OpenMP

http://openmp.org/wp
https://computing.llnl.gov/tutorials/openMP
http://openmp.org/mp-documents/ntu-vanderpas.pdf
http://msdn.microsoft.com/en-us/library/tt15eb9t.aspx

What is OpenMP?

OpenMP: Open Multi-Processing*

An Application Program Interface (API) that may be used to explicitly direct ***multi-threaded, shared memory parallelism***

Long version: OpenMP : **Open** specifications for **Multi-Processing** via collaborative work between interested parties from the hardware and software industry, government and academia.

History, source for information

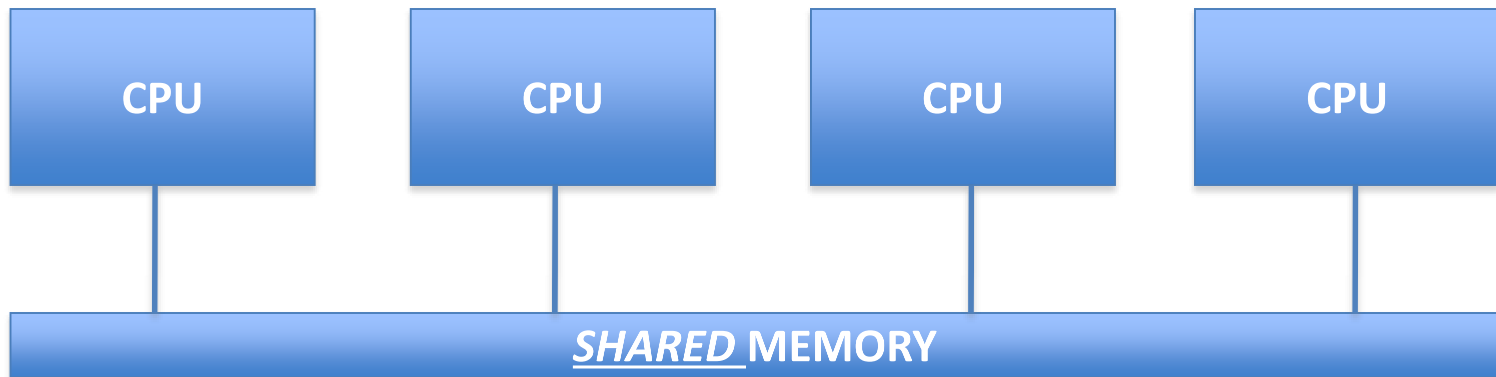
In the early 90's, vendors of shared-memory machines supplied similar, directive-based, Fortran programming extensions: The user would augment a serial Fortran program with directives specifying which loops were to be parallelized.

The compiler would be responsible for automatically parallelizing such loops across the SMP processors.

The OpenMP standards:

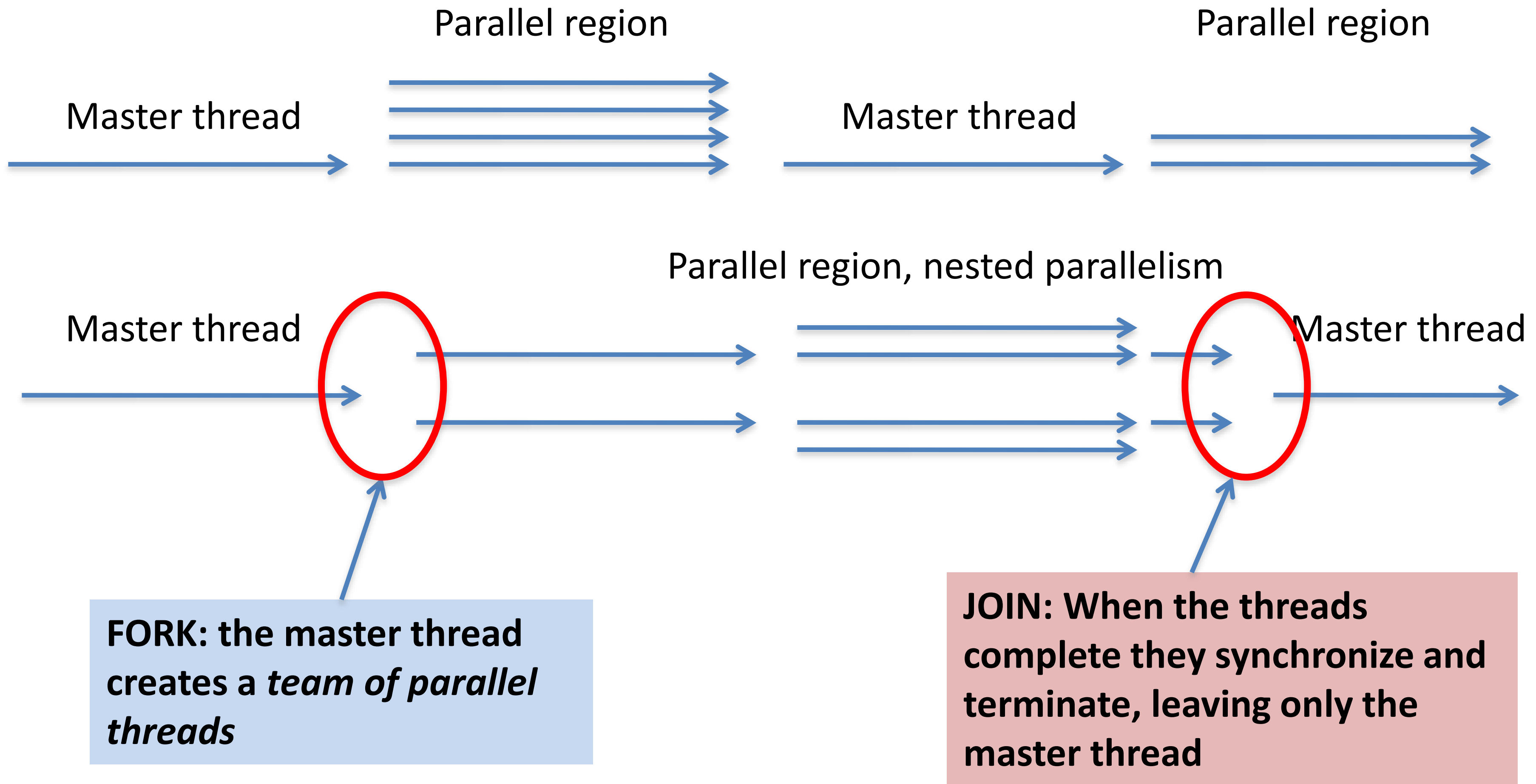
Version	Year
1.0 Fortran	- 1997,
1.0 C/C++	- 1998
2.0 Fortran	- 2000
2.0 C/C++	- 2002
2.5 F/C/C++	- 2005
3.0 F/C/C++	- 2008
3.1	- 2011
4.0	- 2013

