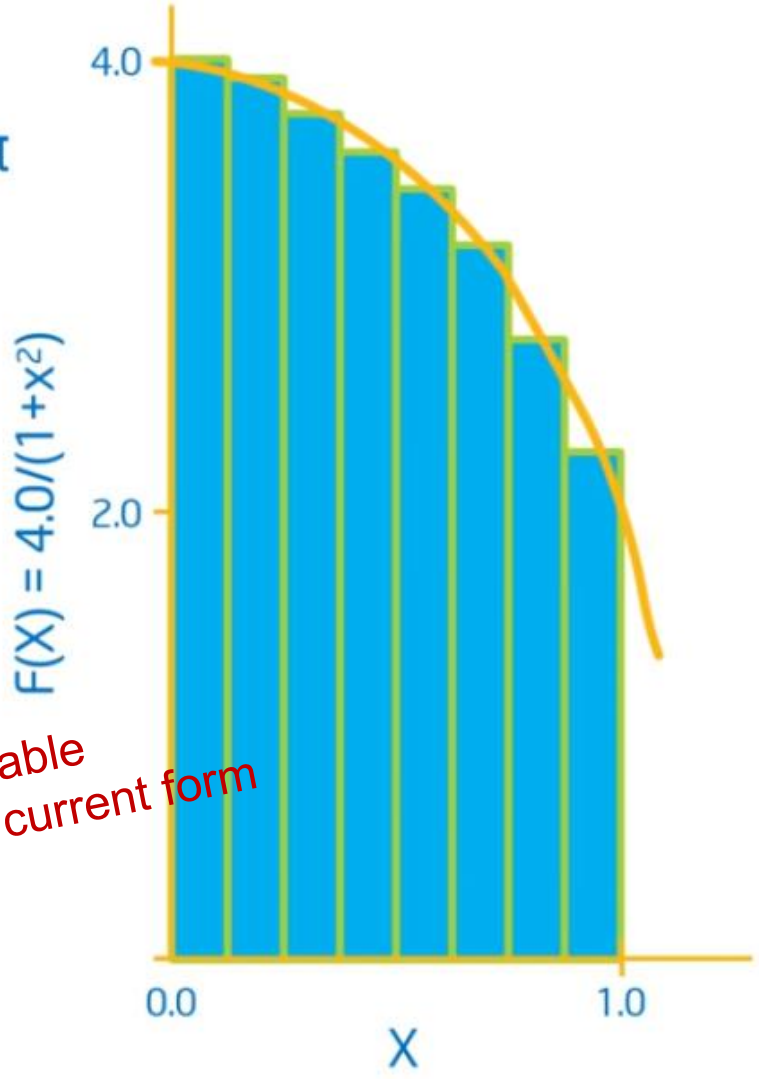
```
double un_opt(){
 int i;
 double x, pi, sum=0.0;
 double step;

  step=1.0/(double) num_steps;

 for (i=0; i<num_steps; i++){
  x=(i+0.5)*step;
  sum = sum + 4.0 / (1.0 + x*x);
}
pi = step * sum;

return pi;
}
```



An easy but complete example
Calculate the value of Pi (2)

$$\int_0^1 \frac{4.0}{(1+X^2)} \, dx = \pi$$

```
double un_opt(){
 int i;
 double x, pi, sum=0.0;
 double step;

  step=1.0/(double) num_steps;

 for (i=0; i<num_steps; i++){
  x=(i+0.5)*step;
  sum = sum + 4.0 / (1.0 + x*x);
}
pi = step * sum;

return pi;
}
```



All threads will be writing to 'sum' variable
We cannot parallelize this code in its current form
What should we do?

Pi Parallel Implementation #1 (not efficient)

- **Solution 1**: Promote sum to an array. All threads will be writing to a different array element.

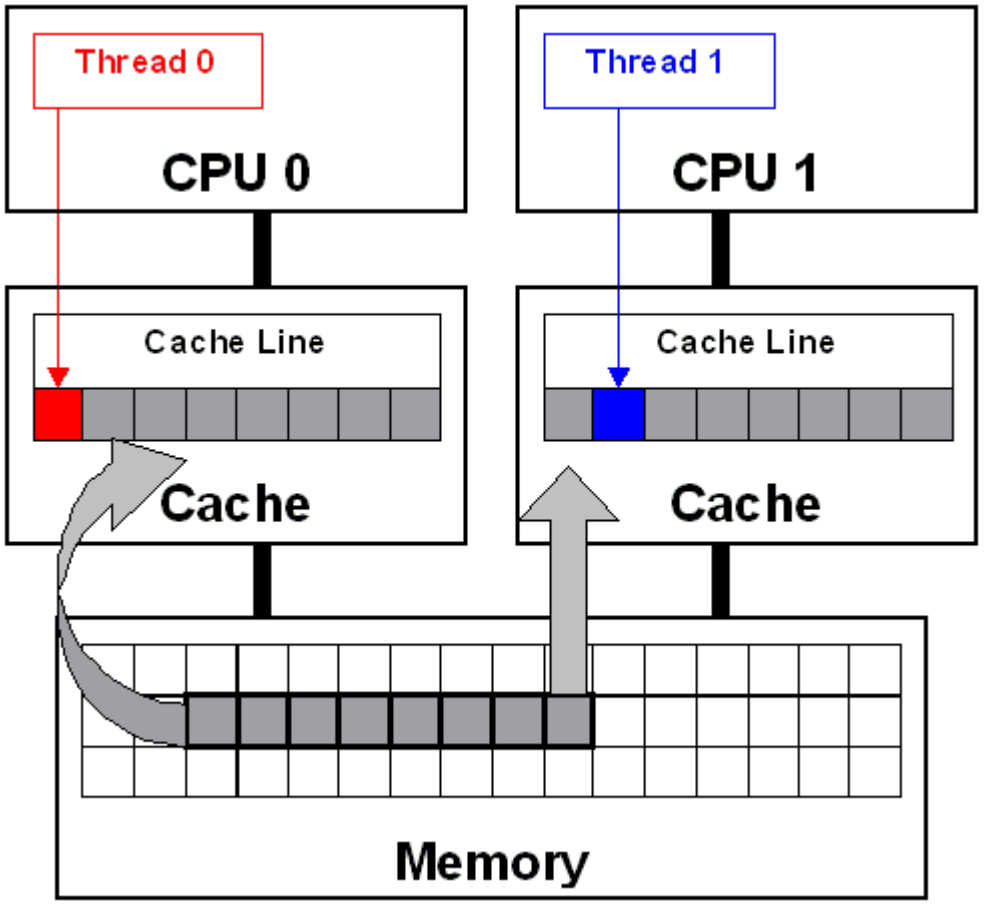
```
int i, nthreads;
double step, x, pi=0.0, sum[NUM_THREADS];

step=1.0/(double) num_steps;
omp_set_num_threads(NUM_THREADS);

#pragma omp parallel
{
   int i,id,nthrds; //local data
   double x;        //local data
   id=omp_get_thread_num();
   nthrds = omp_get_num_threads();

   if (id==0) nthreads=nthrds; //save a copy of num of
threads as the enviroment might choose to give us less
threads than requested. What if I ask 1000 threads?

   for (i=id, sum[id]=0.0; i<num_steps; i=i+nthrds){
      x=(i+0.5)*step;
      sum[id] = sum[id] + 4.0 / (1.0 + x*x);
   }
}

//after this point I have lost the local variables. So, for
the sum to be visible, I must promote sum to an array.
for (i=0, pi=0.0; i<nthreads; i++)
  pi+=sum[i] * step;
```

