12th International SuperComputing Camp 2021

Virtual

6-11 December 2021

OpenMP Tutorial

Robinson Rivas Suárez
Universidad Central de Venezuela

V-camp

```
har **argv) {

ny_rank;
&argv);
(MPI_COMM_WORLD, &world_size);
k(MPI_COMM_WORLD, &my_rank);

__global__ void initialize_boundaries(real* __restrict__ const a_new, real* __restrict__ const a,
                          const real pi, const int nx, const int ny) {
    for (int iy = blockIdx.x * blockDim.x + threadIdx.x; iy < ny; iy += blockDim.x * gridDim.x) {
```

Robinson

BsC in Computer Science (Maracaibo, Venezuela)

Master in Computer Science (Caracas, Venezuela)

Network Engineer (OIC, Japan)

Overview

- In this tutorial we will discuss OpenMP, the *de facto* standard for Shared Memory architectures
- There is a lot of good material over the Internet. We suggest to follow the OpenMP Consortium official releases
- For this presentation, we used Intel public material
- OpenMP can be run and tested from almost any desktop device, using Windows, Linux or Mac based Operating Systems, as well as they have gcc compiler
- Please read the material ***before*** making exercises!

Overview

- That’s really necessary?
- Why to think about ***shared memory***
- OpenMP and new models of programming
- Everyday platforms
- New trends

Back to school

- Often, we professors teach our students that algorithms must be designed, planned and then programmed as much abstract as possible, i.e. only thinking on the ***problem to be solved*** rather than ***the computer that actually solves it***.
- This way of thinking is so-called abstract design

Real World

- In modern life, however, technical and physical details drives to ***different ways to implement the same idea*** due to hardware constraints
- In this dissertation, I will present two different such ways, based on deep differences between architectural schemas

An opinion

People who are more than casually interested in computers should have at least some idea of what the underlying hardware is like. Otherwise the programs they write will be pretty weird

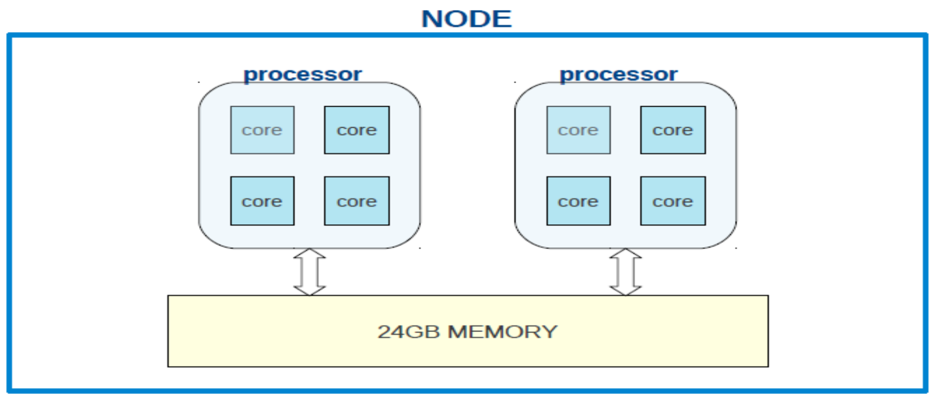
— Donald Knuth,

The Art of Computer Programming,

Volume 1: Fundamental Algorithms

Shared Memory

- More than two processors using the ***same*** memory system, at least ***parts of the same memory system***
- It happens when there are many cores per processor. It was rare in the very early years, but it is ***really common today***



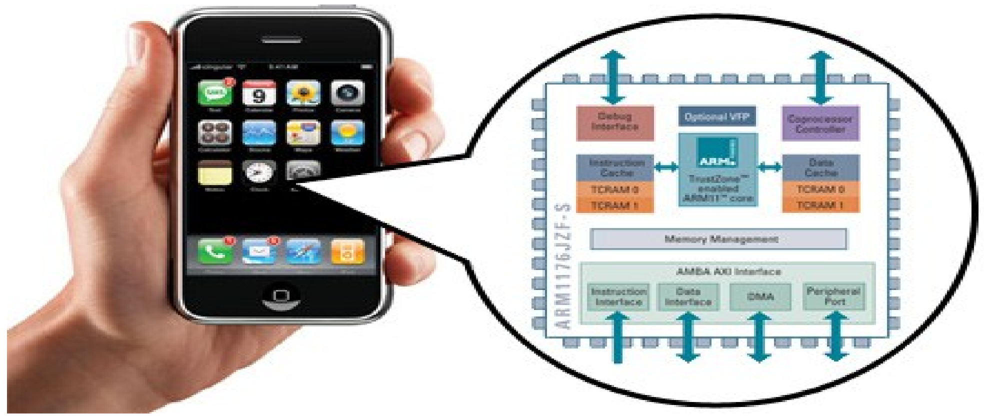
Shared Memory

- Think in a standard smartphone today. Tipically, ARM processors have 4 cores (even more)
- So, ***computers we have in our pockets*** needs to be programmed in an efficient yet reliable way
- HPC in terms of shared

memory is not just for top

500 list supercomputers

… it matters everywhere!!



OpenMP

- OpenMP is an standard set of instructions available for the most important HPC programming languages
- It was originally proposed as an open specification in 1996
- First official draft of the

OpenMP consortium was

released in 1997

OpenMP is managed by the nonprofit technology consortium *OpenMP Architecture Review Board* (or *OpenMP ARB*), jointly defined by a group of major computer hardware and software vendors, including AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Red Hat, Texas Instruments, Oracle Corporation, and more. source: Wikipedia

Advantages of OpenMP

- Portability: code runs in any platform, since ***thread creation*** is encapsulated inside the actual O.S.
- Programmer has not to control the thread behavior. It releases him/her from the heaviest part of thread management
- Because is intended to work in shared memory, OpenMP does not deal with message passing. It eliminates the most common source of errors in parallel programming

Advantages of OpenMP

- Higly scalable: performance usually goes better when code is executed in more cores
- By design, OpenMP permits to keep the sequential code as it is. It allows to use the programs in different environments without need to recompile (on the same architecture)

Scalability

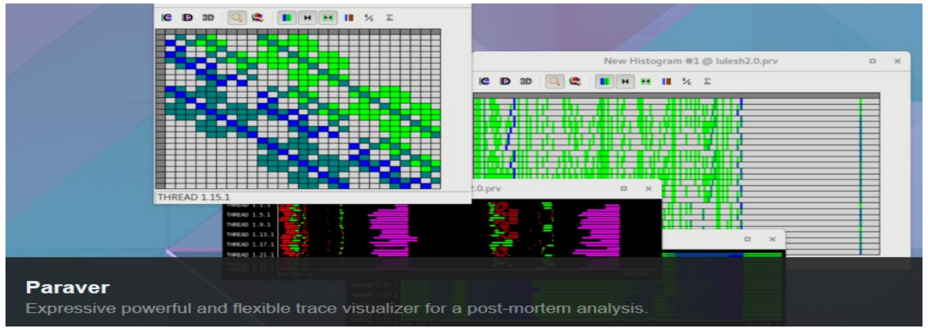
- This advantage is not universal.
  - Some times, having a big number of threads available could lead to slower programs
  - Some times, having more threads than actual cores could lead to faster programs
    - Can **you** think in such scenarios?