http://openmp.org/wp/

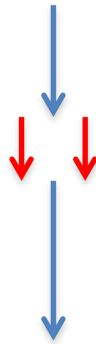


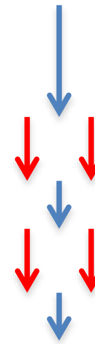
Idea: many CPUs can access the same memory space.
All data stored in this memory can be shared.
Different CPUs can:
1) operate on different chunks of memory (executing the same instruction)
2) execute different instructions while operating on the same data
3) mix of 1 and 3

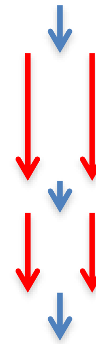
OpenMP Programming Model



An existing sequential code







OpenMP allows incremental parallelization of an *existing* code. Starting with a working sequential code one can create parallel regions one at a time.

Message passing approach requires more substantial changes in the code.

Threads

We have 12 core processor

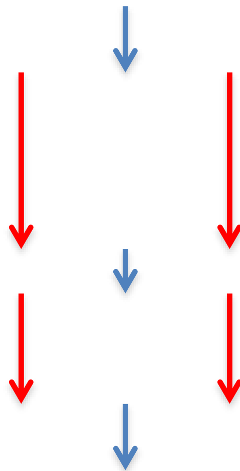
Each core can run up to 8 hardware threads

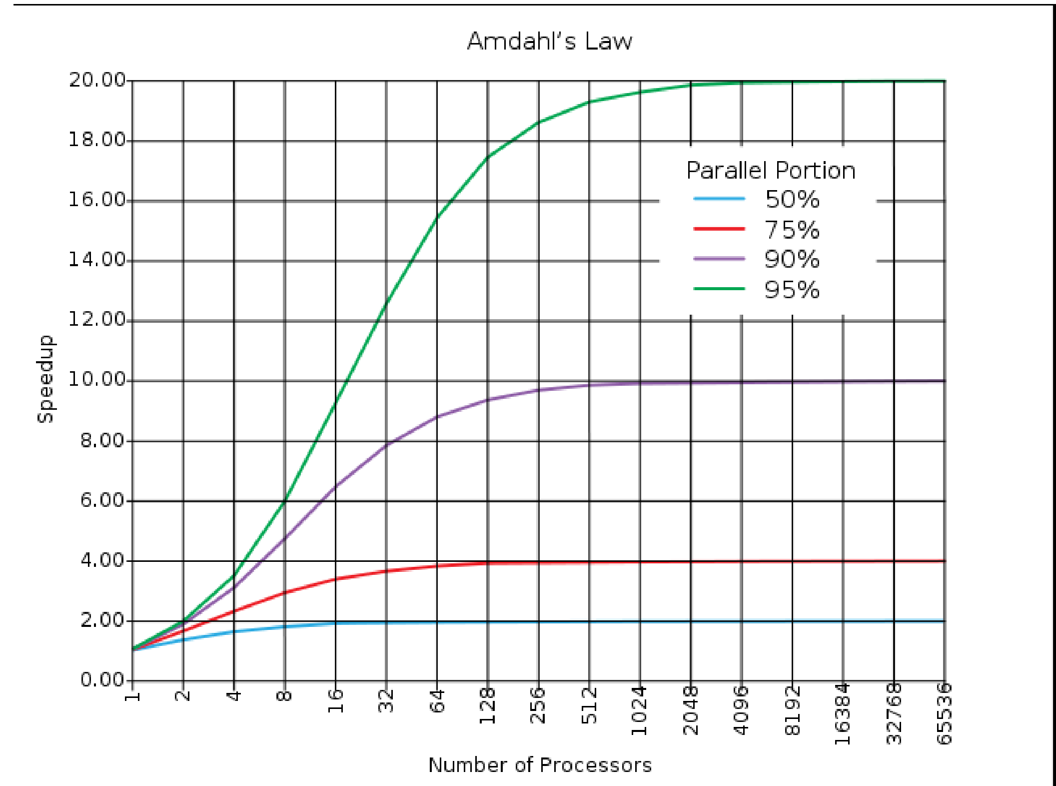
What is the total number of threads available for an application?

Who is managing all those threads?