Pi Parallel Implementation #1 (not efficient)

- the OpenMP version that uses just one thread is slower than the serial version. (Why?)
  - This is because OpenMP adds an overhead.
- By using more threads, the execution time is reduced, but the ***scalability*** is low. (Why?)
  - This is because of the *cache false sharing* problem in 'sum[]' array.
  - Thread0 uses sum[0], Thread1 uses sum[1] etc, but sum[0:7] share the same cache line.
  - Although threads do not use share data, they use the same cache line, which is a shared hardware resource too
  - **False sharing** is a well-known performance issue

False Sharing

- x86/x64 processors have private L1 and L2 caches and shared L3 cache and DDR memory.
- False sharing occurs when threads on different processors modify variables that reside on the same cache line.
- This invalidates the cache line and forces an update, which hurts performance.
- memory system must guarantee *cache coherence*



Taken from https://software.intel.com/content/www/us/en/develop/articles/avoiding-and-identifying-false-sharing-among-threads.html

Pi Parallel Implementation #2 (not recommended)

- This implementation addresses the false sharing problem by using ***different cache lines*** to store the threads' sum variables.
- The sum[] array is promoted to a 2-d array ***sum[NUM_THREADS][PAD], where PAD=8***
- the cache line size in Intel processors is 64bytes and thus can store 8 double values.
- Thus sum[0][0] will always be stored into another cache line than sum[1][0], sum[2][0] etc
- ***Inefficiencies of implementation #2***
  - **It is not portable – different PAD values amongst different processors**
  - **Wastes memory**

Pi Parallel Implementation #3

This version is portable (not recommended)


```
#pragma omp parallel
{
   int i,id,nthrds; //local data
   double x, sum=0.0;         //sum is now local, each thread has its own copy
   id=omp_get_thread_num();
   nthrds = omp_get_num_threads();
   if (id==0) nthreads=nthrds; //save a copy of num of threads as the
enviroment might choose to give us less threads than requested. What if I ask
1000 threads?

   for (i=id; i<num_steps; i=i+nthrds){
      x=(i+0.5)*step;
      sum = sum + 4.0 / (1.0 + x*x);
   }

   #pragma omp critical //mutual exclusion. only one thread at a time can
enter this block. I could use 'atomic' instead of 'critical' it is the same
    {
    pi+=sum * step;
   }
}
```

Pi Parallel Implementation #3
This version is portable

- In this version there is no shared array and thus there is no false sharing.
- This version does not consider the cache line size and thus it is portable.
- This version achieves the same performance as version #2.
- In the '#pragma omp critical' clause, all the threads use their private sum variable to update the shared pi variable.
  - This is not performed in parallel; only one thread at a time can enter the critical block.
  - We could also use 'atomic' instead of 'critical'.

Pi Parallel Implementation #4
Recommended Version


```
#pragma omp parallel
{
 double x;
#pragma omp for reduction(+:sum) //Each thread has its own
copy of sum. Each thread does its own summation and when
they are done, they are combined with the global copy of sum.
 for (i=0; i<num_steps; i++){
  x=(i+0.5)*step;
  sum = sum + 4.0 / (1.0 + x*x);
  }
}
```

Pi Parallel Implementation #4
Recommended Version

- Or like that using just a single #pragma line (more about this next week)

```
#pragma omp parallel for private(x) reduction(+:sum) //x is not defined inside the parallel region. Thus by default it is a shared variable. private(x) creates a private x variable in each thread. Be Careful: x is unitialized no matter what its previous value is.
 for (i=0; i<num_steps; i++){
  x=(i+0.5)*step;
  sum = sum + 4.0 / (1.0 + x*x);
}
```

Serial VS Parallel version
See how elegant OpenMP is


```
double un_opt(){
 int i;
 double x, pi, sum=0.0;
 double step;

  step=1.0/(double) num_steps;

 for (i=0; i<num_steps; i++){
  x=(i+0.5)*step;
  sum = sum + 4.0 / (1.0 + x*x);
}
pi = step * sum;

return pi;
}
```

```
double version6(){
int i;
double x, pi, sum=0.0;
double step;

  step=1.0/(double) num_steps;

#pragma omp parallel for private(x) reduction(+:sum)
for (i=0; i<num_steps; i++){
  x=(i+0.5)*step;
  sum = sum + 4.0 / (1.0 + x*x);
}
pi = step * sum;

return pi;

}
```

Further Reading

- Guide into OpenMP: Easy multithreading programming for C++, available at https://bisqwit.iki.fi/story/howto/openmp/#ParallelConstruct
- OpenMP Application Programming Interface Examples, available at https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwiOip2R-rrqAhX8XRUIHa5HC0QQFjAAegQIAxAB&url=https%3A%2F%2Fwww.openmp.org%2Fwp-content%2Fuploads%2Fopenmp-examples-4.5.0.pdf&usg=AOvVaw3BDlLKC3VhdJI1iTj1RE_p

Part 2 - Outline

- Dot Product example
- The schedule Clause
- Vectorization using OpenMP
- Matrix-Vector Multiplication example

Dot Product Example (1)

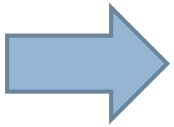
- Drawing upon what we have learned so far, can you parallelize this program using OpenMP?

```
double dot_prod_serial(double *a, double *b){

double sum = 0.0;
int i;

for (i=0; i<N; i++)
  {
    sum += (a[i] * b[i]);
  }


return sum;
}
```

Dot Product Example (2)

- Drawing upon what we have learned so far, can you parallelize this program using OpenMP?

```
double dot_prod_serial( ... ){

double sum = 0.0;
int i;

for (i=0; i<N; i++)
  {
    sum += (a[i] * b[i]);
  }

return sum;
}
```



```
double dot_prod_parallel_ver1( ... ){

double sum = 0.0;
int i;

#pragma omp parallel for reduction(+:sum)
for (i=0; i<N; i++)
  {
    sum += (a[i] * b[i]);
  }

return sum;

}
```

The Schedule Clause (1)

- ***The schedule clause:*** It affects how loop iterations are mapped onto threads.
  - ***Schedule (static, [,chunk])*** . [ ] is optional.
    - Assigns blocks of iterations of size chunk to each thread.
    - Used when the amount of iterations is pre-determined and predictable in advance.
    - Scheduling is done at compile time.
    - OpenMP divides iterations into chunks that are approximately equal in size and it distributes at most one chunk to each thread.