OpenMP issues

- Debugging process is not that easy. OpenMP doesn’t have an efficient native error handling mechanism
- By now, internal details of threads are hidden for programmer. It is of course an advantage for readyness and portability, but doesn’t allow programmers to do fine control to improve performance on specific architectures

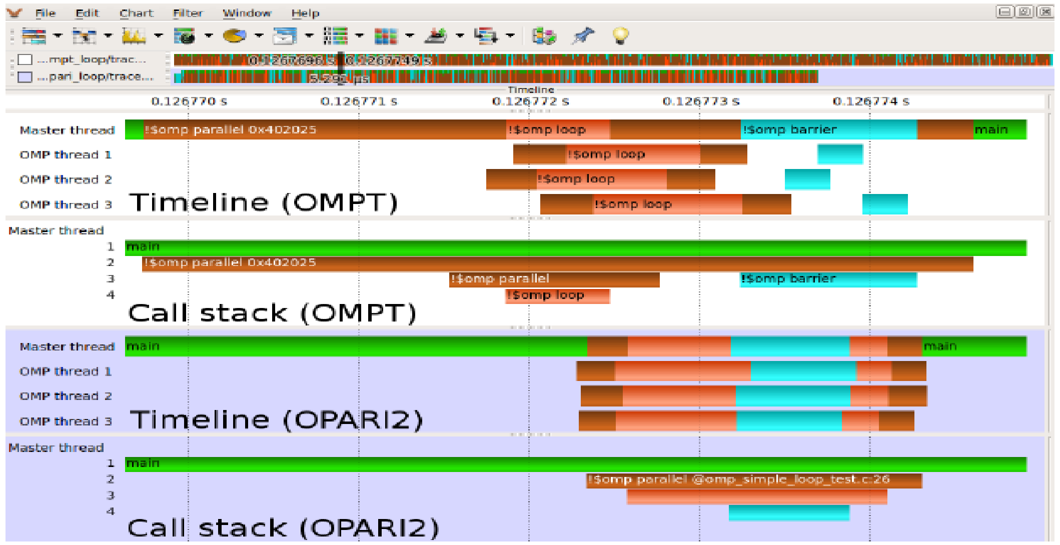
OpenMP issues

- Many tools has being designed to help with this. But are not part of the standard itself.
- Some examples are profilers like Valgrind, VAMPIR, SCORE-P, etc
- BSC (Barcelona-Spain)has developed a few useful tools to help better understanding of OpenMP parallel code


Paraver
Expressive powerful and flexible trace visualizer for a post-mortem analysis.

OpenMP issues

- OMPT (OpenMP Tools Special Working Group) and OPARI2 (Jülich Research, U. of Oregon) were designed to serve as tools to achieve runtime OpenMP indicators
- i.e., OpenMP community is dynamic and active

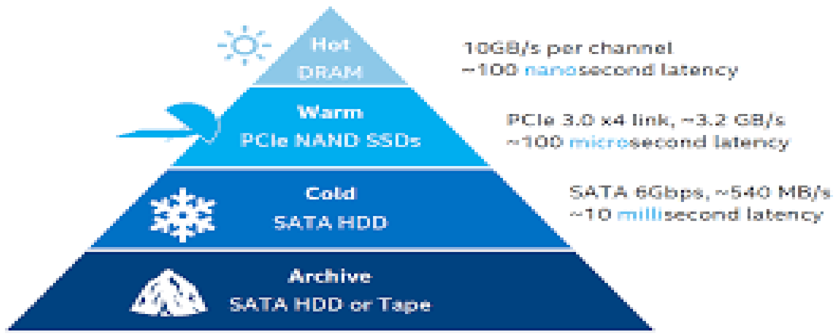


A Comparison between OPARI2 and the OpenMP Tools Interface in the Context of Score-P September 2014

OpenMP issues

- OpenMP doesn’t have memory-related instructions. You can’t deal with cache, memory layers or new architecture models. Everything is seem as ***plain shared*** memory (*)
- Startup time for threads is high compared to sequential time for execution. So, if the problem is really small, OpenMP parallel version could be worst than sequential version

(*) we’ll be back about this later



what OpenMP doesn’t is …

OpenMP is designed as a series of ***compiler directives*** plus an small suite of library functions. So:

- OpenMP is **NOT** a library nor a set of pre-compiled code
- OpenMP is **NOT** designed to run well in a distributed-memory environment

what OpenMP doesn’t is …

- OpenMP is **NOT** a programming language, so you have not to learn new instructions (not so ***much*** new instructions)
- OpenMP does not solve ***your*** problems with parallel code generation. For instance, it doesn’t partition data in any way but array splitting, so you must take care of data structures

WARNING

“Premature optimization is the root of all evil.”

— Donald Knuth,

The Art of Computer Programming,

Volume 1: Fundamental Algorithms

Example 1

```
#include <stdio.h>
int main(char **argv,int argc)
{
#pragma omp parallel
    printf(“Salve, mundi\n”);     //Hello world in latin ;-)
exit (0);
}
```

Compiling & running

- Compile:

```
#gcc  -fopenmp hello.c –o hello
```

In general: `gcc –fopenmp {source.c}`

```
             options -o executable
```

- Run:

```
#export OMP_NUM_THREADS=4
#./hello
```

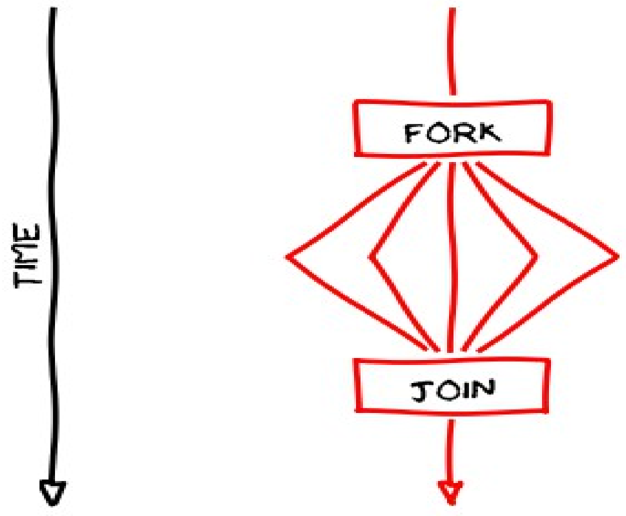
Example 1

Compile & run this code with 2,4 and 8 threads

```
#include <stdio.h>
int main(char **argv,int argc)
{
#pragma omp parallel
    printf("Salve, mundi\n"); //Hello world in latin ;-)
exit (0);
}
```

Number of Threads

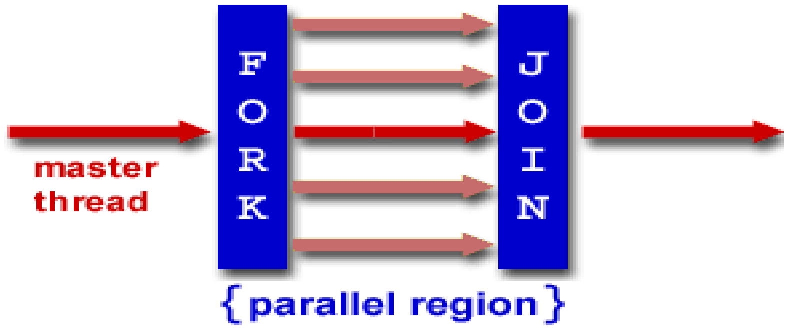
- You must configure the environment variable OMP_NUM_THREADS depending on your O.S.
- This must be done ***before*** program execution. Once compiled, you don't need to recompile. Don't worry, you can decide the number of threads



Fork!

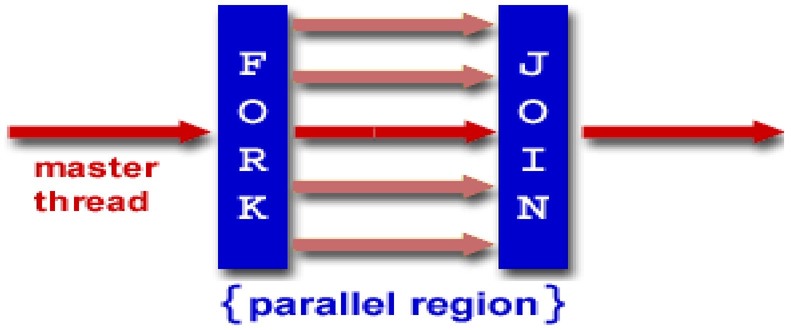
Fork-join model

- Programs are executed sequentially, as a normal monoprocessor one ***until*** a fork instruction is reached
- In this very moment, begins a ***"parallel"*** section. It means, really, that system creates as many threads as are specified by the environment or program