Threads and cores , affinity

OpenMP and Amdahl's Law

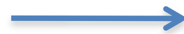




OpenMP: example

```
int main () {
  int i,N=5000;
  double dx  =0.1;
  double *x;
  x = new double[N];
  for (i = 0; i < N; ++i)
     x[i] = i*dx;
//do something with x

  delete[] x;
  return 0;
}
```



```
#include <omp.h>

int main () {
  int i,N=5000;
  double dx  =0.1;
  double *x;
  x = new double[N];
#pragma omp parallel
{
#pragma omp for
  for (i = 0; i < N; ++i)
     x[i] = i*dx;
}
//do something with x

  delete[] x;
  return 0;
}
```

What OpenMP offers?

OpenMP is an **explicit programming model**, offering the programmer full control over parallelization.

```
#include <omp.h>

int main () {
  int i,N=5000;
  double dx  =0.1;
  double *x;
  x = new double[N];
#pragma omp parallel
{
#pragma omp for
  for (i = 0; i < N; ++i)
     x[i] = i*dx;
}
//do something with x

  delete[] x;
  return 0;
}
```

OpenMP is **NOT**:

Meant for **distributed memory** parallel **systems**.

Necessarily implemented **identically** by all vendors.

Guaranteed to make the most efficient use of shared memory.

Required to check for data dependencies, data conflicts, race conditions, or deadlocks.

Required to check for code sequences that cause a program to be classified as non-conforming.

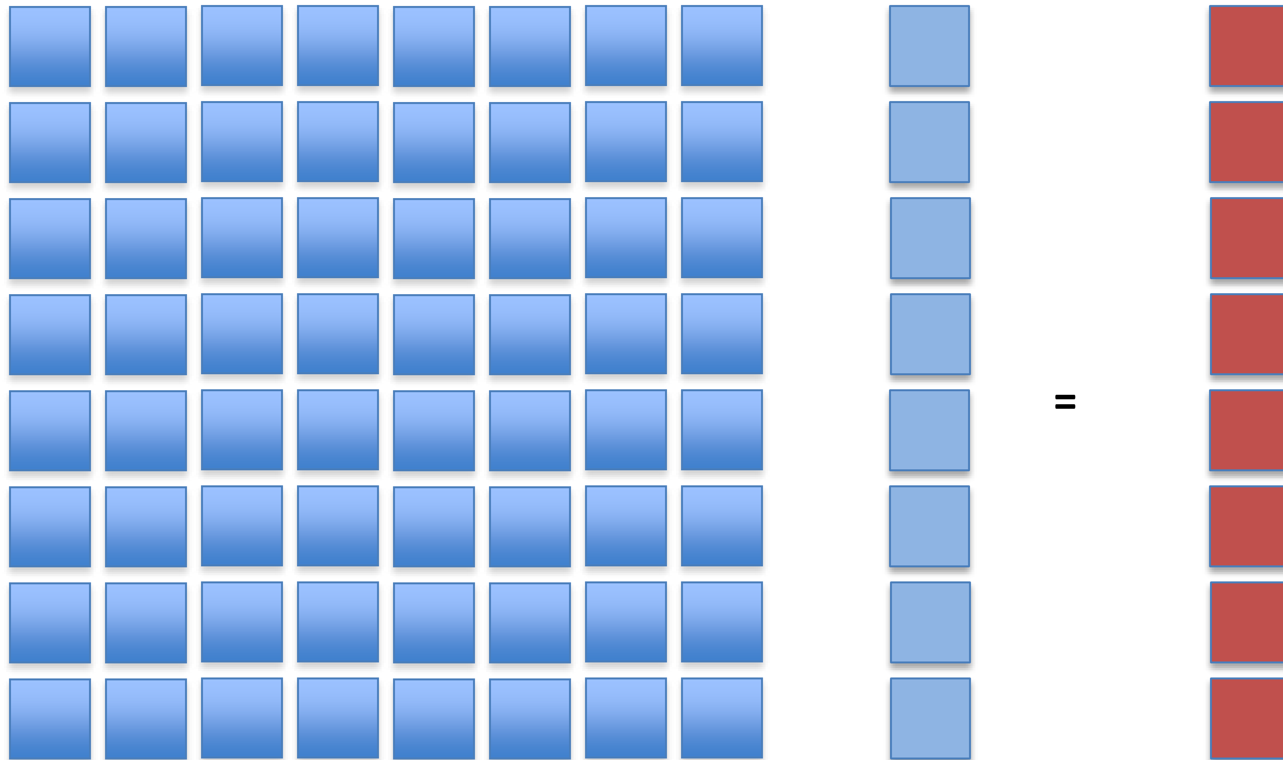
Meant to cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization.

Designed to guarantee that input or output to the same file is synchronous when executed in parallel.

The programmer is responsible for synchronizing input and output!

OpenMP provides the capability to implement both coarse-grain and fine-grain parallelism