The Schedule Clause (2)

- ***The schedule clause:*** It affects how loop iterations are mapped onto threads.
  - ***Schedule (dynamic, [,chunk]).***
    - *Omp scheduler d*ecides at runtime which iterations will be allocated to each thread
    - ***Each thread asks (at runtime) for a chunk of iterations, executes them, then asks for another chunk and so on***
    - Used when the amount of iterations is unpredictable, highly variable work per iteration.
    - Scheduling is done at runtime.
    - No particular order in which the chunks are distributed to the threads.
    - The order changes each time when we execute the for loop.
    - The dynamic scheduling type is appropriate when the iterations require different computational costs.
    - The dynamic scheduling type has *higher overhead than the static scheduling* type because it dynamically distributes the iterations during the runtime.

The Schedule Clause (3)

- ***The schedule clause:*** It affects how loop iterations are mapped onto threads.
  - ***Schedule (guided, [,chunk]).***
    - **Not really used often.**
    - The guided scheduling type is similar to the dynamic scheduling type.
    - The difference with the dynamic scheduling type is in the size of chunks.
      - The size of a chunk is proportional to the number of unassigned iterations divided by the number of the threads. Therefore, the size of the chunks decreases.
      - The initial chunks are larger, because they reduce overhead.
      - The smaller chunks fills the schedule towards the end of the computation and improve load balancing.
    - This scheduling type is especially appropriate when poor load balancing occurs toward the end of the computation.

The Schedule Clause (4)

- ***The schedule clause:*** It affects how loop iterations are mapped onto threads.
  - ***Schedule (runtime)***
    - Schedule and chunk size taken from the *omp_schedule* enviroment variable (or the runtime library).
    - Used when we are not sure about which one is best (static or dynamic)
  - ***Schedule (auto)***
    - Schedule is left up to the runtime to choose
    - This option is new in OpenMP.
    - It lets the compiler to decide and do its best.

Schedule (static) clause

What does this program print?

Consider N=8


```
#pragma omp parallel
{
  int tid = omp_get_thread_num();
  if (tid == 0) //if master thread
    {
    int nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
   printf("Thread %d is starting...\n",tid);

#pragma omp for reduction(+:sum) schedule(static)
for (i=0; i<N; i++)  {
    sum += (a[i] * b[i]);
    printf("Thread %d: executes iteration i= %d\n", tid, i);
  }
}
```

```
Number of threads = 4
Thread 0 is starting...
Thread 0: executes iteration i= 0
Thread 0: executes iteration i= 1
Thread 1 is starting...
Thread 1: executes iteration i= 2
Thread 2 is starting...
Thread 2: executes iteration i= 4
Thread 2: executes iteration i= 5
Thread 1: executes iteration i= 3
Thread 3 is starting...
Thread 3: executes iteration i= 6
Thread 3: executes iteration i= 7
```

Schedule (dynamic) clause

What does this program print?

Consider N=8


```
#pragma omp parallel
{
  int tid = omp_get_thread_num();
  if (tid == 0) //if master thread
    {
    int nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
   printf("Thread %d is starting...\n",tid);

#pragma omp for reduction(+:sum) schedule(dynamic)
for (i=0; i<N; i++)   {
    funct(i)
    printf("Thread %d: executes iteration i= %d\n", tid, i);
  }
}
```

```
Number of threads = 4
Thread 0 is starting...
Thread 0: executes iteration i= 0
Thread 0: executes iteration i= 1
Thread 0: executes iteration i= 2
Thread 0: executes iteration i= 3
Thread 0: executes iteration i= 4
Thread 0: executes iteration i= 5
Thread 3 is starting...
Thread 3: executes iteration i= 7
Thread 2 is starting...
Thread 1 is starting...
Thread 0: executes iteration i= 6
```

Schedule (static, 4) clause

What does this program print?

Consider N=8


```
#pragma omp parallel
{
  int tid = omp_get_thread_num();
  if (tid == 0) //if master thread
    {
    int nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
   printf("Thread %d is starting...\n",tid);

#pragma omp for reduction(+:sum) schedule(static, 4)
for (i=0; i<N; i++)   {
    sum += (a[i] * b[i]);
    printf("Thread %d: executes iteration i= %d\n", tid, i);
  }
}
```

```
Number of threads = 4
Thread 0 is starting...
Thread 0: executes iteration i= 0
Thread 0: executes iteration i= 1
Thread 0: executes iteration i= 2
Thread 0: executes iteration i= 3
Thread 2 is starting...
Thread 3 is starting...
Thread 1 is starting...
Thread 1: executes iteration i= 4
Thread 1: executes iteration i= 5
Thread 1: executes iteration i= 6
Thread 1: executes iteration i= 7
```

Schedule (dynamic, 4) clause

What does this program print?

Consider N=8


```
#pragma omp parallel
{
  int tid = omp_get_thread_num();
  if (tid == 0) //if master thread
    {
    int nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
   printf("Thread %d is starting...\n",tid);

#pragma omp for reduction(+:sum) schedule(dynamic, 4)
for (i=0; i<N; i++)   {
    sum += (a[i] * b[i]);
    printf("Thread %d: executes iteration i= %d\n", tid, i);
  }
}
```

```
Number of threads = 4
Thread 0 is starting...
Thread 0: executes iteration i= 0
Thread 0: executes iteration i= 1
Thread 0: executes iteration i= 2
Thread 0: executes iteration i= 3
Thread 0: executes iteration i= 4
Thread 0: executes iteration i= 5
Thread 0: executes iteration i= 6
Thread 0: executes iteration i= 7
Thread 2 is starting...
Thread 1 is starting...
Thread 3 is starting...
```

Thread 0 asked for 4 iterations, it executed them and the scheduler decided to give the other 4 iterations to thread 0.