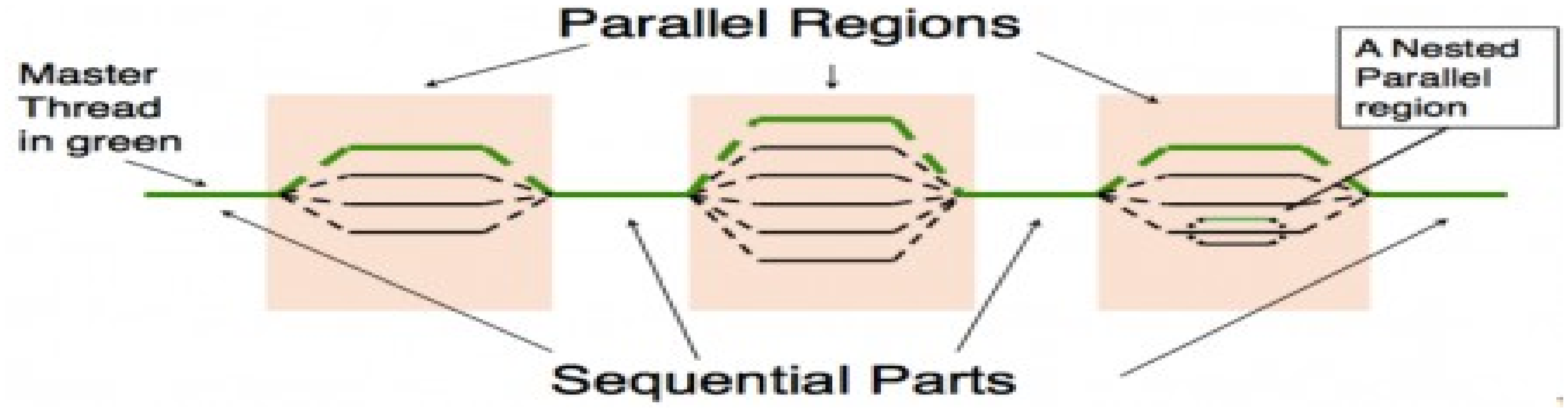
Fork-join model

- ALL threads runs the same code
- Once threads ends their execution, they ***"join"*** into a single thread again
- Internal variables inside threads are disposed. Specification doesn't says nothing about this memory



Fork-join model

- More in detail:



- Threads are not necessarily related to the number of actual cores
  - Threads can be nested
  - Number of threads can be increased/decreased as you need

Private and shared

- OpenMP is designed for shared memory architectures, but of course there is important to share variables in many contexts
- Variables declared ***before*** parallel section, will be **shared** among running threads
- Variables created ***inside*** parallel section (or appropiately defined) are **private** in that context

Example 2

```
int privateX=0;
#pragma omp parallel
{ int sharedX=0;
  privateX++;
  sharedX++;
  printf(“privateX %d sharedX %d\n”,
         privateX,sharedX);
}
```

Example 2

```
int privateX=0;
#pragma omp parallel
{ int sharedX=0;
  privateX++;
  sharedX++;
  printf("Hello world priv %d share %d\n",privateX,sharedX);
}
```

Variable ***privateX*** is actually shared between threads, while variable ***sharedX*** is private, it is, a copy is stored inside each thread. Run this example with 4,8,16,32 cores. Can you see a pattern?

Note: confusion was intentional

Example 2

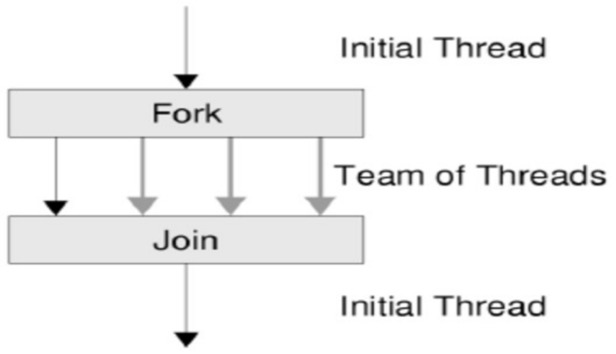
- OpenMP has the modifiers **`shared(vars)`** and **`private(vars)`**
- Modify the example to have a for instruction that modifies a shared variable, like this

```
#pragma omp parallel shared(x)
for (i=0;i<100;i++)
      x++;
```

And then run with 2,4 and 8 threads

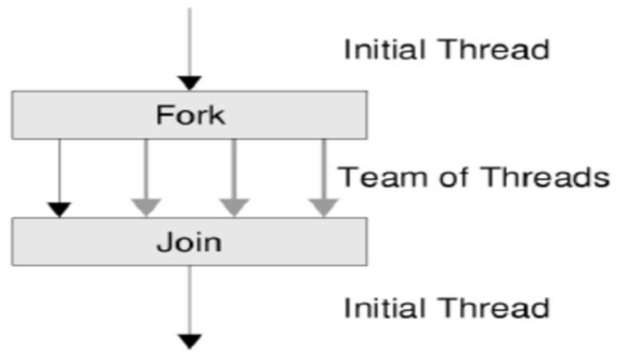
Execution order

- Thread's execution order is non deterministic. OpenMP specification doesn't define any policy or special numbering for threads (except the master one: Thread 0).
- Moreover, threads are intended to be ***really*** concurrent, so any thread can start/finish at any time ***after*** parallel section is reached.



Execution order

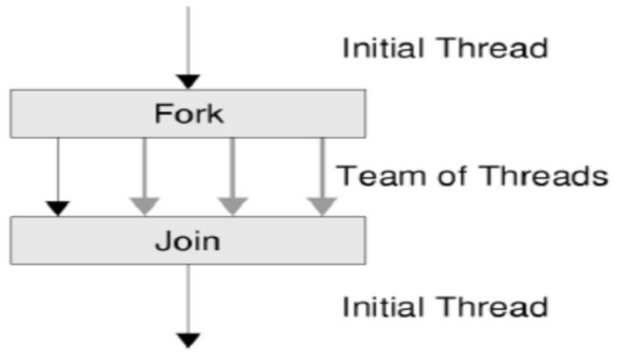
- All threads but the master (numbered as Thread 0) are destroyed when ***all of them*** reaches the end of parallel section
- You can’t do any assupmtion about memory allocation, order of creation nor order of destruction of threads.



Execution order

REMEMBER

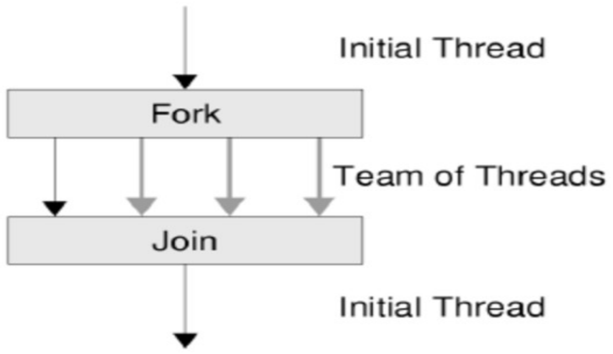
You can’t do any assupmtion about memory allocation, order of creation nor order of destruction of threads.



Execution order

PLEASE REMEMBER !

You can't do any assupmtion about memory allocation, order of creation nor order of destruction of threads.



Data parallelism

- OpenMP implements data parallelism. This is particularly useful for problems where you want to run ***the same instructions over multiple pieces of*** data.
- These pieces of data must be disjunct. But be careful: control over data is your responsibility as programmer
- The most common use is when we deal with arrays or matrixes

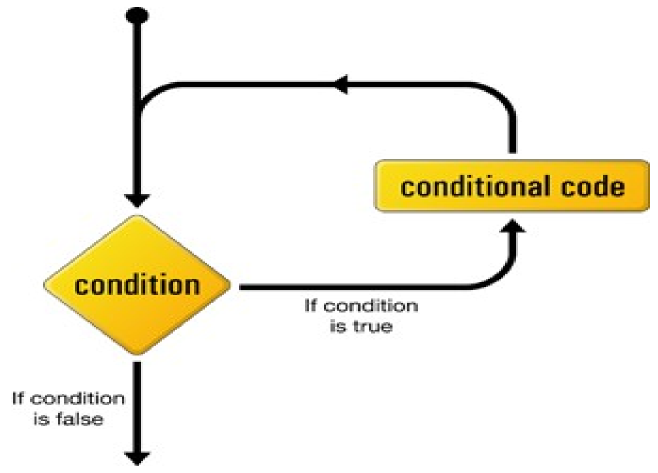
```
for (i=start;i<end;i++)
  x[i] = doSomething(a[i])
```

Remark

- Many real-life problems can be modeled as a combination of functional and data parallelism.
- Just splitting data to make threads deal with small portions seems not to be ***always*** the smartest approach

Data parallelism

- OpenMP designers gave an special treatment for “**for**” instruction (“**DO**” instruction in Fortran)
- So, programmers doesn’t need to change anything on their semantics: OpenMP handles everything (with certain restrictions)