A u =b



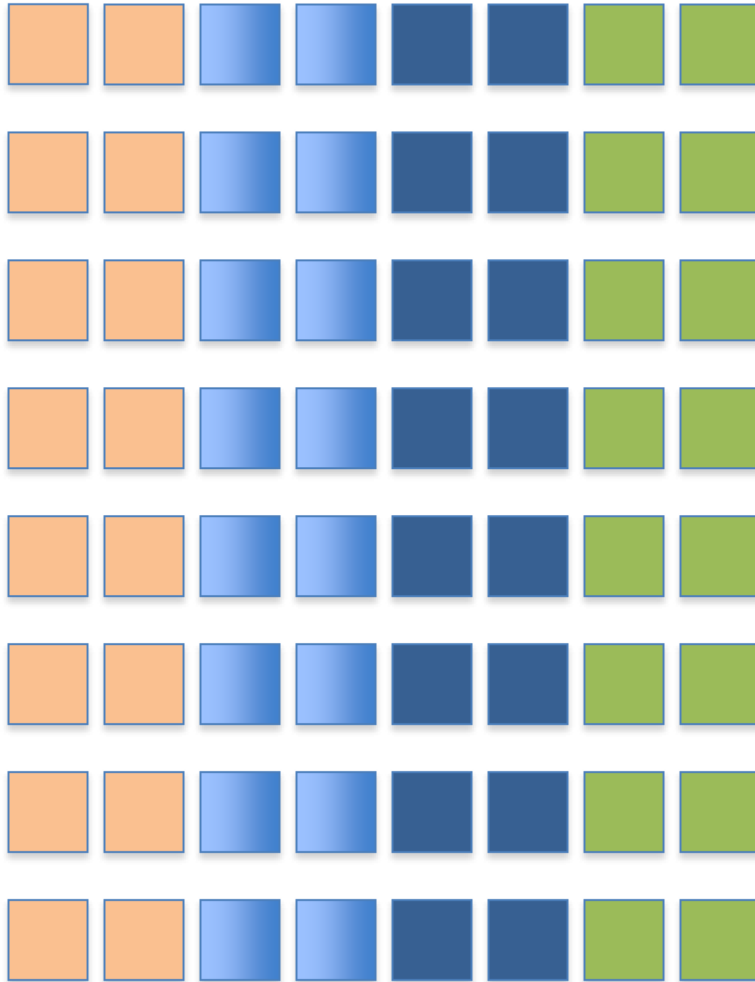
Granularity

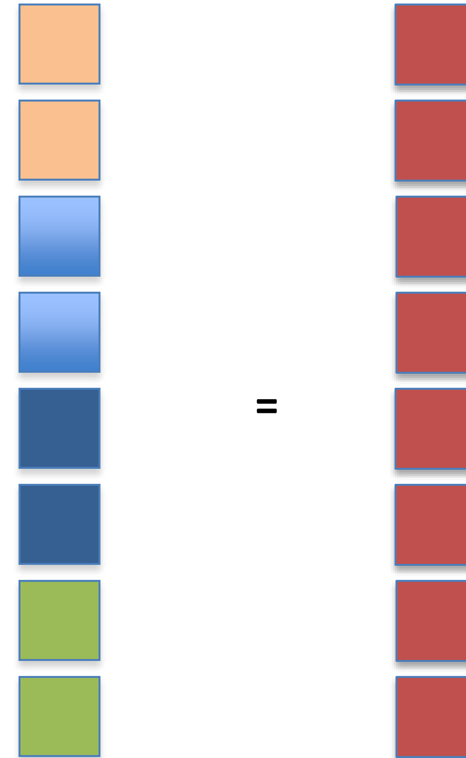
In parallel computing, granularity is a qualitative measure of the ratio of computation to communication (or synchronization).

Coarse: *relatively large amounts of computational work are done between communication events*

Fine: *relatively small amounts of computational work are done between communication events*