The Schedule Clause (5)

- In this module we will be writing programs where the number of iterations per thread can be calculated at compile time (static programs)
- Thus, we will be using the ***schedule(static, chunk)*** clause

Vectorization using OpenMP
The '*#pragma omp simd*' Construct

- OpenMP 4.0 introduced omp simd, accessed via #pragma omp simd as a standard set of hints that can be given to a compiler to encourage it to auto-vectorise code.
- Compilers may not vectorize loops when they are complex or possibly have dependencies, even though the programmer is certain the loop will execute correctly as a vectorized loop.
- The simd construct **assures the compiler** that the loop can be vectorized.
- **Be careful.** Using ***omp simd*** bypasses the compiler analysis.
- So, **use with caution** as
  - Incorrect results are possible
  - Poor performance is possible
  - memory errors are possible

Use x86-64 Intrinsics or '#pragma omp simd' ?

- The OpenMP 'simd' clause
  - is easy to use
  - can provide good performance for simple programs
- The x86-64 Intrinsics
  - are harder to use
  - provide better performance
  - can allow for other optimizations too further improving performance, e.g., register blocking

‘#pragma omp simd’ in Visual Studio

- The ‘simd’ clause is a new feature and supported only in Visual Studio 2019 via the command line
- Supported in Linux too

Vectorization
Do not forget to align your arrays

- We can either allocate aligned memory ***statically*** using
  - *float A[N] __attribute__((aligned (64)));* *//In Linux only*
  - *__declspec(align(64)) float A[N]* *//In Visual studio only*

- or ***dynamically*** using
  - *_mm_malloc (N * sizeof(float),64);* *//Linux and Visual Studio*

Vectorization using OpenMP

- ***#pragma omp simd*** : The simd construct can be applied to a loop to indicate that the loop can be vectorized
  - ***aligned(y,x,a:64)*** : The aligned clause asserts to the compiler that an array is aligned.
    - Using this clause allows the compiler to safely use SIMD instructions that have strict alignment requirements.
    - If this clause is used, the programmer is responsible for ensuring that the data is in fact aligned.
  - ***reduction(+:tmp)*** : The reduction clause instructs the compiler to perform a vector reduction on a variable.
    - Same as in multithreaded
- **How can we be sure that the compiler vectorized the code?** To be verify that, compile using **-fopt-info-vec-optimized** option (gcc only)

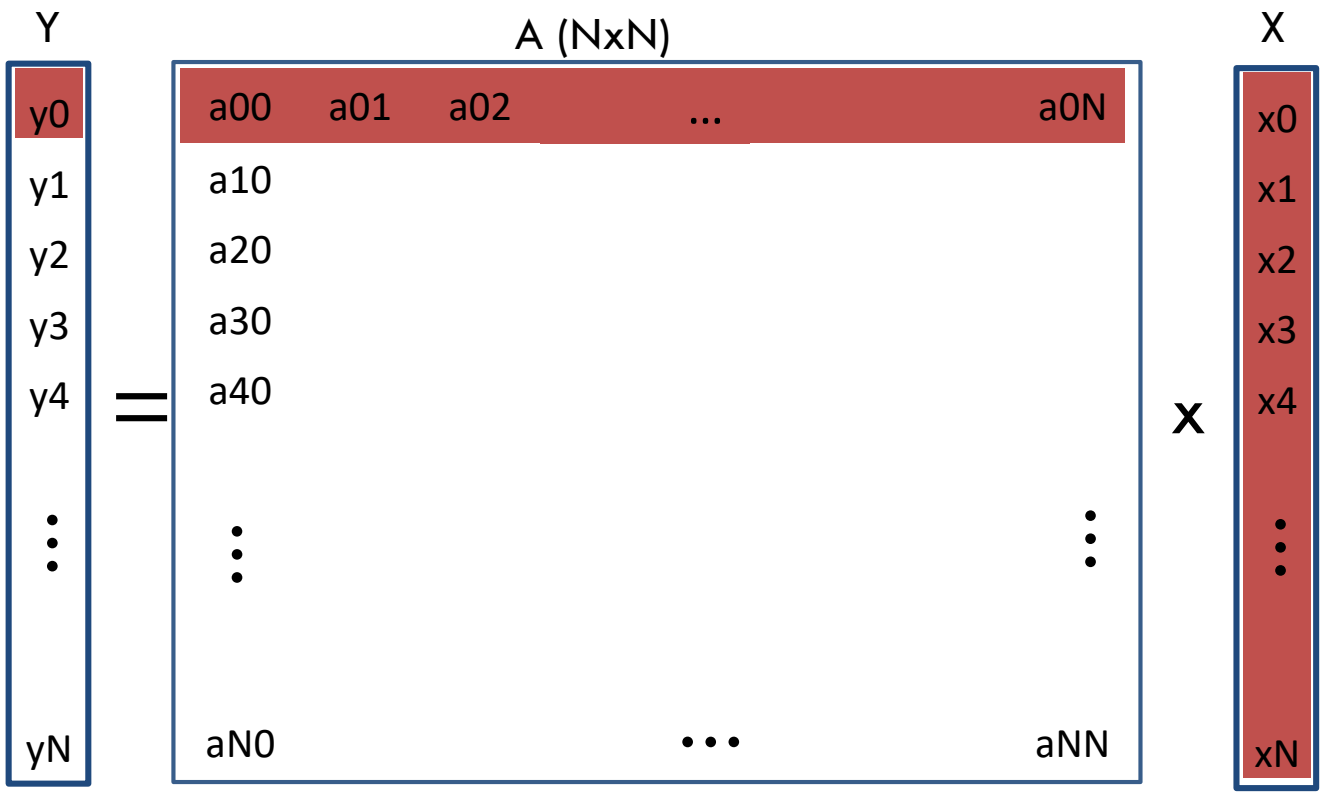
An Introduction to the storage attributes

- We can define in the #pragma clause whether a variable is shared amongst all threads or is private.
  - ***shared*** (a): all threads can access 'a'.
  - ***private*** (a): each thread creates an **un-initialized copy** of 'a'
- The 'i' loop is the loop next to the omp clause and it is private by default.
- The 'j' loop is not private by default and thus it must be defined as private
- 'Y' is shared by default and thus it is not needed, but it helps readability and might prevent bags.

```
#pragma omp parallel for shared(Y) private(i, j)
 for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    Y[i]=…
```

Matrix-Vector Multiplication Serial Version


```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    y[i]+=a[N*i+j] * x[j];
```



Matrix-Vector Multiplication Multi-threaded Version

- Both code versions are valid

```
#pragma omp parallel for private(I,j)
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
 y[i]+=a[N*i+j]*x[j];
```

```
#pragma omp parallel for shared(y,a,x) private(i,j) schedule(static)
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
 y[i]+=a[N*i+j]*x[j];
```

Matrix-Vector Multiplication Multi-threaded Version (2)

- Performance results on a PC with four physical cores
  - N=128
    - 2 threads -> speedup x1.85
    - 3 threads -> speedup x2.55
    - 4 threads -> speedup x3.13
  - N=4096
    - 2 threads -> speedup x1.95
    - 3 threads -> speedup x2.93
    - 4 threads -> speedup x3.75