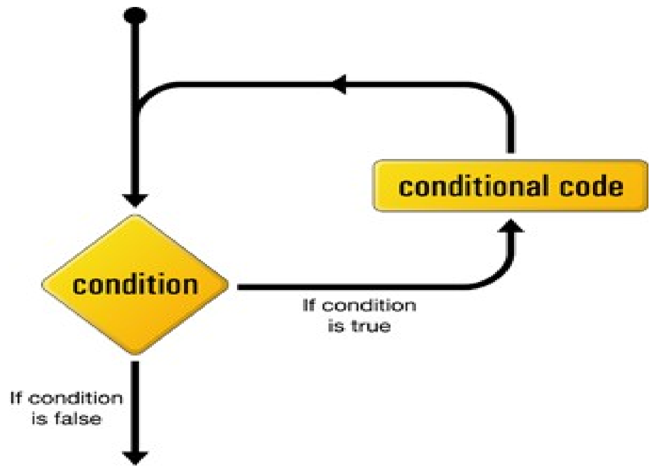


Example 3

```
#include <stdio.h>
int main(char **argv,int argc)
{
#pragma omp parallel for
    for  (int i=0; i<N; i++)
         printf(" i value: %d \n",i);
}
```

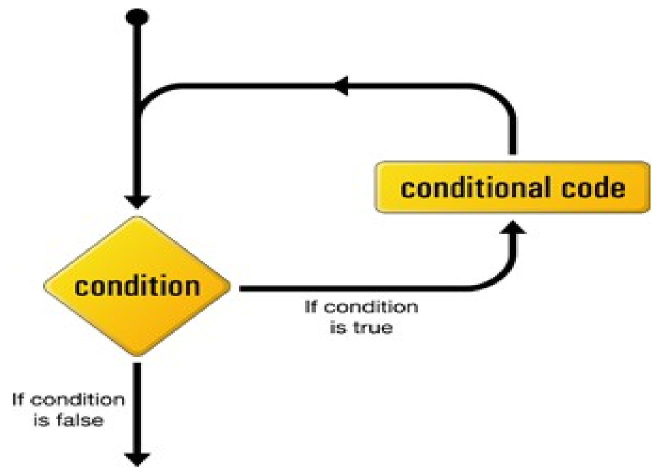
Remarks on parallel for

- **`parallel for`** must be used ***just before*** the **`for`** instruction
- No other variables but ***incremental one*** must be used
- Incremental variable must **not** be modified inside the **`for`** instruction



Remarks on parallel for

- If there are multiple **`for`** instructions, parallelism applies to the ***innermost***
- Each thread uses an internal copy of incremental variable



Example 4

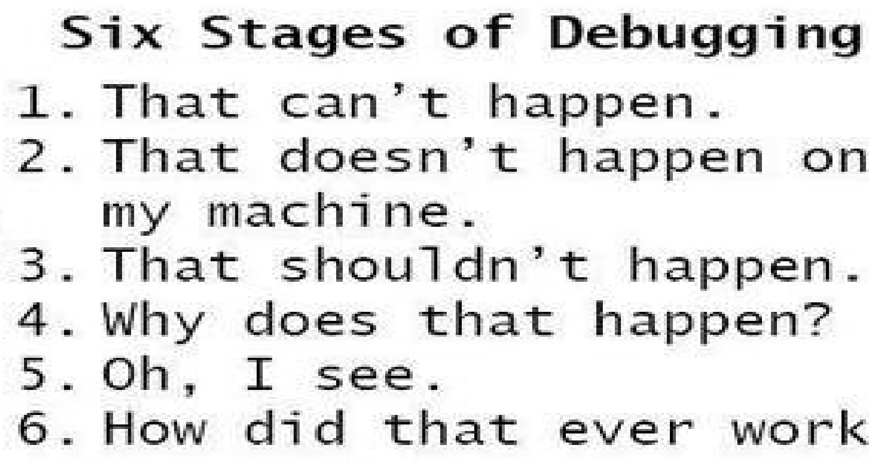
```
float product(float* a, float* b, int N)
  { float sum = 0.0;
#pragma omp parallel for shared(sum)
    for  (int i=0; i<N; i++)
               { sum += a[i] * b[i]; }
      return sum;
    }
```

Compile & run, discuss the results

Parallel execution

- This code works perfect in sequential case (as it is very simple). But, if you run it in many cores, you could get erroneous values most of times
- Where is the bug?

Parallel execution



Six Stages of Debugging

1. That can't happen.
2. That doesn't happen on my machine.
3. That shouldn't happen.
4. Why does that happen?
5. Oh, I see.
6. How did that ever work

Concurrency

- A deeper sight shows that

sum += a[i] * b[i];

Does not prevent that two threads collide ***writing*** the variable “sum”

In fact, if there are many threads, collisions are almost ***sure***

To avoid this, only ***one*** thread must access the variable

Critical Section

- OpenMP has clauses that allows ***only one thread*** to execute specific parts of code
- This part of code is called ***critical section*** and can be executed by ***only one*** thread, even if other lines of code are being executed

Example 5

```
float product(float* a, float* b, int N)
  { float sum = 0.0;
#pragma omp parallel for shared(sum)
     for (int i=0; i<N; i++)
          {
#pragma omp critical
               sum += a[i] * b[i];
          }
      return sum;
  }
```

Critical Section

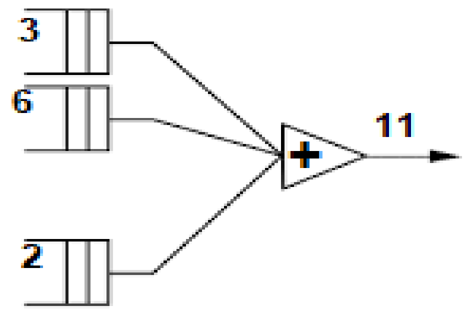
Is there any problem within this solution?

```
#pragma omp critical
            sum += a[i] * b[i];
```

Reductions

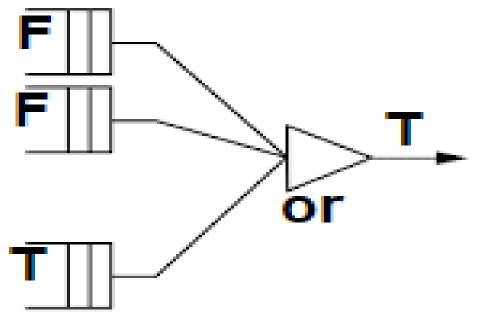
- The best way to avoid this performance degradation, is to keep ***private copies*** of the variable, and ***at the very end*** of computing, sum all of them
- This is achieved in OpenMP with the clause

reduction(operator:variables)



Reduction

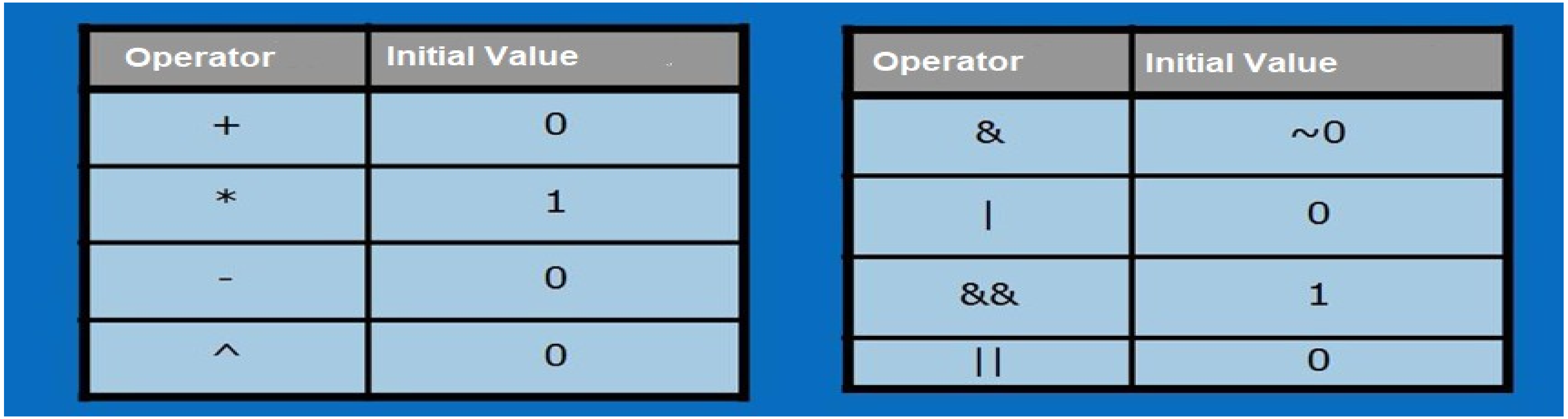
- With this clause, we do say to compiler: “*please keep a private copy of X, initialize it, and apropriately combine all the private X’s at the end*”
- Operators can be: sums, divisions, logical operators, etc