Fine-grain parallelism

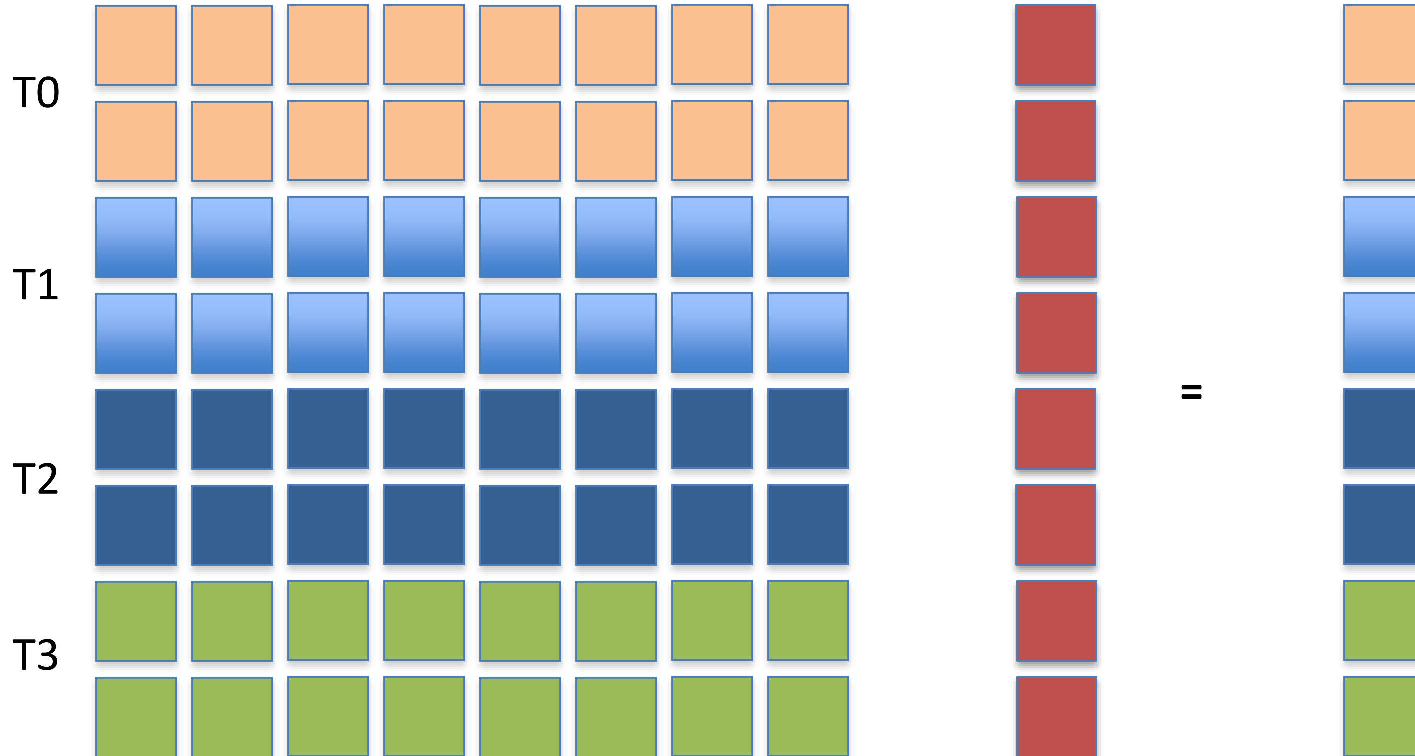




=

```
for (row = 0; row < R; ++row){
  sum = 0.0;
#pragma omp for (reduction +:sum)
  for (col=0; col < C; ++col)
    sum += A[row][col]*u[col]
  b[row] = sum;
}
```

Coarse-grain parallelism



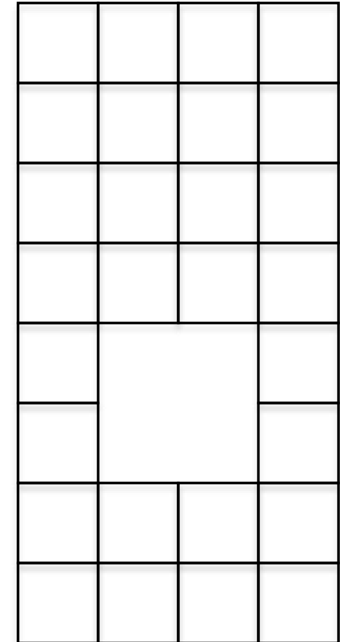

```
#pragma omp for
for (row = 0; row < R; ++row)
  b[row] = _ddot(N,A[row],u);
```

Coarse-grain parallelism

Assume we solve a PDE using finite element discretization.

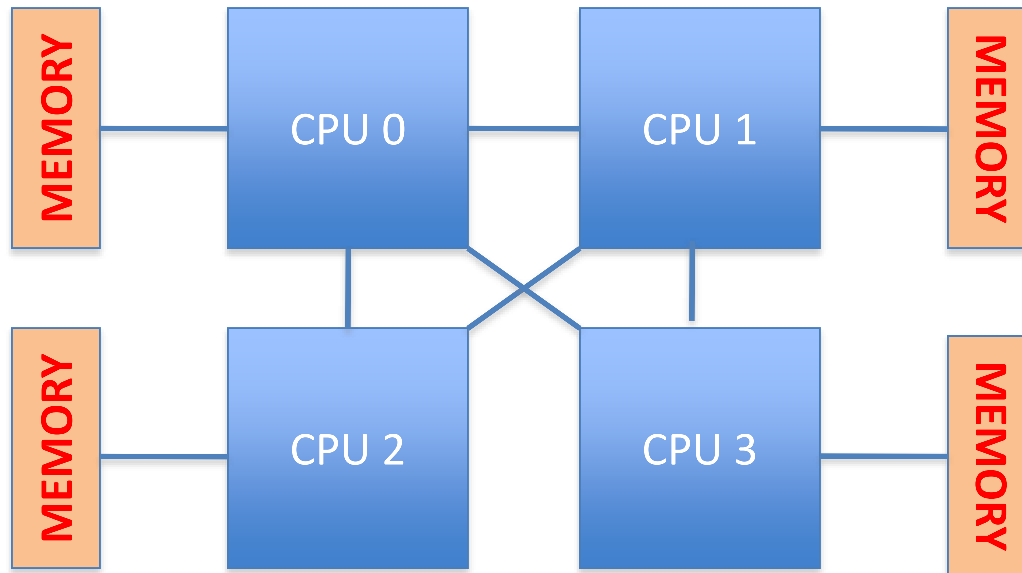
We want to compute derivatives of some function **u**.

The derivatives can be computed locally (element-wise) using some derivative operator: $du^e = A^e u^e$.

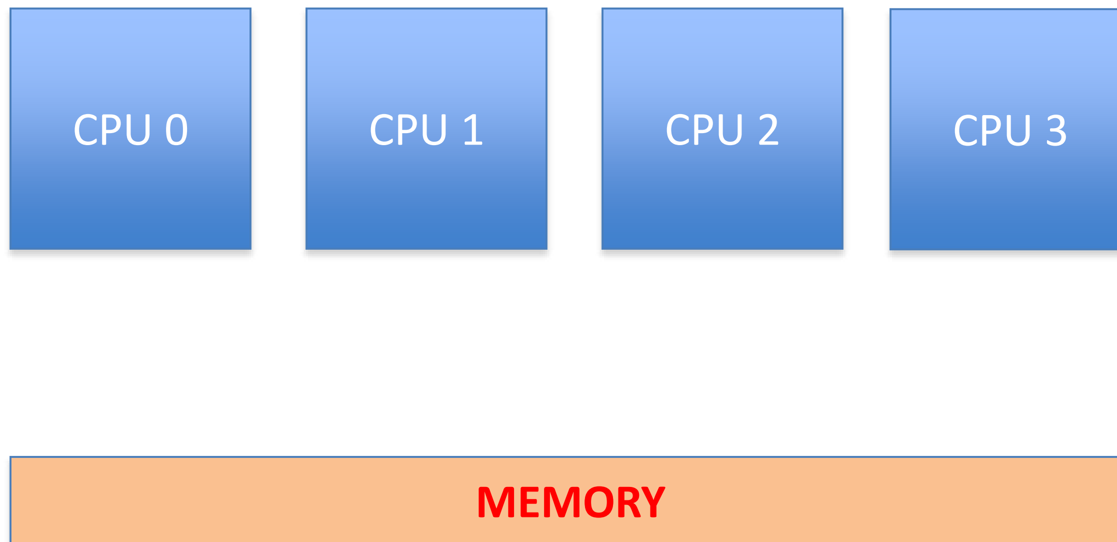


```
#pragma omp for
for (element = 0; element < Nel; ++element)
   du[element] = _dgemv ( Ae [element], u[element] );
```