Why the scalability is low for N=128?

```
#pragma omp parallel for shared(y,a,x) private(i,j) schedule(static)
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
 y[i]+=a[N*i+j]*x[j];
```

Matrix-Vector Multiplication Multi-threaded Version (3)

- Why the scalability is low for N=128?
  - Because **the overhead for creating and synchronizing the threads is comparable to the threads' execution time**.
  - The code will scale well only when each thread executes at least a minimum amount of instructions
  - The amount of instructions per thread is found experimentally and depends on the target platform

Matrix-Vector Multiplication
Multi-threaded and Vectorized Version

In my PC, the omp simd clause gives a speedup x2

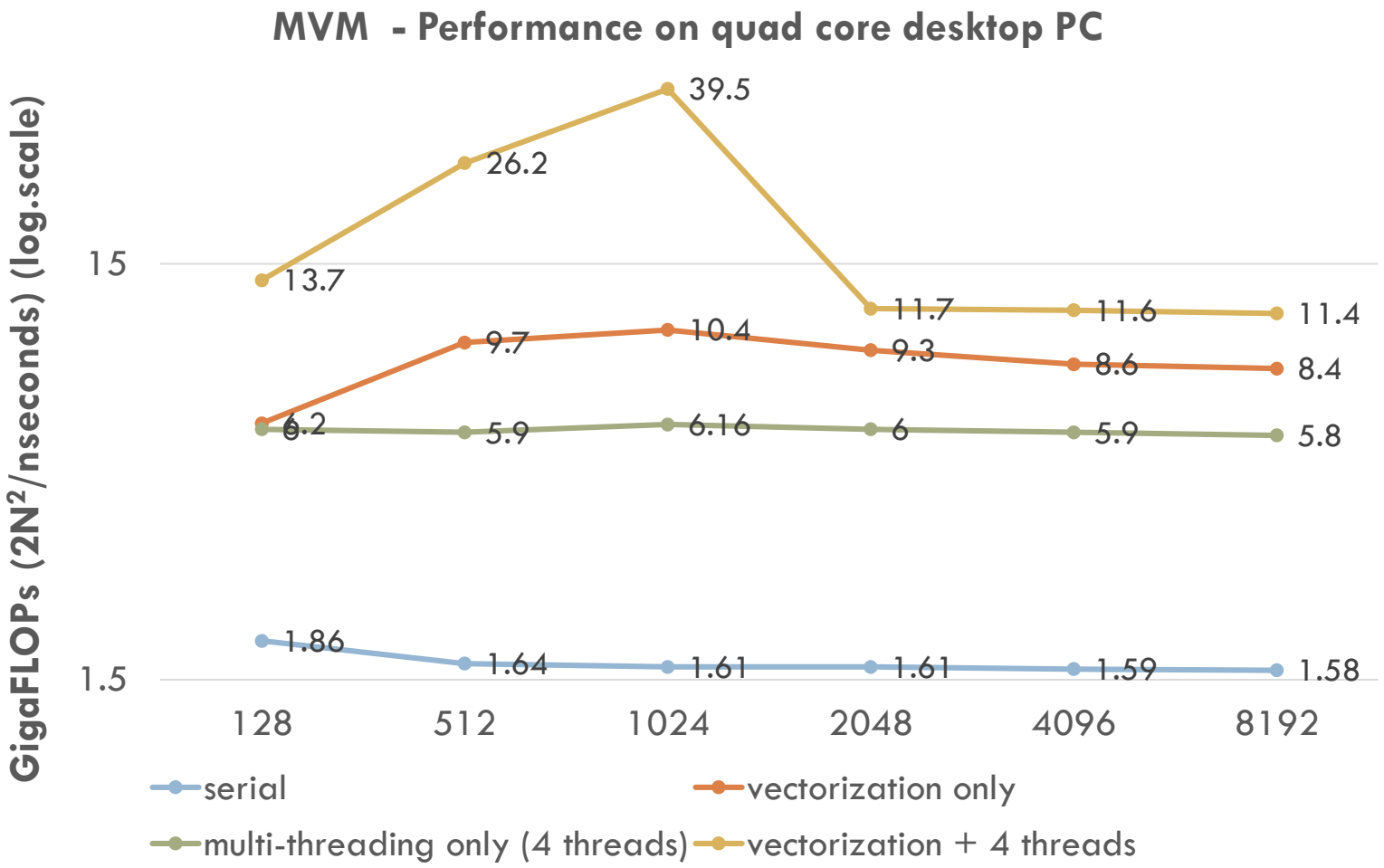
```
#pragma omp parallel for private(j, tmp)
for (i=0; i<N; i++) {
 tmp=y[i];
#pragma omp simd aligned(y,x,a:64) reduction(+:tmp)
 for (j=0; j<N; j++) {
  tmp+=a[N*i+j]*x[j];
  }
y[i]=tmp;
}
```

In my PC, this code performs x7.55 times faster than the serial version …
We can do much better …

Matrix-Vector Multiplication
Vectorization using x86-64 intrinsics vs OpenMP

- Execution time values are similar
  - See MVM_parallel_ver5() routine
- But …
  - By using x86-64 AVX intrinsics, we can enable other optimizations which cannot be applied to the openmp version, further improving performance, e.g., register blocking
  - x86-64 AVX intrinsics can provide improved performance in the general case.

Performance in GigaFLOPS
Can we do better?



Performance in GigaFLOPS
Can we do better? (2)

- Yes we can. Much better
- Optimizations such as
  - register blocking
  - loop tiling
  - software prefetching
- Can boost performance

- Register blocking is the most efficient optimization for MVM algorithm and can boost performance in **90GigaFLOPS**
  - A code example is provided

Let's study the MVM_parallel.c (see GitHub)

Routine	Gflops for N=1024
MVM_serial()	1.8
MVM_parallel_ver1() – omp parallel	6.7
MVM_parallel_ver4() – omp simd + omp parallel	42.7
MVM_parallel_ver5() – omp parallel + AVX	53
MVM_parallel_ver6() – omp parallel + AVX + reg. blocking factor of 8	82.5

Further Reading

- Effective Vectorization with OpenMP 4.5, available at https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjCksHTqr3qAhX4SRUIHSmlBYAQFjAAegQIBhAB&url=https%3A%2F%2Finfo.ornl.gov%2Fsites%2Fpublications%2Ffiles%2FPub69214.pdf&usg=AOvVaw22CMKDJzHHKHKSFzm8P9qr
- Chapter 51 in OpenMP Application Programming Interface, Examples, available at https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjlooTyqr3qAhWYaRUIHZmEC58QFjAAegQIAhAB&url=https%3A%2F%2Fwww.openmp.org%2Fwp-content%2Fuploads%2Fopenmp-examples-4.5.0.pdf&usg=AOvVaw3BDlLKC3VhdJI1iTj1RE_p