Reduction

- Inside parallel clause:
  - Variables are initialized with ***neutral value*** depending on operation.
  - Local copies are updated independently
  - At end of clause, local copies are join together using ***operator***

Reduction



Operator	Initial Value
+	0
*	1
-	0
^	0

Operator	Initial Value
&	~0
|	0
&&	1
||	0

Example 6

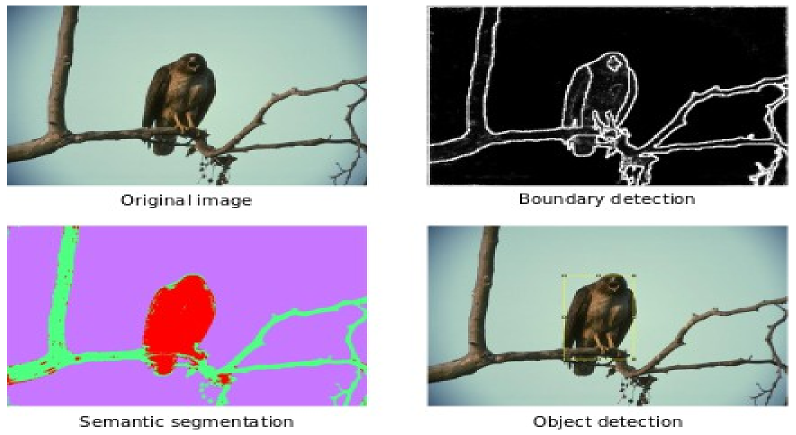
```
float product(float* a, float* b, int N)
  { float sum = 0.0;
#pragma omp parallel for shared(sum) reduction(+:sum)
    for(int i=0; i<N; i++)
       { sum += a[i] * b[i];
       }
 return sum;
}
```

Comment on parallel for

- Since matrixes operations are, by far, the most common parallel operation in Scientific Computing, ***parallel for*** clause is very important in terms of performance
- You must be VERY CAREFUL about unexpected side-effects.

Load Balance

- Not all the tasks derived from a ***parallel for*** will consume the same time. For instance, think about probabilistic algorithms or filters applied to images
- In such cases, some threads will work harder than others



Load Balance

- Suppose you have 12 tasks, whose times (in seconds) are these:

Time: {1, 3, 40, 30, 5, 6, 200, 100, 30, 1, 2, 150}

Total sequential time: **568 min**

So, if you have two threads, these are the expected execution times:

Thread 0: {1, 3, 40, 30, 5, 6} = 85 min

Thread 1: {200, 100, 30, 1, 2, 150} = **483 min**

Load Balance

Remember: execution time is as <u>fast</u> as the <u>slowest</u> thread into competition

Load Balance

- However, if you have 3 threads, these are the times:

  Thread 0: {1, 3, 40, 30} = 74 min

  Thread 1: {5, 6, 200, 100} = **311 min**

  Thread 2: {30, 1, 2, 150} = 183 min

- And 4 threads:

  Thread 0: {1, 3, 40} = 44 min

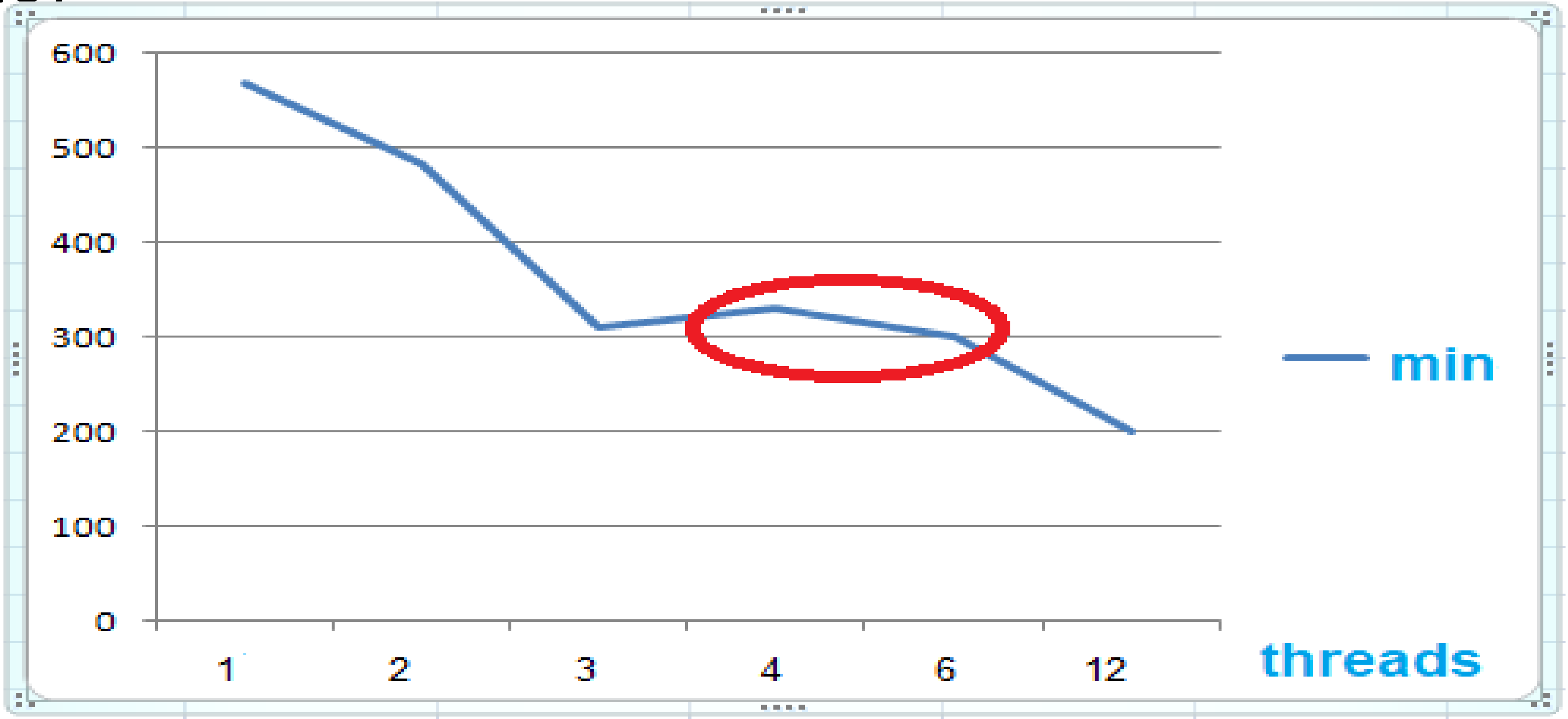
  Thread 1: {30,5, 6 } = 41 min

  Thread 2: {200, 100 , 30} = **330 min**

  Thread 3: {1, 2, 150} = 153 min

Load Balance

- This is the time for different number of threads (worst time for each scenario)



Load Balance

- A way to avoid load unbalance, is scheduling jobs
- Threads can take dinamically as much work as they can deal with
- This leads to more efficient execution. Think in tasks as a card dealer assigning jobs as threads ends its executions

Schedule clause

- This clause indicates OpenMP to divide the N indexes among T threads in different ways.

- schedule(static)
  - Divides indexes into equal chunks of size N/T. Default
- schedule(static, K)
  - Assign chunks of size K using ***round robin*** method

Schedule clause

- schedule(dynamic)
  - Assigns ***ONE*** index to each Thread. As well as each Thread ends its execution, a new index is delivered
- schedule(dynamic, K)
  - The same as above, but with chunks of size K

Schedule clause

- schedule(guided)
  - Uses an exponential formula to decrease the size of chunk
- schedule(guided, K)
  - The same as above, but chunks starts with size K
- schedule(runtime)
  - Depends on the environment variable OMP_SCHED

When do we use SCHEDULE?