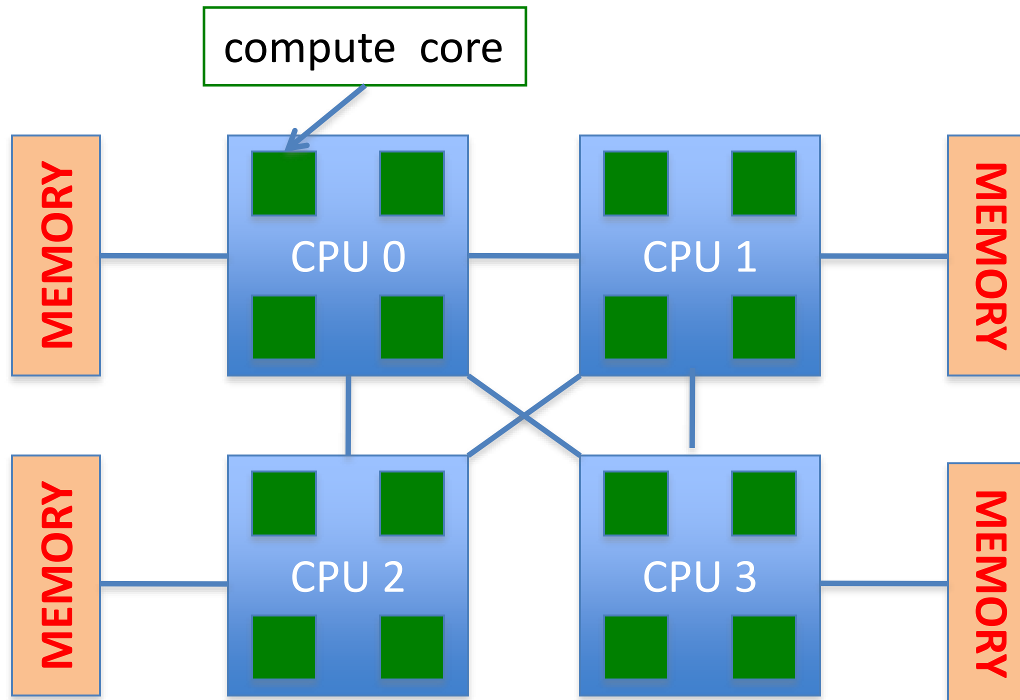
When shall we use a fine-grain parallelism and when coarse-grain?



When shall we use a fine-grain parallelism and when coarse-grain?



When shall we use a fine-grain parallelism and when coarse-grain?



Lets get into details

Most OpenMP parallelism is specified through the use of compiler directives which are imbedded in C/C++ or Fortran source code.

The OpenMP provides for dynamically altering the number of threads which may used to execute different parallel regions. (not always supported).

The OpenMP provides for dynamically altering the number of threads which may used to execute different parallel regions (not always supported).

OpenMP specifies nothing about parallel I/O

```
#include <omp.h>

int main () {
  int i,N;
  double dx  =0.1;
  double *x;
  .
#pragma omp parallel
{

}
  .
  return 0;
}
```

Serial region: only master threads executes

Parallel region: executed by all threads

```
#include <omp.h>

int main () {
  int i,N;
  double dx  =0.1;
  double *x;
  .
#pragma omp parallel
{


}
  .
  return 0;
}
```

i, N, dx, x – are shared variables

```
#include <omp.h>

int main () {
  int i,N;
  double dx  =0.1;
  double *x;
  .
#pragma omp parallel
{
  int j;
  double a,b;
  double *y;

}
  .
  return 0;
}
```

j, a, b, y– are private variables

```
#include <omp.h>

int main () {
  int i,N;
  double dx  =0.1;
  double *x;
  .
#pragma omp parallel private (N)
{
  int j;
  double a,b;
  double *y;

}
  .
  return 0;
}
```

N – is a private variable

C / C++ Directives Format

#pragma omp	directive-name	[clause, ...]	newline
Required for all OpenMP C/C++ directives	A valid OpenMP directive. Must appear after the pragma and before any clauses.	Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.	Required. Precedes the structured block which is enclosed by this directive.

```
#pragma omp    parallel   shared (a, b, c, x)    private (beta, pi)
{

}
```

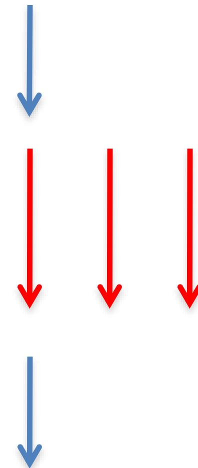
PARALLEL Region Construct

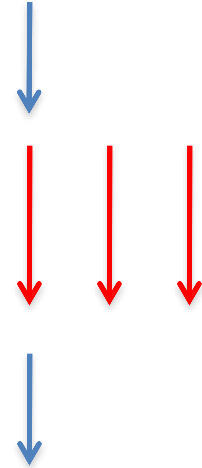
Parallel region is the fundamental OpenMP parallel **construct.**

A **parallel region is a block of code** that will be executed by multiple threads.