Part3 - Outline

- More advanced OpenMP Topics
  - Omp barrier, omp nowait, omp single, omp master, Collapse clause
- Changing the storage attributes
- omp sections
- Parallel Programming Design Patterns
- Divide and conqueror Design Pattern
- Omp tasks
- Environmental Variables

The ‘omp single’ Construct

- ***#pragma omp single :*** The single construct specifies that the given statement/block is executed **by only one thread**.
  - It is **unspecified which thread**.
  - **Other threads skip the statement/block and wait at an implicit barrier at the end of the construct**.
  - Do not assume that the single block is executed by whichever thread gets there first.
  - According to the standard, the decision of which thread executes the block is implementation-defined.

```
#pragma omp parallel {
   funct1();  //all threads execute this
   #pragma omp single
   {
   funct2(); //just one thread executes this
    } //other threads wait here for the single thread to finish
  funct3(); //all threads execute this
}
```

The ‘omp master’ Construct

- ‘#pragma omp master’: The master construct is similar to single, except that the statement/block is run by the *master* thread, and there is no implied barrier
- **Other threads skip the construct without waiting.**
- The following two examples are equivalent.

```
#pragma omp parallel //example1
{
 funct1();  //executed by all threads
#pragma omp master {
  funct2(); //just Thread0
  }
funct3(); //all threads
}
#pragma omp parallel //example2
 {
 funct1();
  if(omp_get_thread_num() == 0)  {
    funct2();
  }
funct3();
 }
```

The ‘omp barrier’ Construct

- ***#pragma omp barrier:*** Each thread waits at the barrier until all threads arrive.

```
#pragma omp parallel
{
 int id=omp_get_thread_num();
 A[id]=funct1(id);

#pragma omp barrier //no thread will execute funct2, before A[] is stored
B[id]= funct2(id, A);
}
```

The ‘nowait’ Directive

- ***#pragma omp nowait***: nowait overrides the barrier implicit at the end of a directive.
  - The nowait directive can only be attached to: sections, for and single.

```
#pragma omp parallel
{
   #pragma omp for nowait
      for (i=1; i<n; i++)
          b[i] = (a[i] + a[i-1]) / 2.0;

   #pragma omp for nowait
      for (i=0; i<m; i++)
         y[i] = sqrt(z[i]);
}
```

The ‘collapse’ Clause

- ***Collapse clause***: Two or more loops are merged to one and parallelized.
  - Used when the number of iterations in the loop is small.
  - Use the collapse-clause to increase the total number of iterations that will be partitioned across the available number of omp threads.

```
#pragma omp parallel for collapse(2)
for ( i=0; i<15; i++) //there are only 15 iterations to parallelize. Using collapse there will be 1200
    for( j=0; j<80; j++)
      func(i,j);
```

Nested Parallelism

- ***Loop nesting:*** Nested parallelism" is disabled in OpenMP by default (it can be used though), and thus a second pragma will be ignored at runtime.

```
#pragma omp parallel for
for ( i=0; i<15; i++)
   #pragma omp for // This is ignored, nesting like this is not allowed by default
   for( j=0; j<80; j++)
     func(i,j);
```

Changing the Storage Attributes

- ***shared (a):*** all threads can access 'a'
- ***private (a):*** each thread creates an un-initialized copy of 'a'
- ***firstprivate (a):*** each thread creates an initialized copy of 'a'
- ***lastprivate (a)*** : the value of 'a', of the last iteration of the loop, is stored back as global. If a loop goes from i=[0,N-1], then the thread that executed the iteration N-1, its value of tmp will be copied out to the global scope.
- ***default (private | shared | none).*** The default clause forces a programmer to explicitly specify the data-sharing attributes of all variables in a parallel region.
  - E.g., #pragma omp parallel for default(shared) private(a, b).
  - You can also write parallel regions with the default(none) clause and then specify the private and shared ones.

Changing the Storage Attributes
An Example


```
void wrong(){
int tmp=0;
#pragma omp parallel for private(tmp) //create a var tmp that is
private (un-initialized)
 for (i=0; i<N; i++)
  tmp+=j;  //Problem, the first value of tmp is not zero

printf (tmp); //problem, will see the global tmp, not the private.
The private tmp is disappeared
}
```

Changing the Storage Attributes An Example (2)


```
Void good(){
tmp=1;
#pragma omp parallel for firstprivate(tmp) //create a var tmp that is private and initialized
for (i=0; i<N; i++) {
  if ((i%2)==0)
     A[i]=tmp;
 }
else
     A[i]=0;
}
```

Changing the Storage Attributes
An Example (3)

- Consider the following code.
  1. Are *a,b,c* local to each thread or shared?
  2. What are the a,b,c values inside the parallel region and after, in the code below?

```
int a=1,b=1, c=1;
#pragma omp parallel private(b) firstprivate(c)
{ … }
```

Omp Sections (1)

- ***omp sections***: The section construct is one way to distribute different tasks to different threads.
  - Each section refers to a different task and is executed by one only thread
  - Unlike to the previous examples, where they were based on **loop parallelism**, sections enable **task parallelism** (see next slide)

```
#pragma omp sections
{
  #pragma omp section  //just one thread executes this section
 { funct1(); }
 #pragma omp section //just one thread executes this section
{ funct2();
funct3(); }
#pragma omp section   //just one thread executes this section
{ funct4(); }
} //the other threads wait here
```

```
#pragma omp parallel
shared(a,b,c,d,nthreads) private(i,tid)
  {
  tid = omp_get_thread_num();
  #pragma omp master
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n",
nthreads);
    }
  printf("Thread %d starting...\n",tid);

  #pragma omp sections nowait
    {
    #pragma omp section
      {
      printf("Thread %d doing section
1\n",tid);
      #pragma omp simd
      for (i=0; i<N; i++)
        {
        c[i] = a[i] + b[i];
       // printf("Thread %d: c[%d]=
%f\n",tid,i,c[i]);
        }
      }
    #pragma omp section
      {
      printf("Thread %d doing section
2\n",tid);
      #pragma omp simd
      for (i=0; i<N; i++)
        {
        d[i] = a[i] * b[i];
        //printf("Thread %d: d[%d]=
%f\n",tid,i,d[i]);
        }
      }

    }  /* end of sections */

    printf("Thread %d done.\n",tid);

  }  /* end of parallel section */
```

This Program crashes. Why?


```
int A[1024][1024];

#pragma omp parallel shared(nthreads) private(i,j,tid,A)
 {

 tid = omp_get_thread_num();
 #pragma omp master
  {
  nthreads = omp_get_num_threads();
  printf("Number of threads = %d\n", nthreads);
  }
 printf("Thread %d starting...\n", tid);

 for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    A[i][j] = (i-j)%100;
 }
```

- Array A is private, which means that every thread will try to allocate an array of size NxN.
- The memory segment that is used, is the stack, not the heap, as A[][] is a private array
- the size of the array is very large and the program cannot allocate such space on the threads' stack.
- This makes the program to crash.