- **STATIC**: all tasks are similar on complexity and time execution
- **DYNAMIC**: tasks are quite different in complexity or execution time. Usually, due to probabilistic behavior
- **GUIDED**: when workload is variable but not so much, and we want planning time to be small.

Remark

EACH TIME you have an scheduler, *somebody* has to work coordinating threads

ALWAYS

Example 7

```
#pragma omp parallel for schedule (static, 8)
    for( int i = start; i <= end; i += 2 )
     { if ( TestForPrime(i) )
         gPrimesFound++;
    }
```

Every thread will be assigned with sets of 8 indexes to work with, until all N sets are assigned.

N=(end-start)/threads

Example 7 part 2

```
#pragma omp parallel for schedule (dynamic, 10)
for ( int i = start; i <= end; i += 2 )
     { if ( TestForPrime(i) )
          gPrimesFound++;
    }
```

Each thread, at beginning, works with 10 indexes. Then, once each thread ends, is loaded with 10 more indexes until the total number is reached. It is no important to assign all threads with the same final number of indexes.

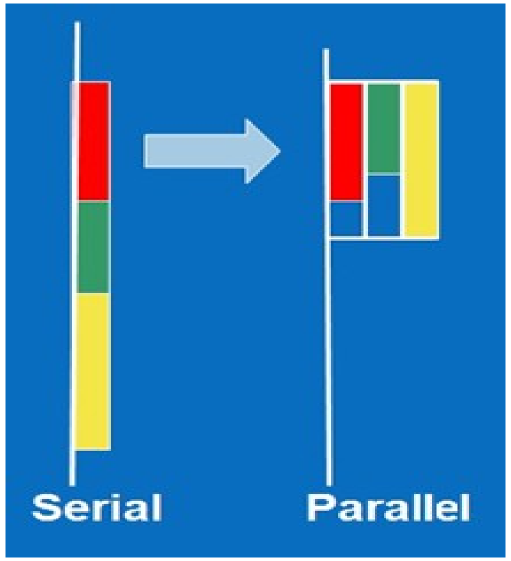
If clause

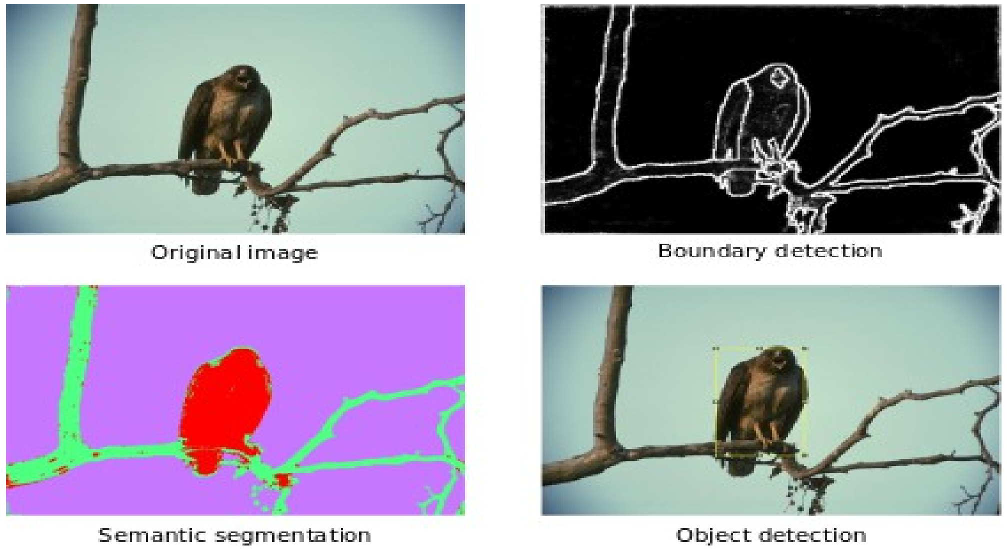
```
#pragma omp parallel for if ( n > 5000 )
for ( i= 0 ; i< n ; i++ )
      { x = (i+0.5)/n; area += 4.0/(1.0 + x*x);
      } pi = area / n;
```

In this example, parallel section is called ONLY if variable ***n*** is greater than 5000. It avoids unnecessary calculations for small sets

Functional parallelism

- In many problems, we ask threads to perform ***different*** tasks. For instance, use different filters for the same input image
- Lets imagine tasks as colours





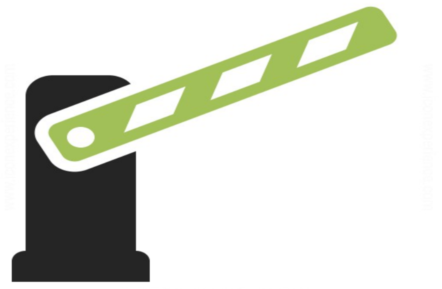
Functional parallelism

- There is the ***sections*** clause so each thread can perform its own code

```
#pragma omp parallel sections
{
    #pragma omp section
    redCode();
    #pragma omp section
    greenCode();
    #pragma omp section
    yellowCode();
}
```

Control clauses

- A few control clauses allows programmers to decide more precisely the behavior of threads
- For instance, you can block some threads because some conditions, or make all wait for some conditions



Single clause

- This clause makes ***just one thread*** to execute some part of code. It is not as ***private***, so only one executes during whole running

```
#pragma omp parallel
{
  commonPart();
  …
  commonPart();
#pragma omp single
   {
     backup();
   }  // all threads waits here
  moreCommonCode();
}
```

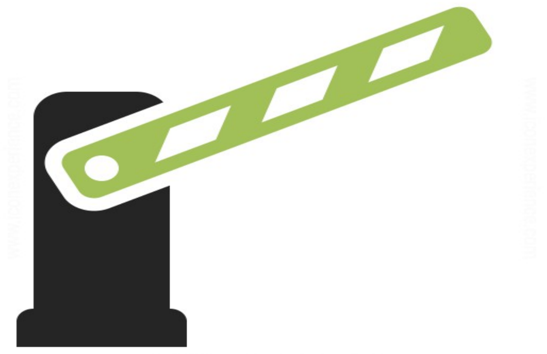
Master clause

- All threads but MASTER skips this part of code

```
#pragma omp parallel
{
  commonCode();
  …
  commonCode();
#pragma omp master
   {
     masterCode();
   }  if thread isn’t master, just skip this
  modeCommonCode();
}
```

Remarks

Single and master clauses seems to have the same behavior, but there is an ***implicit barrier*** that makes the difference.



Barrier clause

- Every thread waits until ***barrier*** is reached

```
#pragma omp parallel shared (A, B, C)
{ task1(A,B);
  printf("ALL finished task 1\n");
#pragma omp barrier
  task2(B,C);
 printf("Task 2 finished!!\n");
}
```

OpenMP library

- OpenMP has a set of ***library functions*** that helps to improve the usage of compiler clauses
- These functions makes OpenMP more powerful and versatile, bringing programmer even more control over their programs