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

When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team.





When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team.

Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.

There is an implied barrier at the end of a parallel section. Only the master thread continues execution past this point.

```
.
double x = 0;
#pragma omp parallel
{

  x = x *2.0;
  .
}
 .
```

Be aware that all threads are executing the same instruction!

If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.

```
.
void function (int N, double *x){

#pragma omp parallel
{
 #pragma omp for
    for (int k = 0; k < N; ++k){
      x[k] = rand();
     if (x(k] < 0.5)  return;
    }
   .
}
}
```

Few more rules on PARALLEL region

A parallel region must be a structured block that does not span multiple routines or code files.

It is illegal to branch into or out of a parallel region.

```
.
.
#pragma omp parallel
{
  .
  goto  stage2;
  .
}
stage2:
```

How Many Threads can we use?

Evaluation of the **IF clause.**
if n <= k → serial code.
Only a single IF clause is permitted!
Question: when shall we use it?

```
#pragma omp parallel if (n > k)
{
}
```

Setting of the **NUM_THREADS clause.**
Only a single NUM_THREADS clause is permitted

```
#pragma omp parallel num_threads(4)
{
    int i = omp_get_thread_num();
        printf("Hello from thread %d\n", i);
}
```

Use of the **omp_set_num_threads() function**

```
omp_set_num_threads(4);
#pragma omp parallel
  {
   int i = omp_get_thread_num();
      printf("Hello from thread %d\n", i);
}
```

Setting of the **OMP_NUM_THREADS environment variable**

```
export OMP_NUM_THREAD=8
./a.out
```

Parallel for

```
int k;
double *x, *y, *z, *a, *b, *c;

#pragma omp parallel
{
 #pragma omp for
    for (k = 0; k < N; ++k)
      z[k] = x[k] * y[k];

#pragma omp for
    for (k = 0; k < N; ++k)
      a[k] = b[k]+c[k];

#pragma omp for
   for  (k = 1; k < N-1; k++)
      d[k] = (z[k-1]+z[k+1]) / a[k];
}
}
```

Implicit barrier

Implicit barrier

Implicit barrier

Parallel for

```
int k;
double *x, *y, *z, *a, *b, *c;

#pragma omp parallel
{
 #pragma omp for nowait
    for (k = 0; k < N; ++k)
      z[k] = x[k] * y[k];

#pragma omp for
    for (k = 0; k < N; ++k)
      a[k] = b[k]+c[k];

#pragma omp for
   for  (k = 1; k < N-1; k++)
      d[k] = (z[k-1]+z[k+1]) / a[k];
}
```

Implicit barrier

Implicit barrier

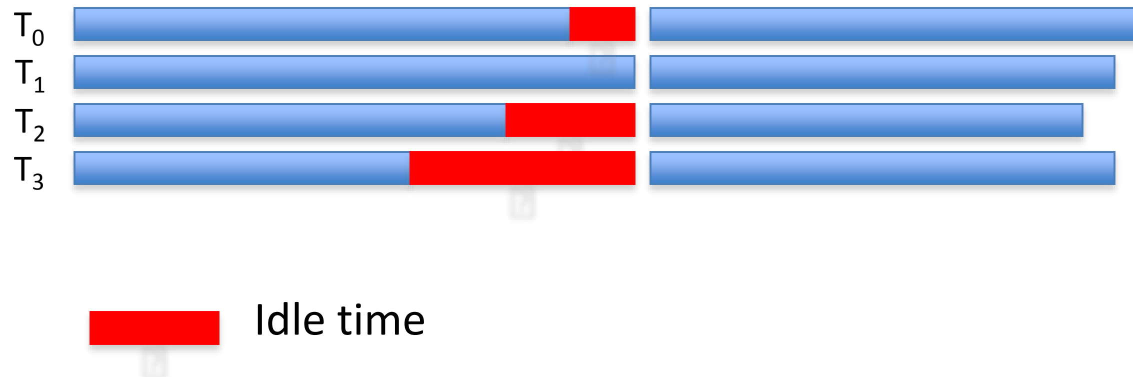
```
int k;
double *x, *y, *z, *a, *b, *c;

#pragma omp parallel
{
 #pragma omp for nowait
    for (k = 0; k < N; ++k)
      z[k] = x[k] * y[k];

#pragma omp for nowait
    for (k = 0; k < N; ++k)
      a[k] = b[k]+c[k];

#pragma omp barrier
#pragma omp for
   for  (k = 1; k < N-1; k++)
      d[k] = (z[k-1]+z[k+1]) / a[k];
}
```

Effect of barriers



Minimizing the idle time reduces the total execution time and improves scalability!

Designing your code look for opportunities to eliminate barriers

Private and shared variables

Private

No storage association with original object

All references are to the local object

Values are undefined on entry and exit

Shared

Data is accessible by all threads in the team

All threads access the same address space

All threads can modify the data

```
TET      *dptr;
TET      *Element;
#pragma omp parallel
{
 #pragma omp for shared(Element) private
     (dptr)
    for (i = 0; i < N; ++i){
      dptr = &Element[i];
     }
}
```

Private and shared variables

```
TET      *Element;
#pragma omp parallel
{
TET      *dptr;
 #pragma omp for shared(Element)
    for (i = 0; i < N; ++i){
      dptr = &Element[i];
     }
}
```

Variable *dptr is declared inside parallel region, hence it is private

A shared variable exists in only one memory location and all threads can read or write to that address.

It is the programmer's responsibility to ensure that multiple threads properly access SHARED variables (such as via CRITICAL sections).
If the SHARED variables is not modified within a loop - no need to synchronize an access to it, otherwise synchronization might be required.