The 'Omp task' Clause (advanced topic)

- ***Omp task***: When a thread encounters a task construct, a task is generated. The moment of execution of the task depends on the runtime system.
  - **A thread that executes a task might be different from the thread that originally encountered it.**
  - The tasks are independent units of work and executed in any order
  - The code associated with a task construct will be executed only once.

```
#pragma omp parallel
{
 #pragma omp single
 {
  #pragma omp task
     { funct1(); }
  #pragma omp task
     { funct2(); }
  #pragma omp task
     { funct3(); }
 #pragma omp taskwait //all the tasks must end here
 }
}
```

Environmental Variables (1)

- The OpenMP specification defines several **environment variables** that control the execution of OpenMP programs.
  - ***OMP_NUM_THREADS*** : Sets the number of threads to use during execution of a parallel region. You can override this value by a **NUM_THREADS** clause, or a call to **OMP_SET_NUM_THREADS()**.
  - ***OMP_STACKSIZE*** : Sets the stack size for each thread.
  - ***OMP_WAIT_POLICY*** : The OMP_WAIT_POLICY environment variable provides a hint to an OpenMP implementation about the desired behavior of waiting threads. It can be either ACTIVE or PASSIVE.
    - **In active**, the thread actively spins waiting for something to be available. This consumes CPU power.
    - **In passive**, the thread is put into sleep. Putting a thread into sleep and waiting it up, costs a lot

Environmental Variables (2)

- The OpenMP specification defines several **environment variables** that control the execution of OpenMP programs.
  - ***OMP_PROC_BIND*** : It can be either true or false. It sets the thread affinity policy to be used for parallel regions at the corresponding nested level.
    - If the environment variable is set to false, the execution environment may move a thread to another CPU core.
    - If it is true, threads are not shuffled among the cores.
    - Use true for cache intensive algorithms.
- **Example**: to set OMP_PROC_BIND=true in Linux, type the following command:

  export OMP_PROC_BIND=TRUE //sets it

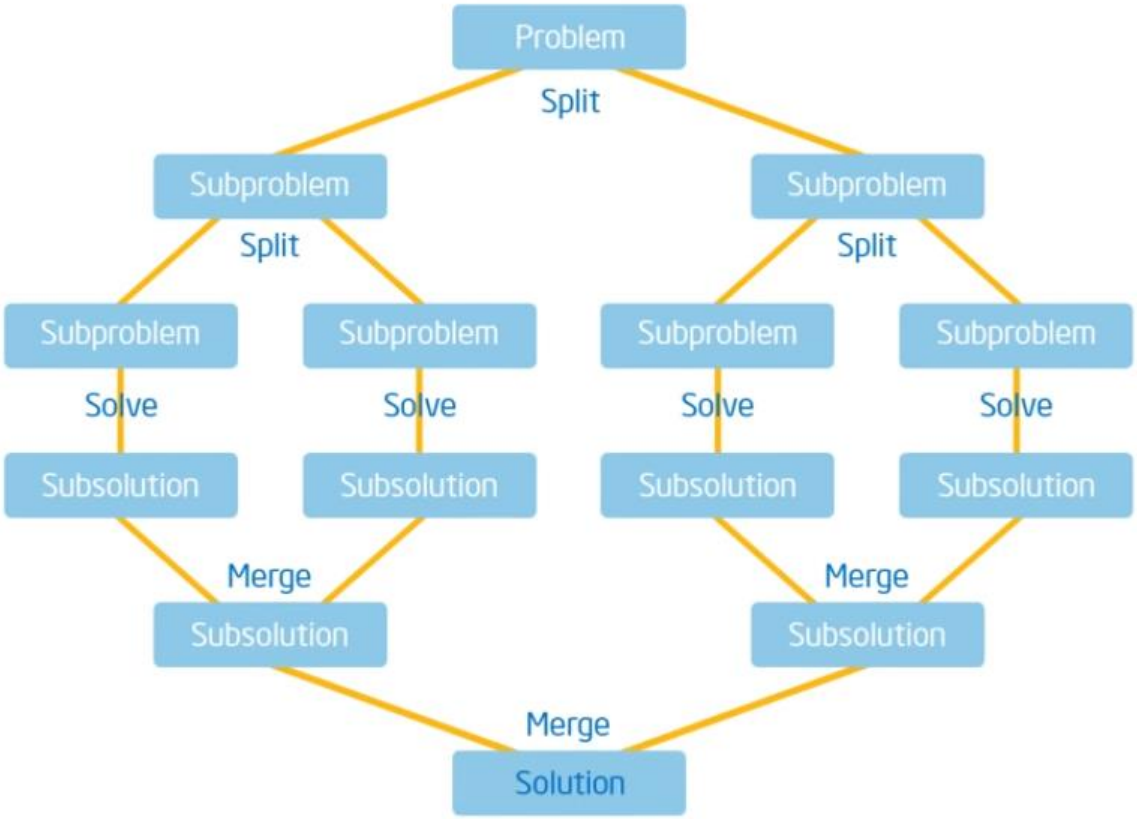
  echo $OMP_PROC_BIND //prints it, to make sure it worked
- To learn more about environment variables visit https://www.openmp.org/spec-html/5.0/openmpch6.html_.

Design Patterns for Parallel Programming

- No matter which programming language you use, there are specific algorithmic concepts that are universal
- These are the ***design patterns for parallel programming*** – So far, we have used just one programming language and three different patterns
- **The design patterns we have seen so far are:**
  - **Single Program Multiple Data (SPMD) pattern**
    - A single program runs on many processing elements
    - Create a collection of units of execution (here threads) and each one will run the same program
    - Remember Pi program - version1(),version2(),version3() routines
  - **Loop parallelism pattern**
    - Most used in OpenMP – we have seen many examples
  - **Task Parallelism**
  - **Divide and Conquer pattern** (next slide)

Divide and Conquer design paradigm

- ***Divide and Conquer algorithm:*** recursively breaking down a problem into two or more sub-problems, until these become simple enough to be solved directly.
  - The solutions to the sub-problems are then combined to give a solution to the original problem