OpenMP library

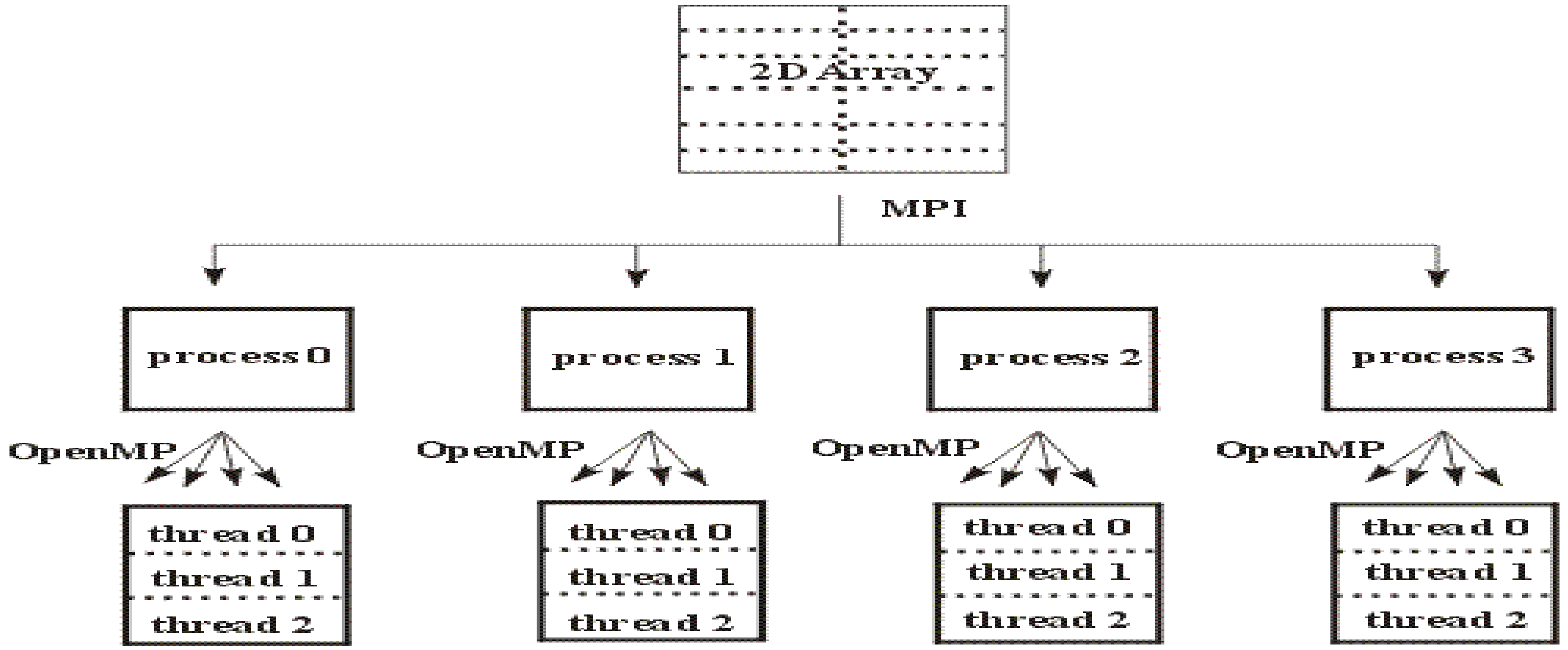
- **omp_set_num_threads**: sets the number of threads for an specified parallel section
- **omp_get_num_threads**: takes the number of actual threads running

OpenMP library

- **omp_get_max_threads**: gets the number of threads that will be created in the next parallel section
- **omp_get_thread_num**: gets the ***threadID*** of any thread (0.. N-1). Is used to assign different instructions to each thread

OpenMP – MPI

- Modern architectures allows the mixing of OpenMP and MPI in the same programs!!



Example 8

```
#include <mpi.h>
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
int i; int node,numnodes;
Int  size=32;
int main(int argc,char **argv)
  { MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&node);
    MPI_Comm_size(MPI_COMM_WORLD,&numnodes);
    MPI_Bcast(&size, 1, MPI_INT, 0, MPI_COMM_WORLD);
#pragma omp parallel
  printf(" thread %d of %d inside node %d of %d\
              nodes\n",omp_get_thread_num(),omp_\
              get_num_threads(),node,numnodes);
MPI_Finalize();
}
```

Exercise 1

- This code calculates **π**:

```
static long num_steps=100000;
double step, pi;
void main()
    { int i; double x, sum = 0.0;
      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;
     printf("Pi = %f\n",pi);
}
```

Exercise 1

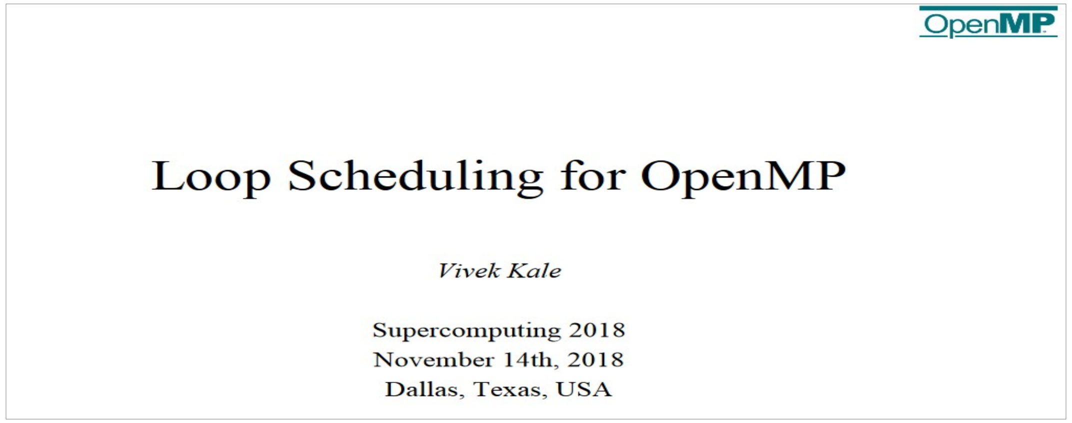
- Use OpenMP to parallelize this code, and discuss:
  - Which variables must be private or shared?
  - Must be critical sections?
  - If not, can you solve using other methods?

Exercise 2

- Consider the following: search for an element within an ***unordered*** array. Suppose we have this ***very simple code***

```
int i,found;
found=0;
for (i=0;(i<N) && (!found);i++)
     if (a[i]==element) found=1;
```

New Trends - UDS

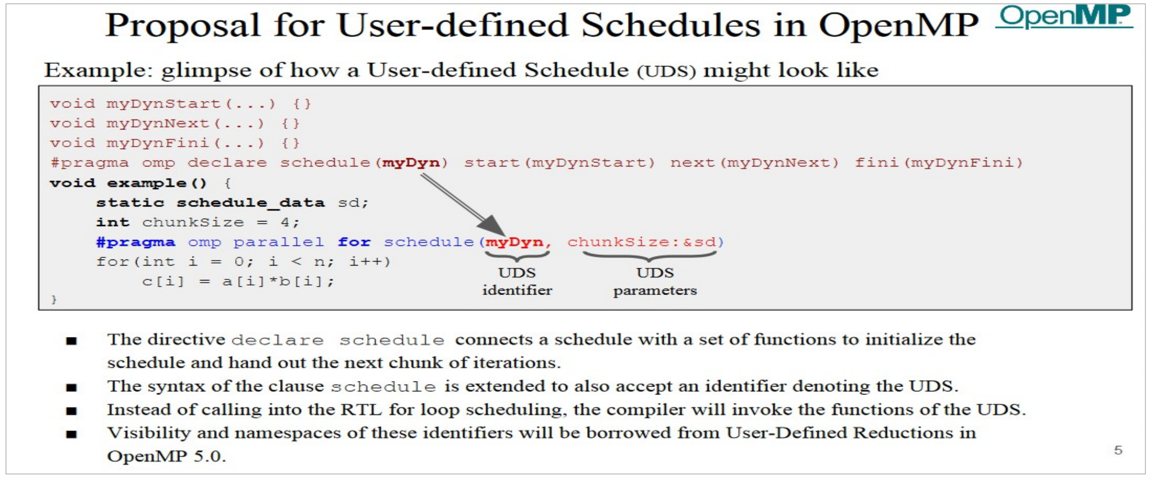


OpenMP

Loop Scheduling for OpenMP

Vivek Kale

Supercomputing 2018
November 14th, 2018
Dallas, Texas, USA



Proposal for User-defined Schedules in OpenMP

OpenMP

Example: glimpse of how a User-defined Schedule (UDS) might look like

```
void myDynStart(...) {}
void myDynNext(...) {}
void myDynFini(...) {}
#pragma omp declare schedule(myDyn) start(myDynStart) next(myDynNext) fini(myDynFini)
void example() {
    static schedule_data sd;
    int chunkSize = 4;
    #pragma omp parallel for schedule(myDyn, chunkSize:&sd)
    for(int i = 0; i < n; i++)
        c[i] = a[i]*b[i];
}
```

UDS identifier

UDS parameters

- The directive `declare schedule` connects a schedule with a set of functions to initialize the schedule and hand out the next chunk of iterations.
- The syntax of the clause `schedule` is extended to also accept an identifier denoting the UDS.
- Instead of calling into the RTL for loop scheduling, the compiler will invoke the functions of the UDS.
- Visibility and namespaces of these identifiers will be borrowed from User-Defined Reductions in OpenMP 5.0.