Private and shared variables

```
TET      *Element;
TET      *dptr;
#pragma omp parallel
{
#pragma omp for private (dptr) shared
      (Element)
    for (i = 0; i < N; ++i){
      dptr = &Element[i];
     }
}
```

PRIVATE variables behave as follows:
A **new object of the same type** is declared once for each thread.

All **references to the original object are replaced with references to** the **new object**.

Variables declared PRIVATE should be assumed to be **uninitialized** for each thread.

Private and shared variables

```
Double *x;
double sum = 0;
#pragma omp parallel
{
#pragma omp for firstprivate (sum) shared (x)
    for (i = 0; i < N; ++i)
      sum = sum + x[i];

}
```

Variables declared PRIVATE should be assumed to be **uninitialized** for each thread.

We may use FIRSTPRIVATE clause to initialize the private variable.

For a FIRSTPRIVATE clause on a parallel construct, the initial value of the new private object is the value of the original object that exists immediately prior to the parallel construct for the thread that encounters it.

Private and shared variables

For additional information see
THREADPRIVATE
LASTPRIVATE
COPYPRIVATE

Reduction operations

```
#include <omp.h>
main ()  {
   int   i, N = 100;
   double *x;
   x = new double[N];
   function_evaluate(x);

  double sum = 0;
#pragma omp parallel
{
#pragma omp for reduction(+:sum)
  for (i = 0; i < N; ++i)
     sum += x[i];
}
  fprintf(stdout,” sum = %f\n“,sum);
  delete[] x;
  return 0;
}
```

A private copy for each list variable (in our case - “sum”) is created for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

Reductions can be applied on scalar variables only (at least upto OpenMP 2.5)

SECTIONS

The SECTIONS directive is a non-iterative work-sharing construct.

It specifies that the enclosed **sections of code are** to be **divided among the threads** in the team.

Independent SECTION directives are nested within a SECTIONS directive.

Each SECTION is executed once by a thread in the team.

Different sections may be executed by different threads. It is possible that for a thread to execute more than one section if it is quick enough and the implementation permits such.

```
#pragma omp parallel
{
#pragma omp sections nowait
{
  #pragma omp section
   {
   }
   #pragma omp section
   {
   }
   #pragma omp section
   {
   }
}
}
```

Use sections for task (functional) parallelism! Also maybe useful for coarse-grain parallelism.

Sequential blocks inside PARALLEL region

```
for (i = 0; i < N; ++i)
    function(x[i]);

Read_y_from_a_file(y);


for (i = 0; i < N; ++i)
    function(y[i]);
```

```
#pragma omp parallel
{
#pragma omp for
for (i = 0; i < N; ++i)
    function(x[i]);
}

Read_y_from_a_file(y);

#pragma omp parallel
{
#pragma omp for
for (i = 0; i < N; ++i)
    function(y[i]);
}
```

Sequential blocks inside PARALLEL region: SINGLE

```
#pragma omp parallel
{
#pragma omp for
for (i = 0; i < N; ++i)
    function(x[i]);
}

Read_y_from_a_file(y);

#pragma omp parallel
{
#pragma omp for
for (i = 0; i < N; ++i)
    function(y[i]);
}
```

```
#pragma omp parallel
{
#pragma omp for
for (i = 0; i < N; ++i)
    function(x[i]);

#pragma omp single
Read_y_from_a_file(y);

#pragma omp for
for (i = 0; i < N; ++i)
    function(y[i]);
}
```

Sequential blocks inside PARALLEL region: CRITICAL

```
double  global_sum = 0;
#pragma omp parallel
{
double sum = 0;
#pragma omp for nowait
for (i = 0; i < N; ++i)
    sum += function(x[i]);

#pragma omp critical
global_sum += sum;

}
```

Sequential blocks inside PARALLEL region

The **SINGLE** directive specifies that the enclosed code is to be executed by **only one thread** in the team. May be useful when dealing with sections of code that are not thread safe (such as I/O). May be also useful in hybrid MPI-OpenMP codes. Threads in the team that do not execute the SINGLE directive, wait at the end of the enclosed code block, unless a NOWAIT/nowait clause is specified.

The **CRITICAL** directive specifies a region of code that must be executed by **only one thread at a time.**

Threads synchronization

ATOMIC
FLUSH
BARRIER

OpenMP versus MPI

Why pure MPI applications scale better than OpenMP?

Remember the Amhdal's law ?

"#pragma omp parallel" is a sequential portion of the code and it costs about 60,000 – 80,000 cycles!

Reading a file from disk by less MPI ranks may actually speedup the code execution (if sufficiently many MPI tasks are involved)

Use of barriers isn't helpful for scaling-up

```
#pragma omp parallel
{
#pragma omp for
for (i = 0; i < N; ++i)
    function(x[i]);

#pragma omp single
Read_y_from_a_file(y);

#pragma omp for
for (i = 0; i < N; ++i)
    function(y[i]);
}
```

OpenMP versus MPI: who is going to win?

- Number of cores per compute node/socket will increase – more opportunities to share memory.

- Memory / core ration will most probably decrease – it might be more important to share memory, particularly in “memory-hungry” applications.

- Cost (number of cycles) of creating thread is going down.

- Networks for MPI communication are improved.

- MPI is using shared memory blocks for intra-node communication.

- Use of GPUs allows fine-grain parallelism on the chip plus MPI communication between the cards (currently using CPUs).

Compiler	**Flag**
IBM	-qsmp=omp
Intel	-openmp
PathScale	-mp
PGI	-mp
GNU	-fopenmp