Fibonacci Sequence

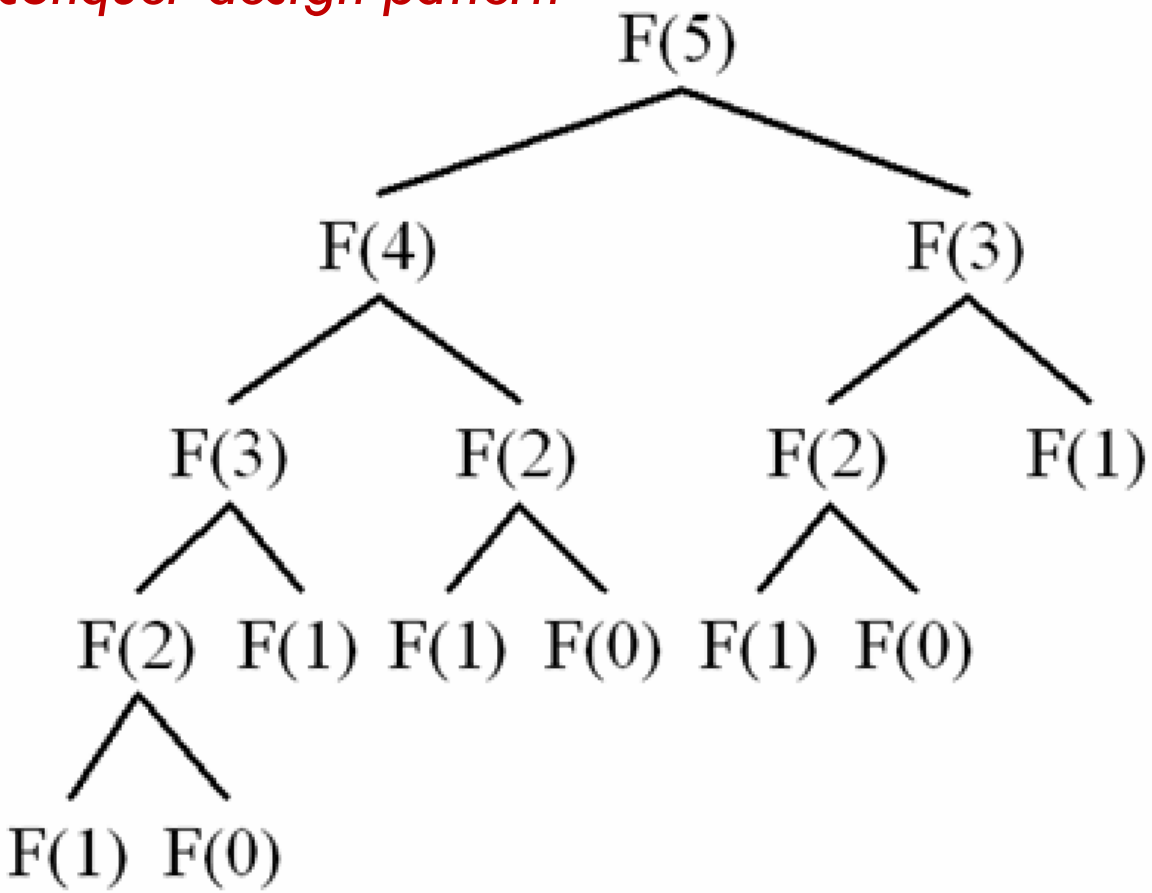
- In mathematics, the **Fibonacci numbers**, commonly denoted F_n, form a sequence, called the **Fibonacci sequence**, such that each number is the sum of the two preceding ones, starting from 0 and 1.
  - $F_0=0$, $F_1=1$ *and* $Fn=F_{n-1}+F_{n-2}$
  - The beginning of the sequence is thus:
  - *0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …*

```
int fib (int n){
int x,y;

if (n<2)
 return n;

x=fib (n-1);
y=fib (n-2);

return x+y;
}
```

Fibonacci Sequence
Serial Code

- How can we parallelize this problem?
  - *Using the divide and conquer design pattern*

```
int fib (int n){
int x,y;

if (n<2)
 return n;

x=fib (n-1);
y=fib (n-2);

return x+y;
}
```



Fibonacci Sequence Multi-threaded Code


```
int Fibonacci (int n){
int fib;

#pragma omp parallel
{
#pragma omp single
fib=kernel(n);
}

return fib;

}
```

```
int kernel (int n){
int x,y;

if (n<2) return n;

#pragma omp task shared(x) //x must be
shared otherwise, it will be lost when the
task ends. x is undefined outside the task
 x=kernel(n-1);

#pragma omp task shared(y)//y must be
shared otherwise, it will be lost when the
task ends. y is undefined outside the task
 y=kernel(n-2);

#pragma omp taskwait
return x+y;
}
```

Parallelize Pi example using divide and conquer design pattern (1)

- *The main idea behind this implantation is to recursively split the pi program's loop into half until the number of iterations is smaller than a threshold.*

- **Step1.** We want just one thread to execute the Pi routine
  - All the threads will be created from a single source thread

```
double divide_conquer(){

int i;
double step, pi, sum=0.0;

step=1.0/(double) num_steps;

#pragma omp parallel
{
#pragma omp single
sum=pi_kernel (0,num_steps, step);
}

pi = step * sum;

return pi;
}
```

Parallelize Pi example using divide and conquer design pattern (2)

- **Step2**. Is the problem small enough to compute it?
  - Yes. If so use one thread to compute it
  - No. Split it to two tasks, and recursively execute the pi routine
- **Step3**. merge the results of the sub-problems

```
double pi_kernel(int start, int finish, double step){
   int i,blk; double x,sum=0.0,sum1,sum2;

if (finish-start < BULK){ //if problem small enough
 for (i=start; i<finish; i++){
  x=(i+0.5)*step;
  sum = sum + 4.0 / (1.0 + x*x);
    }
 }
else { blk=finish-start;

 #pragma omp task shared(sum1)
 sum1=pi_kernel(start,finish-blk/2, step);
 #pragma omp task shared(sum2)
 sum2=pi_kernel(finish-blk/2, finish, step);
 #pragma omp taskwait
 sum=sum1+sum2;
}
return sum;
}
```

Further Reading

- OpenMP Architecture, available at https://www.openmp.org/
- Guide into OpenMP: Easy multithreading programming for C++, available at https://bisqwit.iki.fi/story/howto/openmp/#ParallelConstruct
- OpenMP Application Programming Interface Examples, available at https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwiOip2R-rrqAhX8XRUIHa5HC0QQFjAAegQIAxAB&url=https%3A%2F%2Fwww.openmp.org%2Fwp-content%2Fuploads%2Fopenmp-examples-4.5.0.pdf&usg=AOvVaw3BDlLKC3VhdJI1iTj1RE_p
- GNU libgomp available at https://gcc.gnu.org/onlinedocs/libgomp/index.html