5

New Trends - Tasking

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

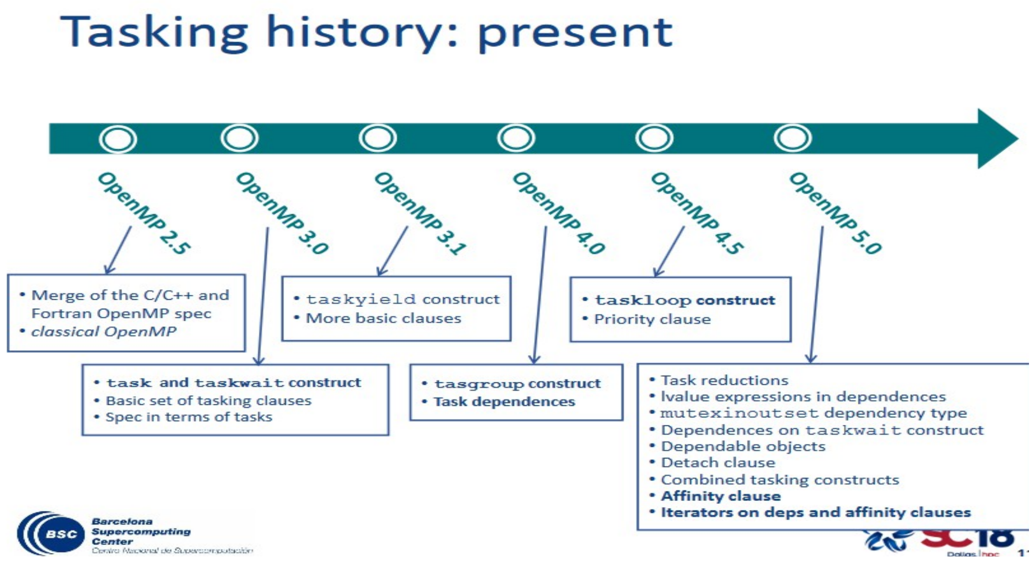
EXCELENCIA SEVERO OCHOA

OpenMP Tasking:
Past, Present and Future

Sergi Mateo Bellido
Compiler engineer

11/2018

SC18 Dallas, TX | hpc inspires.



Tasking history: present

OpenMP 2.5:
- Merge of the C/C++ and Fortran OpenMP spec
- *classical OpenMP*

OpenMP 3.0:
- `task` and `taskwait` construct
- Basic set of tasking clauses
- Spec in terms of tasks

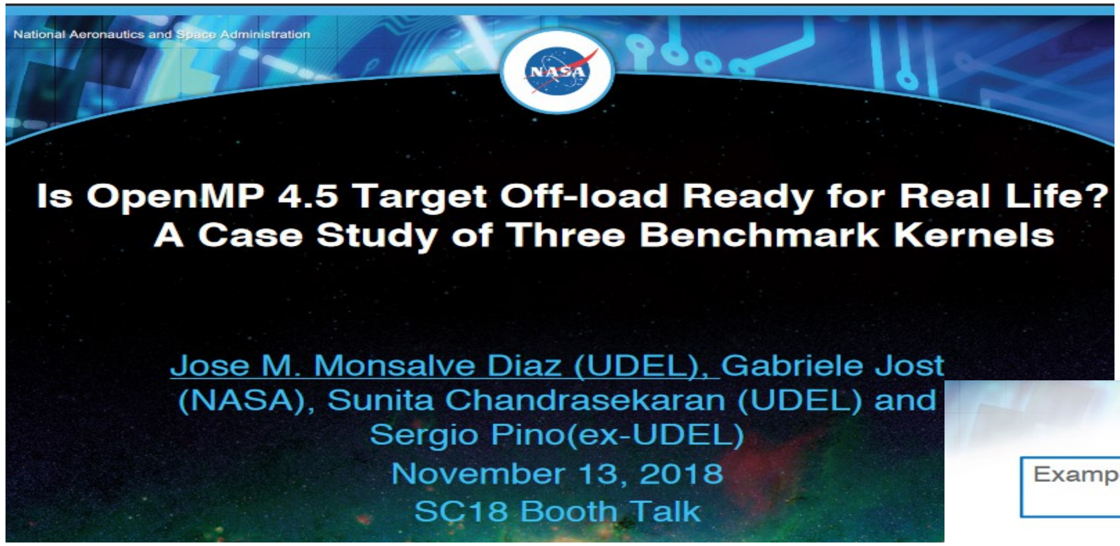
OpenMP 3.1:
- `taskyield` construct
- More basic clauses

OpenMP 4.0:
- `tasgroup` construct
- Task dependences

OpenMP 4.5:
- `taskloop` **construct**
- Priority clause

OpenMP 5.0:
- Task reductions
- lvalue expressions in dependences
- `mutexinoutset` dependency type
- Dependences on `taskwait` construct
- Dependable objects
- Detach clause
- Combined tasking constructs
- **Affinity clause**
- **Iterators on deps and affinity clauses**

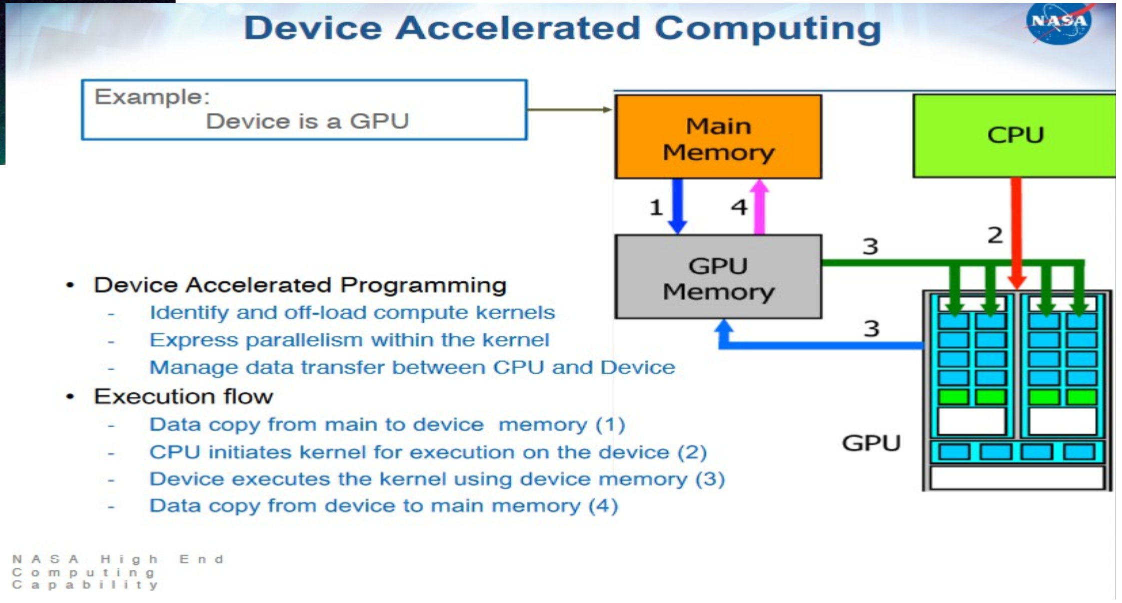
New Trends - Locality



National Aeronautics and Space Administration

Is OpenMP 4.5 Target Off-load Ready for Real Life? A Case Study of Three Benchmark Kernels

Jose M. Monsalve Diaz (UDEL), Gabriele Jost (NASA), Sunita Chandrasekaran (UDEL) and Sergio Pino(ex-UDEL)
November 13, 2018
SC18 Booth Talk



Device Accelerated Computing

Example:
Device is a GPU

- Device Accelerated Programming
  - Identify and off-load compute kernels
  - Express parallelism within the kernel
  - Manage data transfer between CPU and Device
- Execution flow
  - Data copy from main to device memory (1)
  - CPU initiates kernel for execution on the device (2)
  - Device executes the kernel using device memory (3)
  - Data copy from device to main memory (4)

NASA High End Computing Capability

12th International SuperComputing Camp 2021

Virtual

6-11 December 2021

Thanks for taking this tutorial!!!

```
har **argv) {

y_rank;
&argv);
(MPI_COMM_WORLD, &world_size);
k(MPI_COMM_WORLD, &my_rank);
```

Robinson Rivas-Suarez
robinson.rivas@ciens.ucv.ve

V-camp

```
__global__ void initialize_boundaries(real* __restrict__ const a_new, real* __restrict__ const a,
                    const real pi, const int nx, const int ny) {
    for (int iy = blockIdx.x * blockDim.x + threadIdx.x; iy < ny; iy += blockDim.x * gridDim.x) {
```